5 Splitting Algorithms

Let's start with a motivating example (from "Learning with optimal interpolation norms" by Combettes, McDonald, Miccheli, & Pontil, in *Numerical Algorithms*, 2019).

Example 5.1 (linear SVM training) Given two datasets $\mathcal{D}_1, \mathcal{D}_2 \subset \mathcal{H}$, let's consider the problem of training a sparse linear separator

$$\underset{x\in\mathcal{H}}{\text{minimize}} \quad \left(\sum_{d\in\mathcal{D}_1} \max\{0, 1-\langle d \mid x\rangle\}\right) + \left(\sum_{d\in\mathcal{D}_2} \max\{0, 1+\langle d \mid x\rangle\}\right) + +\lambda\|x\|_1. \tag{73}$$

A solution, x^*, to (73) is used to *classify* unobserved data u by looking at the sign of a scalar product: if $\langle u \mid x^*\rangle > 0$, we predict $u \in \mathcal{D}_1$; for $\langle u \mid x^*\rangle < 0$, we predict $u \in \mathcal{D}_2$. The first two sums are composed of **hinge loss** functions based on our two datasets. The final term $\|\cdot\|_1$ is used to promote **sparsity** of x. Loosely speaking, x is "sparse" if it has a small number of nonzero components. The larger value of λ, the more sparse x^* becomes. Once a sparse solution x^* is found, this makes classification predictions very efficient (particularly in high-dimensional settings), because we only have to look at the nonzero components of x^* in order to compute $\langle u \mid x^*\rangle$.

If we look at the libraries in Remark 4.11, the objective function in Example 5.1 does not appear anywhere! We can find the prox of an individual hinge loss and the prox of $\|\cdot\|_1$. However it looks like we can't compute the prox of the sum! This exemplifies a more general problem, an instance of which is summarized below.

Question 5.2

Given $f, g \in \Gamma_0(\mathcal{H})$ and access to compute Prox_f *and* Prox_g*, can we efficiently compute* $\operatorname{Prox}_{f+g}$*?*

If a generic answer to Question 5.2 were available, we could just apply the Proximal Point Algorithm (Theorem 4.15) to declare victory. For some settings, we have a positive answer to this question:

Example 5.3 Let U and V be orthogonal vector subspaces of $\mathcal{H}$. Then (as demonstrated in class)

$$\operatorname{Prox}_{\iota_U+\iota_V} = \operatorname{Prox}_{\iota_{U\cap V}} = \operatorname{Proj}_{U\cap V} = \operatorname{Proj}_U \circ \operatorname{Proj}_V = \operatorname{Prox}_{\iota_U} \circ \operatorname{Prox}_{\iota_V}. \tag{74}$$

Hence, for this setting, the prox of the sum is expressed as the composition of prox operators.

However, outside of special instances like Example 5.3, a satisfactory answer to Question 5.2 has not been found. Therefore, the research community has circumvented this issue by producing algorithms which converge to minimizers of $(f+g)$ without requiring an answer to Question 5.2. Loosely speaking, this class of algorithms are known as *splitting algorithms*, and their hallmark characteristic is that they *only rely on operators associated with the individual summands of the objective function*. Below, we present the Forward-Backward algorithm, which minimizes a sum of a smooth function $g \in \Gamma_0(\mathcal{H})$ and another (potentially nonsmooth) function $f \in \Gamma_0(\mathcal{H})$. Instead of requiring Prox_{f+g}, we only rely on Prox_f and evaluating ∇g. The specific form below comes from Proposition 28.13 in Bauschke/Combettes' book.

Theorem 5.4 (Forward-Backward Algorithm) *Let $f \in \Gamma_0(\mathcal{H})$, let $L > 0$, and let $g \in \Gamma_0(\mathcal{H})$ be L-smooth. Let $\gamma \in]0, 2/L[$ and set $\delta = 2 - \gamma L/2$. Let $(\lambda_n)_{n\in\mathbb{N}}$ be a sequence in $[0, \delta]$ such that $\sum_{n\in\mathbb{N}} \lambda_n(\delta - \lambda_n) = +\infty$ and let $x_0 \in \mathcal{H}$. Suppose $(f + g)$ has a minimizer. Iterate*

```
for $n = 0, 1, \dots$
    $y_n = y_n - \gamma\nabla g(x_n)$                                  # Gradient (forward) step        (75)
    $x_{n+1} = x_n + \lambda_n(\text{Prox}_{\gamma f} y_n - x_n)$.     # Prox (backward) step
```

Then the following hold:

(i) $(x_n)_{n\in\mathbb{N}}$ *converges to a minimizer of* $f + g$.

(ii) $(f(x_n) + g(x_n)) - \inf_{x\in\mathcal{H}} f(x) + g(x) \searrow 0$.

The proof idea follows from the Fejer machinery discussed in Remark 4.16. We can provide an intuition which demonstrates that fixed-points of the Forward-backward operators in (75) are minimizers of $f + g$. Since g is differentiable, we can use the sum rule. By Fermat's rule, we have

$$x \text{ minimizes } f + g \Leftrightarrow 0 \in \partial(f + g)(x) \tag{76}$$
$$\Leftrightarrow 0 \in \partial f(x) + \partial g(x) \tag{77}$$
$$\Leftrightarrow 0 \in \partial f(x) + \{\nabla g(x)\} \tag{78}$$
$$\Leftrightarrow -\nabla g(x) \in \partial f(x). \tag{79}$$

Multiplying by γ and adding x, then using our characterization from Exercise 4.6 we find, for every $n \in \mathbb{N}$,

$$x \text{ minimizes } f + g \Leftrightarrow x - \gamma\nabla g(x) \in \partial\gamma f(x). \tag{80}$$
$$\Leftrightarrow x = \text{Prox}_{\gamma f}(x - \gamma\nabla g(x) \tag{81}$$
$$\Leftrightarrow \lambda_n(\text{Prox}_{\gamma f}(x - \gamma\nabla g(x)) = 0 \tag{82}$$
$$\Leftrightarrow x + \lambda_n(\text{Prox}_{\gamma f}(x - \gamma\nabla g(x)) = x, \tag{83}$$

i.e., x is a fixed-point of the algorithm (75).

Example 5.5 Let C be a closed convex subset of $\mathcal{H}$, let $f = \iota_C$, and let $\lambda_n \equiv 1/L$. Then, it follows from Example 4.5 that the popular **projected gradient descent algorithm**

$$(\forall n \in \mathbb{N}) \quad x_{n+1} = \text{Proj}_C(x_n - \gamma\nabla g(x_n)) \tag{84}$$

is a special case of the Forward-Backward algorithm (Theorem 5.4).

Example 5.6 (LASSO (Tibshirani)) The Forward-backward algorithm can be applied to solve the LASSO problem,

$$\underset{z\in\mathcal{H}}{\text{minimize}} \;\; \|Ax - b\|^2 + \|x\|_1. \tag{85}$$

However, let us suppose that we must to only use proximity operators – for instance, either I must (because all of my functions are nonsmooth, as in Example 5.1), or I have read some literature telling me that I may get better convergence behavior[5]. Even for the LASSO problem, we could compute this prox:

Exercise 5.7 Let $A\colon \mathbb{R}^n \to \mathbb{R}^m$ be a matrix and let $b \in \mathbb{R}^m$. Compute the proximity operator of

$$\frac{1}{2}\|A(\cdot) + b\|^2.$$

hint: Proposition 3.6 and Exercises 4.6 and 3.5.[6]

One of the standard "prox-only" splitting algorithms for solving

$$\underset{x\in\mathcal{H}}{\text{minimize}} \;\; f(x) + g(x) \tag{86}$$

is the following (whose form is simplified from Corollary 28.3 in the book).

Theorem 5.8 (Douglas-Rachford algorithm) *Let $f, g \in \Gamma_0(\mathcal{H})$ such that a minimizer of $(f+g)$ exists, let $(\lambda_n)_{n\in\mathbb{N}}$ be a sequence in $[0,2]$ such that $\sum_{n\in\mathbb{N}} \lambda_n(2-\lambda_n) = +\infty$, and let $\gamma > 0$. Let $x_0 \in \mathcal{H}$ and set*

```
for $n = 0, 1, \ldots$
    $y_n = \text{Prox}_{\gamma g} x_n$
    $z_n = \text{Prox}_{\gamma f}(2y_n - x_n)$            (87)
    $x_{n+1} = x_n + \lambda_n(z_n - y_n)$.
```

Then $(x_n)_{n\in\mathbb{N}}$ converges to a minimizer of $f + g$.

Below is an indication as to why this algorithm works.

Exercise 5.9 (optional) Let $R_1 = 2\text{Prox}_{\gamma f} - \text{Id}$ and let $R_2 = 2\text{Prox}_{\gamma g} - \text{Id}$. Show that the Douglas-Rachford algorithm is of the form

$$x_{n+1} = x_n + \frac{\lambda_n}{2}\left(R_1(R_2 x_n) - x_n\right). \tag{88}$$

From Exercise 5.9, we can show an indication as to why fixed points of the Douglas-Rachford algorithm yield solutions of (86). Let $x \in \mathcal{H}$ and $n \in \mathbb{N}$. Then, using Exercise 5.9, x is a fixed-point

[5] e.g., Combettes & Glaudin, "Proximal activation of smooth functions in splitting algorithms for convex image recovery", *SIAM J. Imaging Sci., 2019*, or Briceño-Arias et al., "A random block-coordinate Douglas-Rachford splitting method with low computational complexity for binary logistic regression", *Comput. Optim. Appl.*, 2019.

[6] (or, if you must, use this link for the solution: tiny.cc/b7d7vz)

of the Douglas Rachford algorithm if and only if

$$\begin{aligned} x = x + \frac{\lambda_n}{2}\left(R_1(R_2x) - x\right) &\Leftrightarrow 0 = \frac{\lambda_n}{2}\left(R_1(R_2x) - x\right) && (89)\\ &\Leftrightarrow x = R_1(R_2x) && (90)\\ &\Leftrightarrow x = R_1(2\text{Prox}_{\gamma g}x - x) && (91)\\ &\Leftrightarrow x = 2\text{Prox}_{\gamma f}(2\text{Prox}_{\gamma g}x - x) + x - 2\text{Prox}_{\gamma g}x && (92)\\ &\Leftrightarrow \text{Prox}_{\gamma g}x = \text{Prox}_{\gamma f}(2\text{Prox}_{\gamma g}x - x) && (93)\\ &\Leftrightarrow \begin{cases} y = \text{Prox}_{\gamma g}x \\ y = \text{Prox}_{\gamma f}(2y - x). \end{cases} && (94) \end{aligned}$$

Therefore, in view of Exercise 4.6

$$\begin{aligned} x = x + \frac{\lambda_n}{2}\left(R_1(R_2x) - x\right) &\Leftrightarrow \begin{cases} x - y \in \partial\gamma g(y) \\ (2y - x) - y = \in \partial\gamma f(y). \end{cases} && (95)\\ &\Leftrightarrow \begin{cases} \dfrac{x - y}{\gamma} \in \partial g(y) \\ \dfrac{y - x}{\gamma} \in \partial f(y). \end{cases} && (96) \end{aligned}$$

Adding implies that $0 \in \partial f(y) + \partial g(y)$. One statement I did not mention is that $\partial(f) + \partial(g) \subset \partial(f+g)$ basically always holds (while achieving equality for the reverse inclusion requires extra hypotheses, as discussed in the Sum Rule Theorem 3.14). Hence, by Fermat's rule (Theorem 3.13), $y = \text{Prox}_{\gamma g}x$ is a minimizer of $f + g$!

5.1 What about a sum of functions?

Okay great, thanks to the Douglas-Rachford algorithm (Theorem 5.8), we can minimize a sum of two functions using their proximity operators. But this still does not give us a route to solve Example 5.1. What about an arbitrary number of functions? For $f_1, \ldots, f_m \in \Gamma_0(\mathcal{H})$, I want to minimize

$$\sum_{i=1}^{m} f_i(x). \tag{97}$$

We will start with the following construction.

Exercise 5.10 Let $\mathcal{H}$ be a Hilbert space, and consider the m-fold direct sum $\boldsymbol{\mathcal{H}} = \oplus_{i=1}^{m}\mathcal{H}$. Let us define the **diagonal subspace**:

$$D = \left\{(x_i)_{i=1}^{m} \in \boldsymbol{\mathcal{H}} \mid (\forall i, j \in \{1, \ldots, m\}) \quad x_i = x_j\right\}. \tag{98}$$

Show that the projection onto D is obtained by averaging all of the components:

$$(\forall (x_i)_{i=1}^{m} \in \boldsymbol{\mathcal{H}}) \quad \text{Proj}_D((x_i)_{i=1}^{m}) = \left(\frac{1}{m}\sum_{i=1}^{m} x_i\right)_{i=1}^{m} \tag{99}$$

Example 5.11 (Product space technique) Let $(f_i)_{i=1}^m$ be functions on $\Gamma_0(\mathcal{H})$, and suppose I want to

$$\underset{x\in\mathcal{H}}{\text{minimize}} \ \sum_{i=1}^{n} f_i(x). \tag{100}$$

We are going to solve (100) by reformulating it on the product space $\boldsymbol{\mathcal{H}} = \bigoplus_{i=1}^m \mathcal{H}$. Set

$$f\colon \boldsymbol{\mathcal{H}} \to \left]-\infty,+\infty\right] : (x_i)_{i=1}^m \mapsto \sum_{i=1}^m f_i(x_i) \quad \text{and} \tag{101}$$

$$g = \iota_D, \quad \text{where } D \text{ is defined in (98).} \tag{102}$$

Then

$$\underset{x\in\boldsymbol{\mathcal{H}}}{\text{minimize}} \ f(x)+g(x) = \underset{\substack{x_1\in\mathcal{H}\\ \vdots \\ x_m\in\mathcal{H}\\ (\forall i,j)\ x_i=x_j}}{\text{minimize}} \ \sum_{i=1}^m f_i(x_i) = \underset{x\in\mathcal{H}}{\text{minimize}} \ \sum_{i=1}^m f_i(x). \tag{103}$$

So, using this product space $\boldsymbol{\mathcal{H}}$, we were able to write our original problem (100) in the form (86) which is tractable with the Douglas Rachford algorithm (Theorem 5.8). Now, we only need the proximity operators of f and g. However, from Proposition 4.8 and Exercise 5.10 we know that

$$\text{Prox}_{f}(x_i)_{i=1}^m = (\text{Prox}_{f_1}(x_1), \ldots, \text{Prox}_{f_m}(x_m)) \tag{104}$$

$$\text{Prox}_{g}(x_i)_{i=1}^m = \text{Proj}_{D}((x_i)_{i=1}^m) = \left(\frac{1}{m}\sum_{i=1}^m x_i\right)_{i=1}^m, \tag{105}$$

so we can use DR to minimize (100). This yields the following algorithm.

Algorithm 5.12 (Douglas-Rachford in a product space) Let $(f_i)_{i=1}^m$ be functions in $\Gamma_0(\mathcal{H})$ such that a minimizer of (97) exists, let $(\lambda_n)_{n\in\mathbb{N}}$ be a sequence in $[0,2]$ such that $\sum_{n\in\mathbb{N}} \lambda_n(2-\lambda_n) = +\infty$, and let $\gamma > 0$. Let $\{x_{0,i}\}_{i=1}^m \subset \mathcal{H}$ and set

```
for $n = 0, 1, \ldots$
    $y_n = \frac{1}{m}\sum_{i=1}^m x_i$                       # Prox$_{\gamma g}$ step
    for $i = 1, \ldots, m$
        $z_{n,i} = \text{Prox}_{\gamma f_i}(2y_n - x_{n,i})$     # Prox$_{\gamma f}$ step
    $x_{n+1} = x_n + \lambda_n(z_n - y_n)$.
```
(106)

One drawback is that storage scales linearly with the number of summands in our objective. Unfortunately, I am not aware of many prox-based methods which can avoid this issue.

Exercise 5.13 Show that the storage required by the product-space Douglas-Rachford algorithm (106) scales like $\mathcal{O}(m)$.

5.2 Practitioners' notes and algorithmic advances

Remark 5.14 (Computing a prox) If you find yourself needing to compute the prox of a function, you can oftentimes avoid computing it directly. The easier route is to use existing theory to cobble together the prox of your specific function. The prox operators of many "base" nonlinear functions $f \in \Gamma_0(\mathcal{H})$ appearing in optimization are available in the libraries discussed in Remark 4.11. From there, you can use some of the following rules to compute the prox of your "fancier" function g in terms the prox of your "base" function f. These results (and many, many more) appear in Chapter 24 of the class book. Let $z, u \in \mathcal{H}$.

(i) Let $g = f(\cdot - z)$. Then $\text{Prox}_{\gamma g}(x) = z + \text{Prox}_{\gamma f}(x - z)$.

(ii) Let $g = f + \frac{\alpha}{2}\|\cdot - z\|^2 + \langle \cdot \mid u\rangle$. Then $\text{Prox}_{\gamma g} x = \text{Prox}_{\gamma(\gamma\alpha+1)^{-1} f}((\gamma\alpha + 1)^{-1}(x + \gamma(\alpha z - u)))$.

(iii) Let $g(x) = f(-x)$. Then $\text{Prox}_g(x) = -\text{Prox}_f(-x)$.

(iv) Let $\mathcal{H} = \mathbb{R}^{n\times m}$ be the real Hilbert space of $n \times m$ matrices under the Frobenius norm. Let $r = \min\{m, n\}$ and, for $x \in \mathcal{H}$, we denote its reduced SVD with $x = U \operatorname{diag}(\sigma_1(x), \ldots, \sigma_r(x))V^\top$. Let $f \in \Gamma_0(\mathbb{R})$ be an even function[7]. If, for every $x \in \mathcal{H}$, $g(x) = \sum_{i=1}^r f(\sigma_i(x))_{i=1}^r$, then

$$\text{Prox}_{\gamma g} = U \operatorname{diag}(\text{Prox}_f \sigma_1(x), \ldots, \sigma_k(x), 0, \ldots, 0)V^\top, \tag{107}$$

where $k = \text{rank}(x)$.[8]

(v) Let $g = f \circ L$ where L is an invertible linear operator such that $L^{-1} = L^*$ (e.g., Fourier transform, DCT, or an orthogonal wavelet transform). Then $\text{Prox}_{\gamma g} x = L^*(\text{Prox}_{\gamma f}(Lx))$.

The book has extensive results on how to handle more arcane linear operators than in (v). In the worst case, one may need to "split" a function f from its linear operator $L\colon \mathcal{H} \to \mathcal{G}$, using the graph of the linear operator via the following technique. By setting $G = \{(x, y) \in \mathcal{H} \times \mathcal{G} \mid Lx = y\}$, we can reformulate a problem on $\mathcal{H}$ with a generic linear operator as a problem on the product space $\mathcal{H} \bigoplus \mathcal{G}$ as follows

$$\inf_{x\in\mathcal{H}} f(Lx) = \inf_{\substack{(x,y)\in\mathcal{H}\bigoplus\mathcal{G} \\ Lx=y}} f(y) = \inf_{(x,y)\in\mathcal{H}\bigoplus\mathcal{G}} f(y) + \iota_G(x, y). \tag{108}$$

With the reformulation (108) (where, in the final infimum, f is technically viewed as a function which maps (x, y) to $f(y)$), one can use any basic splitting algorithm. Formulae for computing $\text{Prox}_{\iota_G} = \text{Proj}_G$ are presented in Example 29.19 in the book (note they all require inverting a linear operator).

[7] An **even** function satisfies $f(-x) = f(x)$

[8] This result is used to derive the prox of the nuclear norm $\|\cdot\|_{\text{nuc}} = \sum_{i=1}^r |\sigma_i(\cdot)|$ which is used in low-rank recovery problems in data science and image processing.

In the last several decades, there have been a **lot** of algorithmic "bells and whistles" folks have added to prox-based algorithms for solving problems of the form (97) (and more generic models as well). I will hand-wavily discuss a few of them with terminology one could use to search around in the literature.

(i) **Parallelism**: For most prox-based algorithms, the lion's share of computational time is usually devoted to computing the prox operators (since the other operations are often simple linear algebra operations). Structures like the product-space DR algorithm in (106) allow the prox operators in the inner loop to be activated in parallel.

(ii) **Block-activation** While parallelized algorithms are helpful, processing every $\operatorname{Prox}_{f_i}$ for $i \in \{1, \dots, m\}$ may be intractable or prohibitively slow. For some data science applications (e.g., in Example 5.1), this corresponds to processing a full epoch over a dataset in every iteration. This issue has given rise to the advent of **block-iterative** (or **block-activated**) algorithms where, instead of activating every operator $(\operatorname{Prox}_{f_i})_{i=1}^m$ at every iteration $n \in \mathbb{N}$, we only activate a subset $I_n \subset \{1, \dots, m\}$ of them. By re-using old iterates for the "non-activated" terms in $\{1, \dots, m\} \setminus I_n$, we save a lot of computational effort. There are both *deterministic* and *stochastic* variants of these algorithms. For deterministic algorithms, one can select I_n such that we expect the computations for $(\operatorname{Prox}_{f_i})_{i \in I_n}$ to finish at roughly the same time.

(iii) **Asynchrony** In most block-activated methods, we must wait until all of the calculations of $(\operatorname{Prox}_{f_i})_{i \in I_n}$ are completed before one can perform a *synchronization* step to complete one iteration. However, asynchronous algorithms circumvent this waiting issue.

(iv) **Extrapolation** Many iterative schemes use updates of the form

$$x_{n+1} = x_n + \lambda_n d_n,$$

where d_n is the "direction" we travel from the current iterate, and λ_n can be viewed as a step-size. For most of our algorithms, λ_n has a fixed upper bound. However (particularly in early parts of an optimization algorithm), large steps can yield favorable convergence. So, there are algorithms out there which allow one to use larger values of λ_n which are computed on-the-fly during your iteration (rather than pre-scheduled stepsizes, as in the vanilla DR/FB algorithms).

(v) **Acceleration** is a method for improving the convergence rate of an algorithm.

These terms are often turned into adjectives in the literature, for instance many **Projective Splitting** algorithms[9] are *parallel, block-activated, and asynchronous*.

[9]e.g., "Asynchronous block-iterative primal-dual decomposition methods for monotone inclusions" by Combettes & Eckstein, in *Math. Program.*, 2018, or "Projective splitting with forward steps" by Jonstone and Eckstein in *Math. Program.*, 2022.