Optimization for Learning - FRTN50
Assignment 3

Introduction

In this assignment we will look at different variants of proximal gradient methods. We'll mainly focus on the key concepts behind solving large scale problems. Practical concerns and concepts that facilitate their implementations will also be touched upon.

When problem sizes grow the cost of evaluating gradients grows with it, making gradient descent based methods more and more expensive. Current research is therefore directed towards using cheap approximate gradient evaluations instead of full gradients. This is the idea behind coordinate gradient, stochastic gradient, and their many variants. In this assignment you will implement variants of both.

The assignment consists of two main parts. The first will cover and compare some algorithms for solving the standard composite optimization problem we have previous been working with,

$$\min_{x \in \mathbf{R}^n} f(x) + g(x). \tag{1}$$

The second will cover the finite sum optimization problem commonly found in model fitting, function approximation and supervised learning tasks,

$$\min_{x \in \mathbf{R}^n} \tfrac{1}{N} \sum\nolimits_{i=1}^{N} f_i(x) \tag{2}$$

Composite Optimization

Accelerated Proximal Gradient

Ordinary proximal gradient computes the next iterate based only on the current iterate, completely forgetting the trajectory. However, the trajectory can contain useful information about the function. Accelerated methods (also called momentum methods) store some information regarding the previous iterates and extrapolate based on this information. There are a couple different ways of doing this but here we choose the following Nesterov acceleration approach,

$$\begin{aligned} y_k &= x_k + \beta_k(x_k - x_{k-1}) \\ x_{k+1} &= \operatorname{prox}_{\gamma g}(y_k - \gamma \nabla f(y_k)) \end{aligned}$$

with $x_{-1} = x_0$. The sequence β_k is here a design choice and several different choices have been presented in literature, a common choice is

$$\beta_k = \frac{t_k - 1}{t_{k+1}}, \quad t_0 = 1, \quad t_{k+1} = \frac{1 + \sqrt{1 + 4(t_k)^2}}{2}. \tag{3}$$

Written by: Martin Morin
Latest update: October 9, 2020

If f is μ-strongly convex the following choice can be made

$$\beta_k = \frac{1 - \sqrt{\mu\gamma}}{1 + \sqrt{\mu\gamma}}. \tag{4}$$

Coordinate Gradient Descent

If the proximable-term g in (1) is separable, , it is possible to perform coordinate-wise updates. With $g(x) = \sum_{i=1}^{n} g_i((x)_i)$ where $(\cdot)_i$ denotes the ith element, coordinate gradient descent can be written as

$$\begin{aligned}
&\text{Sample } i \text{ uniformly from } \{1, ..., n\} \\
&(x_{k+1})_i = \text{prox}_{\gamma_i g_i}((x_k)_i - \gamma_i(\nabla f(x_k))_i) \\
&(x_{k+1})_j = (x_k)_j, \quad \forall j \neq i
\end{aligned}$$

where $x_k \in \mathbf{R}^n$. Index i could be selected in a number of different ways but here we will simply sample it uniformly and independently in each iteration.

In general, coordinate gradient requires more iterations compared to standard proximal gradient but the idea is to make each iteration much cheaper. If one coordinate of ∇f can be computed m times cheaper compared to the full gradient, and the total number of iterations required to achieve a given precision is less than m times more, a real world speed up will be achieved.

One such case where coordinate computation is cheap is when f is a quadratic, $f(x) = \frac{1}{2}x^T Qx + q^T x$. One coordinate can be evaluated up to n times cheaper than a full gradient since the coordinates of $\nabla f(x)$ are independent of each other. This can be seen from the expression of the gradient,

$$\nabla f(x) = Qx + q = (Q_1 x + (q)_1, \ldots, Q_n x + (q)_n)$$

where Q_i is the i:th row of Q. More general assumptions for cheap coordinate evaluations exist but for the purposes of this assignment the quadratic is sufficient.

One other benefit of coordinate descent is that it is possible to use coordinate-wise step-sizes, γ_i, for coordinate gradient. This allows the step-size to be adapted to the curvature/smoothness in each direction instead of defining a step-size based on the global smoothness constant. This can greatly reduce the number of iterations needed.

Support Vector Machines

We will compare the algorithms presented above on a *support vector machine* (SVM) problem.

Given training data $x_i \in \mathbf{R}^n$ with corresponding class labels $y_i \in \{-1, 1\}$ for $i \in \{1, \ldots, N\}$, an SVM is a classifier on the form $y_i \approx \text{sign}(w^T \phi(x_i))$ where ϕ is some non-linear *feature map* chosen beforehand. The parameter w is chosen by solving the following problem

$$\min_w h(YX^T w) + \tfrac{\lambda}{2}\|w\|_2^2$$

where $X = [\phi(x_1), \phi(x_2), \ldots, \phi(x_N)]$, $Y = \text{diag}(y_1, y_2, \ldots, y_N)$, λ is a tunable regularization parameter and h is the hinge loss

$$h(z) = \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - z_i).$$

Since the non-smooth term is not proximable, this problem is easiest solved via the dual

$$\min_{\nu} h^*(\nu) + \frac{1}{2\lambda} \|-XY\nu\|_2^2.$$

The last term in the dual problem is a quadratic function which we define as

$$f(\nu) = \frac{1}{2\lambda} \|-XY\nu\|_2^2 = \frac{1}{2\lambda} \nu^T Y X^T X Y \nu = \frac{1}{2} \nu^T Q \nu$$

where the matrix Q is given by $Q_{ij} = \lambda^{-1} y_i \phi(x_i)^T \phi(x_j) y_j$. The matrix Q can be evaluated by only evaluating the *kernel*, $K(x, y) = \phi(x)^T \phi(y)$, associated with the feature map ϕ. It is also possible to evaluate the model itself with only the kernel and dual solution. This means that it is possible to build a SVM classifier without explicitly using the feature map. We will not go into details regarding kernels and only note that it is the standard way of implementing SVMs since the implementation no longer depends on the dimension of the feature map. Kernels therefore allows for the use of more interesting feature maps with very high dimension. For example, one of the most commonly used kernels is the Gaussian RBF kernel whose corresponding feature map is infinit dimensional.

Tasks

Task 1 Find the convex conjugate of h. Also, given a solution to the dual SVM problem, $\nu^\star$, find the primal SVM solution $w^\star$.

Task 2 For the dual SVM problem above, compare proximal gradient (PG), accelerated proximal gradient (APG) and coordinate proximal gradient (CG). The file `svm_problem.jl` contains a function for generating objective functions, problem parameters and initial dual point ν_0.

Use the same initial point, ν_0, for all algorithms. For PG and APG, use a step-size of $\gamma = \frac{1}{L}$ where L is the smoothness-constant of f. Compare both APG with (3) and (4). For CG, compare both the coordinate-wise step-size of $\gamma_i = \frac{1}{Q_{ii}}$ and the uniform choice $\gamma_i = \frac{1}{L}$. The element Q_{ii} is here the i:th diagonal element of the Q matrix of the function f.

Plot $\|\nu_k - \nu^\star\|$ where $\nu^\star$ is the solution. Find $\nu^\star$ by simply solving the problem to extra high precision before generating your plot. For CG, scale the iteration axis with $\frac{1}{N}$ to normalize the computational cost. We assume the evaluation cost of the proximal operator is negligible and the computational cost is dominated by vector multiplication. Each iteration of CG should then be $\frac{1}{N}$th as expensive as PG, so N iterations of CG correspond to the same computational effort as one iteration of PG.

Which method required the least amount of computational effort? Which method required the least amount of iterations? Which was fastest in real time? Can you comment on the similarities/differences between real time performance

and number of iterations needed? How fair is it to compare real time performance? Can it be easily affected? For APG, did the choice of beta sequence have a large or small effect on the result? For CG, was it beneficial to use coordinate wise step-sizes?

Hints:

- `ProximalOperators.jl` expects array inputs and returns arrays, even if they are one dimensional.
- Be mindful of the code inside your update loop. Any operation involving the full iterate, ν_k, is likely to add a considerate amount of computational time to each iteration. This includes any type of logging, printing and convergence checking. It is therefore advisable to only do this every 1000-10000 iteration.
- The macro `@time` can be used to time a function call in `Julia`.

Finite Sum Optimization

Stochastic Gradient Descent

One of the most commonly used families of methods to solve problem (2) are stochastic gradient (SG) methods. The most basic SG method is to, given some initial point x_0, perform

$$\begin{aligned} &\text{Sample } i \text{ uniformly from } \{1, ..., N\} \\ &x_{k+1} = x_k - \gamma_k \nabla f_i(x_k). \end{aligned}$$

where $\gamma_k > 0$. Similarly to CG, the index i can be selected in various different ways but we choose to restrict ourselves to this uniformly random case.

The main benefit of SG methods is that they are N times cheaper per iteration compared to ordinary gradient methods. However, ∇f_i and $\nabla \frac{1}{N} \sum_{i=1}^{N} f_i$ do not need to be well correlated so each iteration does not necessarily bring x_{k+1} closer to the solution. Even in the convex case the sequence of step-sizes, $\{\gamma_k\}_{t=1,2,...}$, must be chosen with care in order to guarantee convergence. Usually this means that $\{\gamma_k\}_{t=1,2,...}$ must go to zero sufficiently fast but not too fast. In this assignment we will use step-sizes on the form

$$\gamma_k = \frac{\gamma}{1 + \beta k}$$

where $\gamma > 0$ and $\beta \geq 0$.

Model Fitting

In the coming tasks we will look at a couple of different model fitting/function approximation/supervised learning problems.

$$\min_{p \in \mathbf{R}^n} \frac{1}{N} \sum_{i=1}^{N} l(m(x_i; p), y_i)$$

where for $i \in \{1, \ldots, N\}$ the variables x_i and y_i are the input and output data of the model/function/mapping we want to fit/find/learn. The function m is the model and it is parameterized by the variable p. The output of the model is compared to the expected output data y_i and the error is penalized with the loss function l.

Automatic Differentiation

Since we will not need to calculate any proximal operators in either the above stated SG or ADAM method we will not be using `ProximalOperators.jl`. Instead we will be using `Flux.jl` and its *Automatic Differentiation* (AD) package called `Zygote.jl`. The use of an AD package will allow us to use neural networks and other (almost) arbitrary functions as models.

AD is exactly what it sounds like, it is an automated way to calculate gradients of functions you have implemented in some programming language. The now classic backpropagation algorithm for neural network is an example of an AD method. Efficient AD implementations are an integral part of machine learning libraries like `TensorFlow`, `PyTorch` and `Flux.jl`.

For simplicity you will not interact with `Flux.jl` directly, instead we have provided a wrapper for setting set up your optimization problems and calculate gradients. The wrapper is very simple, it mainly gathers and vectorizes all parameters in order to provide easy-to-work-with arrays.

One final note on AD. We said it calculates *gradients* and you will use stochastic *gradient* methods. However, AD does not necessarily calculates gradients since the model f and loss l are not necessarily differentiable everywhere and can contain kinks and discontinuities. AD algorithms will in these cases return something closer to a one-sided derivative. This has consequences with regards to algorithm design since even in the convex case we usually need to guarantee some form of local smoothness in order to prove convergence. In practice have SG methods still proven themselves to be useful so we (and the majority of the machine learning community) will for now not make this distinction and simply call everything gradients.

Tasks

In order to perform the following tasks you need to install `Flux.jl` by running the following in the `Julia` REPL.

```
using Pkg
Pkg.add(PackageSpec(name="Flux", version="0.10.4"))
```

Note that we need a specific version of `Flux.jl`.

The wrapper mentioned above is found in the file `flux_wrapper.jl`. The file `stoch-grad.jl` contains the skeleton code you will use in this part of the assignment. It uses ordinary gradient descent to fit a 2nd order polynomial model to a 5th order model. A 2nd order model is of course not enough to fit the 5th order model well, but it is enough for our purposes. Furthermore, the resulting optimization problem is a convex least squares problem so convergence

is guaranteed is guaranteed for a small enough step-size. Read the code and make sure you understand it.

Run the code and make sure that the distance to the solution approaches zero and that the mean loss decrease. `Flux.jl` uses 32-bit floating point numbers instead of the more standard 64-bit numbers. This means that the numerical precision is around only 10^{-6} to 10^{-9}. This loss of precision is a trade-off made in order to reduce the memory requirement of large neural networks. The benefit of larger networks usually outweighs the loss of numerical precision in many modern machine learning problems.

Task 3 Add a stochastic gradient implementation to `stoch-grad.jl` and solve the existing model fitting problem with it. Use $\gamma = 0.01$ and $\beta = 0$, i.e., no step-size decay, and run for $50N$ iterations. How close to the solution do you get? Can you get closer to the solution if you let the algorithm run longer? On average, how close can you get? Redo the experiment with $\gamma = 0.0001$. How does the step-size affect how close to the solution you can get?

Task 4 Set $\gamma = 0.01$ and $\beta = 0.1$ and run the SG algorithm on the existing model fitting problem for $100N$ iterations. This decaying step-size guarantees convergence for this smooth convex problem. But, how close to the solution do you get? Run for $100N$ iterations more. How much closer to the solution did you get? Rerun the experiment with a slower decay speed of $\beta = 0.01$ and compare the results. Which β performs best? Based on the results in this and the previous task, what is the risks of having too small or too large decay speed?

Task 5 To get a better fitting model, increase the order of the model in the `get_poly_model()` function to a 5th order model. [1] Run SG with $\gamma = 0.001$ and $\beta = 0$ for $100N$ iterations. Look at both the mean loss and distance to the solution, has the algorithm reached numerical precision? Reset the problem and instead run ordinary gradient descent for 100 iterations with $\gamma = 0.009$. Which algorithm performs best? In this case, both are actually guaranteed to converge but why is it fair to compare 100 iterations of GD against $100N$ iterations of SG.

Task 6 Change from the polynomial model to the neural network model. Run gradient descent for 100 iterations with $\gamma = 0.02$. Does it appear to converge? Reset the problem and run SG for $100N$ iterations with $\gamma = 0.001$ and $\beta = 0$. How is the fit compared to gradient descent? Run for an additional $100N$ iterations. Does the mean loss improve? Both for the neural network and the 5th order model we used no decay on the step-size, $\beta = 0$. What differentiate the neural network and 5th order models from the 2nd order model with regards to the model we are trying to learn?

Hints:

- Remember to redefine the model/network and algorithm between experiments so that you start from scratch.

[1] We will here ignore issues regarding overfitting. However, in a real world scenario overfitting is of course an important problem to handle.

- It is often advantageous for an optimization algorithm to able to be resumed after an initial number of iterations have been performed. For this, your SG implementation needs to keep track of its state, i.e., the decaying step-size.

Remark: Stochastic Gradient in Practice SG variants are the de facto standard methods for fitting large neural networks. Arguably, the most popular variant is ADAM which incorporates variants of both acceleration/momentum and coordinate scaling. However, the main drawback of ADAM is that it is not guaranteed to converge, not even for smooth convex problems. One of the reasons for this is that the step-size does not necessarily go to zero. Still, ADAM have shown to perform well on a huge amount of deep learning problems.

SG methods often use some form of batching and sampling without replacement that further improves the stochastic approximation of the gradient. Batching is simply to sample several data points and calculate the mean loss and gradient w.r.t. to all of those data points instead of only one. Sampling without replacement no longer samples the data points independently but samples only from the data points not yet used. Once all data points have been used the sampling is reset and started over again. This epoch based way of sampling prevents some data points from being used too often and forces all data to be used regularly.

Submission

Your submission should contain the following.

- All your code.
- A single pdf where you answer and/or discuss the questions raised in each task.

Use plots, figures and tables to motivate your answers.