CSE517A Machine Learning **Spring 2025**

Lecture 2: Optimization for Machine Learning

Instructor: Marion Neumann

Reading: FCML Comment 4.1; LFD 3.3.2 (Gradient Descent); PML[1] Chapter 8 (Optimization)

Learning Objective

Solving the *structural risk minimization problem* entails solving an optimization problem. We want to understand the basic and most popular optimization procedures used in machine learning and be able to weigh up their advantages and disadvantages.

Our Application

Let's assume our spam filter is a **linear classifier** using the log-loss and l_1 regularization. This means we have to solve the following objective function for $\mathbf{w}$:

$$\min_{\mathbf{w}} \frac{1}{n}\sum_{i=1}^{n} \log(1 + e^{-y_i(\mathbf{w}^T\mathbf{x}_i + b)}) + \lambda \sum_{i=1}^{d} |w_i|$$

How is this classifier called?
Training this classifier essentially means to find a solution of this optimization problem. Now, we need an algorithm to do that!

1 Recap: SRM Objective

In general, the objective function of structural risk minimization as defined in the last lecture is given by:

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \min_{\mathbf{w}} \frac{1}{n}\sum_{i=1}^{n} l(h_{\mathbf{w}}(\mathbf{x}_i), y_i) + \lambda r(\mathbf{w}) \tag{1}$$

Another concrete example of this objective function stated in Eq. (1) is *regularized least squares regression*, which is also known as *ridge regression*:

$$\min_{\mathbf{w}} \frac{1}{n}\sum_{i=1}^{n} (\mathbf{w}^\top\mathbf{x}_i - y_i)^2 + \lambda \sum_{i=1}^{d} w_i^2 \tag{2}$$

Or in matrix notation using $X \in \mathbb{R}^{d\times n}$ Eq. (2) can be represented as:

$$\min_{\mathbf{w}} \frac{1}{n} \underbrace{(X^\top\mathbf{w} - \mathbf{y})^\top(X^\top\mathbf{w} - \mathbf{y})}_{=\mathbf{w}^\top XX^\top\mathbf{w} - 2\mathbf{w}^\top X\mathbf{y} + \mathbf{y}^\top\mathbf{y}} + \lambda\mathbf{w}^\top\mathbf{w} \tag{3}$$

We will be using this notation quite a bit, so try to familiarize yourself with it as best as you can. Later we will have to take derivatives, which can also be elegantly done using matrix notation.

Exercise 1.1. Derive the solution for **ols** and **ridge regression** in matrix notation (cf. Section 4 in Lecture 1: SRM to check your solution).

[1] Probabilistic Machine Learning by Kevin Murphy (https://probml.github.io/pml-book/book1.html)

How to solve the minimization problem?

We have a couple of different options here and not every one will necessarily work out for all objectives we will encounter:

(1) Closed form solution, cf. Exercise 1.1

(2) Derive constraint optimization problem (primal/dual) and solve those with a QP-solver, cf. Exercise 1.2

(3) Gradient descent (GD) and it's variants, e.g. for logistic regression (and actually most optimization problems we will encounter)

Note that we treat λ as a *hyperparameter*, that is not optimized with the model parameters (i.e., $\mathbf{w}$). Typically λ is learned via cross-validation using *telescope-search*, *random search*, or *Bayesian optimization*.

Aside: *telescope-search* for hyperparamter optimization:

(1) find coarse range for λ
(2) refine range around best choice for λ ("*zoom in*")

Exercise 1.2. The **support vector machine** (SVM) solution to the linear classification problem

$$h(\mathbf{x}) = sign(\mathbf{w}^\top \mathbf{x} + b)$$

on $D = \{(\mathbf{x}_i, y_i)\}_{i=1\ldots n}$ is the separating hyperplane that maximizes the margin, where $\mathbf{x}, \mathbf{x}_i, \mathbf{w} \in \mathbb{R}^d$ and $b, y_i \in \mathbb{R}$. (Consider the *hard margin* SVM formulation without *slack variables* in this exercise.)

(a) Derive the formula for the margin $\gamma(\mathbf{w}, b)$ of a separating hyperplane $\mathcal{H}$ as the distance between the hyperplane and the nearest training point.

(b) Now, we want to maximize this margin and ensure that $\mathcal{H}$ is indeed separating the training data. Write down the optimization problem that yields the maximum margin classifier and derive the **(primal) svm optimization problem**. Show your derivations and briefly explain every step.

(c) How many parameters (aka *variables*) does the primal optimization problem have? How many constraints?

(d) Derive the unconstrained SVM formulation (SRM objective)

$$\min_{\mathbf{w}} C \underbrace{\sum_{i=1}^{n} \max[1 - y_i(\underbrace{\mathbf{w}^T\mathbf{x}_i + b}_{=f_{\mathbf{w}}(\mathbf{x}_i)}), 0]}_{\text{hinge-loss}} + \underbrace{\|\mathbf{w}\|_2^2}_{l_2\text{-regularizer}}$$

from the primal. What is λ in this representation?

(e) Carefully derive the **dual svm optimization problem**. Include all steps of your derivation with brief verbose explanations.

(f) How many parameters (aka *variables*) does the dual optimization problem have? How many constraints?

(g) When is the primal optimization as solution to the SVM classifier disadvantageous? Discuss two situations.

(h) State the dual form of the SVM classifier used for predictions and explain why only support vectors contribute to the prediction.

2 Gradient Descent

The goal is to minimize a **learning objective** $\ell(\mathbf{w})$ such as the loss function, SRM objective $\mathcal{L}(\mathbf{w})$, or other objectives like the *negative log likelihood* or *negative log posterior*, that we will encounter later in this course. Assuming that $\ell(\mathbf{w})$ is convex, continuous, and differentiable (once or twice), there exists a **global minimum**.

To find this minimum we us the following **strategy**:
Start at a random location in the parameter space and then follow the direction of the descending gradient of ℓ as illustrated in Figure 1.

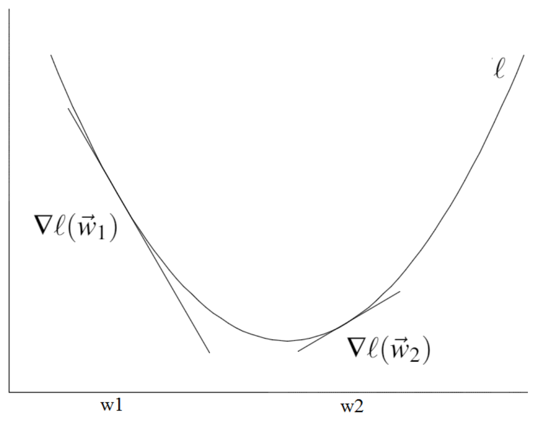


Figure 1: Gradient descent illustration plotting w versus $\ell(w)$ for a one-dimensional w and indicates the gradient of ℓ at two locations w_1 and w_2 in the parameter space.

The basic gradient descent algorithm works as follows:

Algorithm 1 Gradient Descent

```
Initialize $\mathbf{w_0}$
repeat
    $\mathbf{w}_{t+1} \longleftarrow \mathbf{w}_t + \mathbf{s}$
until $||\mathbf{w}_{t+1} - \mathbf{w}_t|| \leq \epsilon$
```

How to choose the *gradient step* $\mathbf{s}$ in Algorithm 1?
We want:

$$\ell(\mathbf{w}_{t+1}) \leq \ell(\mathbf{w}_t) \iff \ell(\mathbf{w}_t + \mathbf{s}) \leq \ell(\mathbf{w}_t) \tag{4}$$

If $||\mathbf{s}||$ is small, we can use the *Taylor approximation*:

$$\ell(\mathbf{w}+\mathbf{s}) = \ell(\mathbf{w}) + \nabla\ell(\mathbf{w})^\top\mathbf{s} + \frac{1}{2}\mathbf{s}^\top\nabla^2 l(\mathbf{w})\,\mathbf{s} + \ldots \tag{5}$$

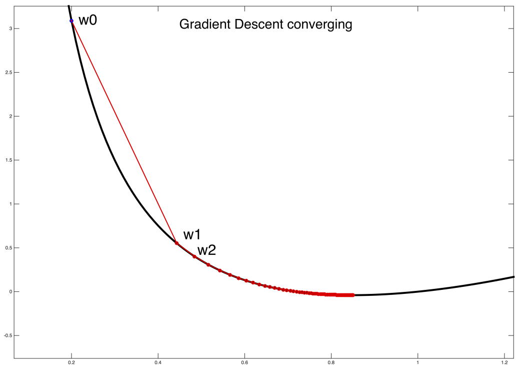
where $\nabla\ell(\mathbf{w})^\top$ is the **first order approximation** and $\nabla^2\ell(\mathbf{w})$ is the **second order approximation**. Gradient descent uses the first order approximation.
Let $g(\mathbf{w}) = \nabla\ell(\mathbf{w})$ be the vector of *partial first order derivatives*, then Equation 4 can be written as:

$$\begin{aligned} \ell(\mathbf{w}_t + \mathbf{s}) \approx \ell(\mathbf{w}_t) + g(\mathbf{w}_t)^\top\mathbf{s} \leq \ell(\mathbf{w}_t) \\ \iff g(\mathbf{w}_t)^\top\mathbf{s} \leq 0 \end{aligned} \tag{6}$$

We achieve this for example when choosing: $\mathbf{s} = -g(\mathbf{w}_t)$.
In general,

$$\mathbf{s} = -\alpha\, g(\mathbf{w}_t) \tag{7}$$

where $\alpha > 0$ is the *learning rate* or *step-size*. Setting α is a "dark art"; only if it is sufficiently small, gradient descent converges (see Figure 2 [Left]). If it is too large the algorithm can easily diverge out of control (see Figure 2 [Right]).



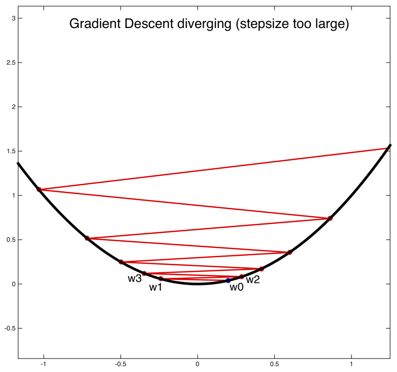


Figure 2: Left: Gradient descent converges when α is small. Right: Gradient descent may diverge when α is too large.

Some choices of the learning rate α

The choice of α is essential to a good GD implementation, cf. Figure 3.

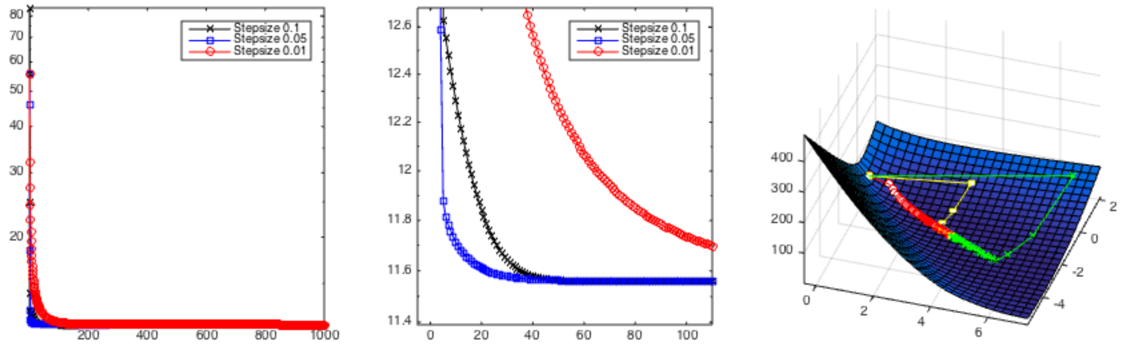


Figure 3: Gradient descent with different step-sizes. The left and middle plots show the losses as a function of the number of iterations. The right plot shows the actual learning objective for a dataset with $d = 2$ features and the path of the objective function values as we update the $2d$ weight vector $\mathbf{w}_t$ over time. The middle image shows the loss zoomed in around the left bottom corner. In this case the steps-size of $\alpha = 0.05$ (blue line left and yellow right) converges the fastest.

Here are some ways of choosing the learning rate α:
(1) **Safe choice**:

$$\alpha_t = \frac{1}{t}$$

GD will converge but very slowly.

(2) **Heuristic/ad-hoc choice**:

$$\alpha_{t+1} = \begin{cases} 1.01\,\alpha_t & \text{if } l(\mathbf{w}_{t+1}) \le l(\mathbf{w}_t) \\ 0.5\,\alpha_t & \text{if } l(\mathbf{w}_{t+1}) > l(\mathbf{w}_t) \end{cases}$$

It is not guaranteed that GD will converge.

(3) **Line search**: Choose the α that minimizes ε_{TR}.

$$\alpha^*_{t+1} = \arg\min_{\alpha} \sum_{i=1}^{n} l_{\mathbf{w}_{t+1}}(\mathbf{x}_i, y_i),$$

where $l_{\mathbf{w}_{t+1}}(\mathbf{x}_i, y_i)$ is the training error at which is not necessarily the value of the objective function we are optimizing in GD. This requires solving an additional optimization problem in every iteration of gradient descent.

Exercise 2.1. GD for OLS

(a) Why would you want to perform GD to train an OLS regression model instead of computing the closed form solution derived in Exercise 1.1?

(b) Write down the gradient descent update for the OLS problem in terms of the matrix $X := [\mathbf{x}_1\ \mathbf{x}_2\ \mathbf{x}_3\ \dots\ \mathbf{x}_n]$, the vector $\mathbf{y} := [y_1\ y_2\ \dots\ y_n]^T$ and the weight vector $\mathbf{w} \in \mathbb{R}^d$.

(c) True or false? Gradient descent on a convex function (such as the OLS objective) is guaranteed to converge. Justify your answer.

(d) Add an elastic net regularizer ($\lambda_1 \|\mathbf{w}\|_1 + \lambda_2 \|\mathbf{w}\|_2^2$ - *simple version*) to your OLS regression model. State the gradient descent update rule for this new objective function in matrix notation.

3 Newton's Method

Let's use the second order approximation in Equation (5). This is Newton's method and it assumes that the loss $\ell(\mathbf{w})$ is *twice differentiable.*
Note that we can represent $\nabla^2 \ell(\mathbf{w})$ by the **Hessian matrix**

$$H(\mathbf{w}) = \begin{pmatrix} \frac{\partial^2 \ell}{\partial w_1^2} & \frac{\partial^2 \ell}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 \ell}{\partial w_1 \partial w_n} \\ \vdots & \cdots & \cdots & \vdots \\ \frac{\partial^2 \ell}{\partial w_n \partial w_1} & \cdots & \cdots & \frac{\partial^2 \ell}{\partial w_n^2} \end{pmatrix},$$

which contains all second order partial derivatives $H_{ij} = \frac{\partial^2 \ell}{\partial \mathbf{w}_i \partial \mathbf{w}_j}$. Note that H a *symmetric* square matrix that is always *positive semi-definite* (cf. problem on hw1).

Reminder: A symmetric matrix M is *positive semi-definite* if it has only non-negative eigenvalues or, equivalently, for any vector $\mathbf{x}$ we must have $\mathbf{x}^\top M \mathbf{x} \ge 0$.

So, to choose the gradient step $\mathbf{s}$ we want $\ell(\mathbf{w} + \mathbf{s})$ to be be small. Hence,

$$\begin{aligned} \mathbf{s} &= \arg\min_{\mathbf{s}} \ \ell(\mathbf{w}) + \nabla \ell(\mathbf{w})^\top \mathbf{s} + \frac{1}{2} \mathbf{s}^\top \nabla^2 \ell(\mathbf{w})\, \mathbf{s} \\ &= \arg\min_{\mathbf{s}} \ \ell(\mathbf{w}) + g(\mathbf{w})^\top \mathbf{s} + \frac{1}{2} \mathbf{s}^\top H(\mathbf{w})\, \mathbf{s} \end{aligned} \qquad (8)$$

where $g(\mathbf{w}) = \nabla\ell(\mathbf{w})$ and $H(\mathbf{w}) = \nabla^2\ell(\mathbf{w})$.
Note that here we are optimizing over a convex parabola that represents the gradient direction and the curavture of our objective function.
To find the minimum of Equation (8), we take its first derivative and equate it with zero and solve for $\mathbf{s}$:

$$\begin{aligned} 0 &= g(\mathbf{w}) + H(\mathbf{w})\,\mathbf{s} \\ \Rightarrow \mathbf{s} &= -H(\mathbf{w})^{-1}g(\mathbf{w}) \end{aligned} \tag{9}$$

Appreciate that now we don't have a learning rate parameter α anymore, which is nice. This choice of $\mathbf{s}$ converges extremely fast if the approximation is sufficiently accurate. Otherwise it can diverge. This is typically the case if the function is flat or almost flat with respect to some input dimension. In that case the second derivatives are close to zero, and their inverse becomes very large – resulting in gigantic steps.

3.1 Avoid Divergence of Newton's Method

To avoid divergence of Newton's method, a good approach is to combine gradient descent and newton's method. Essentially, we try to get to the right place slowly but goal directed by starting off with gradient descent (or even stochastic gradient descent) and then finish the optimization quickly with Newton's method. Typically, the second order approximation, used by Newton's Method, is more likely to be appropriate near the optimum.

Gradient descent always converges after over 100 iterations from all initial starting points. If it converges (Figure 4), Newton's method is much faster (convergence after 8 iterations) but it can diverge (Figure 5). Figure 6 shows the hybrid approach of taking 6 gradient descent steps and then switching to Newton's Method. It still converges in only 10 updates.

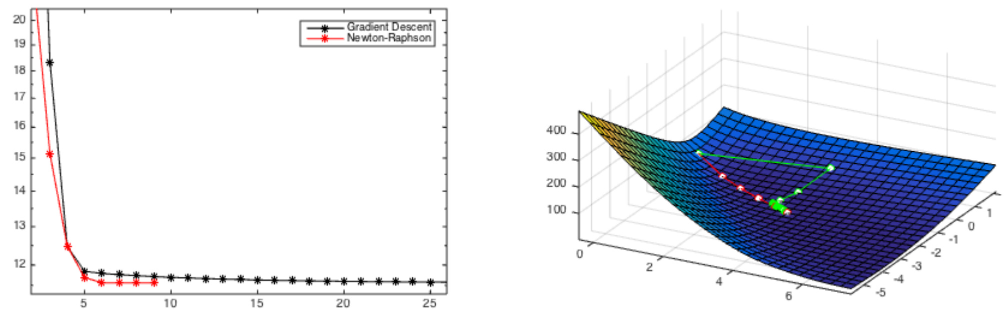


Figure 4: A starting point where Newton's Method converges in 8 iterations.

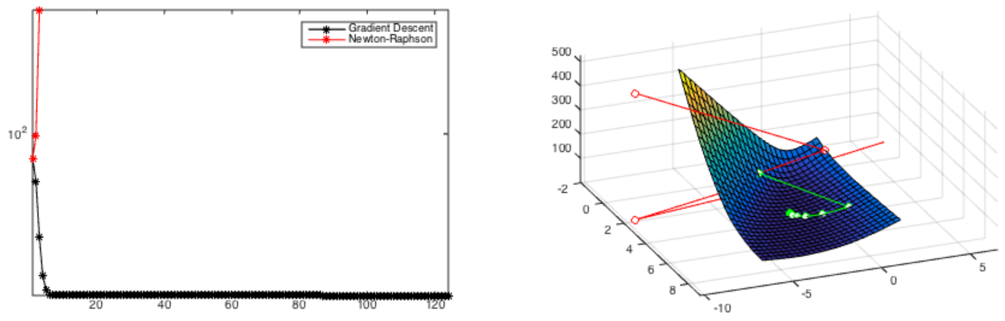


Figure 5: A starting point where Newton's Method diverges.

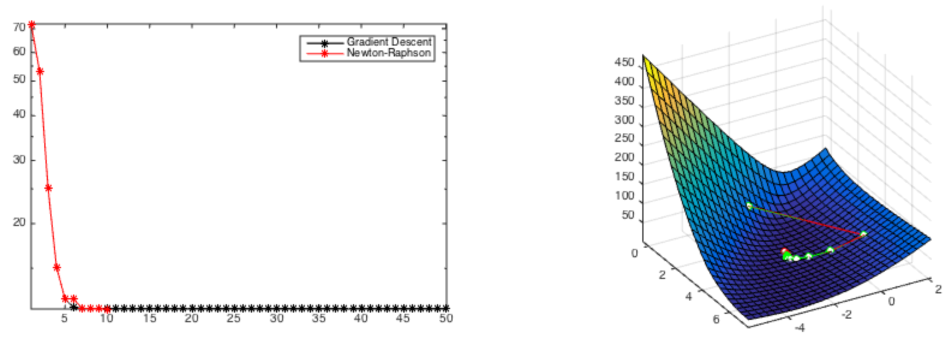


Figure 6: Same starting point as in Figure 5, however Newton's method is only used after 6 gradient steps and converges in a few steps.

3.2 Quasi-Newton Methods

Also note that computing $H(\mathbf{w})^{-1}$ is expensive ($O(d^3)$) and oftentimes we use **approximations** for the Hessian matrix or it's inverse directly. Such methods are commonly known as *quasi-Newton methods* and one of its most popular representatives is **(limited-memory) BFGS** (after Broyden-Fletcher-Goldfarb-Shanno). The easiest, but also crudest approximation is to use $\operatorname{diag}(H(\mathbf{w}))$ instead of $H(\mathbf{w})$, which is easily invertible.

3.3 [optional] Conjugate Gradient

Another way to avoid the computation of the Hessian, but still incorporating 2nd order information is to use conjugate search directions. Essentially we run GD with line search and compute the gradient step $\mathbf{s}$ (i.e., the gradient direction) based on the previously used directions. The new direction will be *conjugate* to the previously used ones. This way we end up using way less search directions, and hence GD converges faster. Algorithm 2 is the basic conjugate gradient descent CGD algorithm.

Algorithm 2 Conjugate Gradient Descent

```
Initialize $\mathbf{w}_0$, $\mathbf{d}_0 = -g(\mathbf{w}_0)$, $\mathbf{g}_1 = -g(\mathbf{w}_0)$
repeat
    $\alpha \leftarrow \arg\min_\alpha l(\mathbf{w}_t + \alpha \mathbf{d}_t)$    // line search
    $\mathbf{w}_{t+1} \longleftarrow \mathbf{w}_t + \alpha\, \mathbf{d}_t$
    $\mathbf{g}_t = \mathbf{g}_{t+1}$, $\mathbf{g}_{t+1} = -g(\mathbf{w}_{t+1})$    // remember old gradient, compute new gradient
    $\mathbf{d}_{t+1} = \mathbf{g}_{t+1} + \beta\, \mathbf{d}_t$    // new conjugate direction
until $||\mathbf{w}_{t+1} - \mathbf{w}_t|| \le \epsilon$
```

Note that $\mathbf{g}_t$ is a vector and there are various possible ways to set β, e.g. $\beta = \frac{\mathbf{g}_{t+1}^\top(\mathbf{g}_{t+1}-\mathbf{g}_t)}{\mathbf{d}_{t+1}^\top(\mathbf{g}_{t+1}-\mathbf{g}_t)}$ (after Hestenes-Stiefel) with dot products in numerator and denominator. Unfortunately, CGD does not work with *noisy gradients* (which are produced by stochastic gradient descent discussed below).

Exercise 3.1. True or false? Justify your answer.

(a) Ordinary least squares regression can be solved with Newton's method.

(b) Newton's Method can be used to optimize the (standard) SVM hinge loss.

Exercise 3.2. If you were to optimize the OLS objective with Newton's method, how many steps would you need until convergence (you can assume the Hessian is invertible)? You can either derive the answer formally or state it verbally with clear justifications.

4 Best Practice

If your data is not too big (approx. $n \leq 100,000$), then quasi-Newton or conjugate gradient descent are the methods of choice. If you have a huge number of training points, then we typically use *approximated gradients* in combination with an *adaptive learning rate*. In the following we discuss these two strategies.

4.1 Momentum Method

When dealing with non-convex functions, e.g. in neural network training, we want to **avoid local minima**. One simple way to do so is to have an adaptive learning rate that uses part of the previous gradient:

$$\mathbf{s} = -\alpha \tilde{g}(\mathbf{w}_t) \tag{10}$$

where $\tilde{g}(\mathbf{w}_t) \leftarrow g(\mathbf{w}_t) + \mu \tilde{g}(\mathbf{w}_{t-1})$. This is known as the momentum method and its idea is to use some portion of the previous gradient ("*momentum*") to push you out of small local minima.
There are various other ways of computing adaptive learning rates, such as ADAM or RMSPROP. See this blog post for more information: http://ruder.io/optimizing-gradient-descent/index.html#whichoptimizertochoose.

4.2 Stochastic Gradient Descent

The idea of stochatstic gradient descent (SGD) is to make GD more **efficient**. We consider $l(\mathbf{w})$ on a subset of training examples. In practice, SGD converges faster than GD; and noisy updates can help escape local minima. Two versions are typically considered:

- use one training point at a time
  $g(\mathbf{w}) = \frac{\partial l(\mathbf{x}_i, y_i)}{\partial \mathbf{w}}$ (very high fluctuations in the objective function)
- mini-batches (randomly partition D in sets with m training examples)
  $g(\mathbf{w}) = \sum_{i=1}^{m} \frac{\partial l(\mathbf{x}_i, y_i)}{\partial \mathbf{w}}$ for $m << n$

SGC is extremely popular as it is easy to implement and fast for large n. As setting the learning rate is challenging we typically combine SGC with the momentum method (or other methods using adaptive learning rate, such as ADAM). Note that there are parallel implementations of SGC, however, there is still a lot of ongoing research in this area.

4.3 Note: Random Restarts for Non-convex Functions

Oftentimes our objective functions are non-convex. We will still use the methods discussed in this lecture to optimize those objectives. The solution we get is now typically not the *global* minima, but a *local* one. so, when dealing with non-convex functions we typically use several randomly selected starting points and choose the best solution among the (S)GD results. This is then called GD with **random restarts**. Note that there is a trival way of parallelizing this algorithm.

5 Summary

We discussed how to optimize convex, continuous, and differentiable functions using gradient descent and Newton's method. Further, we introduced several variations of those methods and a couple of best practice tips on which methods to use. **You should be familiar with the following methods:**

- gradient descent (and it's gradient step derivation)
- Newton's method
- momentum method
- stochastic gradient descent

Exercise 5.1. Practice Retrieving!
For this summary exercise, it is intended that your answers are based on **your own** (current) understanding of the concepts (and not on the definitions you read and copy from these notes or from elsewhere). Don't hesitate to **say it out loud** to your seat neighbor, your pet or stuffed animal, or to yourself before **writing it down**. Research studies show that this practice of retrieval and phrasing out loud will help you retain the knowledge!

(a) Using *your own words*, summarize each of these methods in 2-3 sentences by retrieving the knowledge from the top of your head.

(b) What is the difference between GD and Newton? Name one advantage of GD over Newton. Name one advantage of Newton over GD.

(c) Recap the assumptions for GD.

(d) Discuss the implications of those assumptions being violated. Consider theoretical and practical implications. What can you do (if anything) in the case where each of the assumptions is violated? Consider one at a time and come up with concrete things that can be done (if anything can be done) to deal with the violation.

(e) How does the momentum method improve upon vanilla GD?

(f) How does stochastic GD improve upon the vanilla version?

And always remember: It's not bad to get it wrong. *Getting it wrong is part of learning!* Use your notes or other resources to get the correct answer or come to our office hours to get help!

Our Application

With respect to our **spam filter application**, we are now able to solve the various possible objective functions. That means we are able to *train* various learning models and try to find the best one for a given application/dataset.

This is exactly what you will do in **written homework 1** and **implementation project 1**: you will compute the derivatives of some desired SRM objective functions, implement the gradient descent algorithm, and evaluate various models on a dataset of emails labeled as *spam* and *ham* (not spam).