SUPPORT VECTOR MACHINES (SVM) NATIVELY IMPLEMENTED IN THEANO, ENTIRELY ON THE GPU WITH THE CUDA BACKEND, WITH CONSTRAINED GRADIENT DESCENT

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Contents

Date: 29 mars 2017.

Key words and phrases. Machine Learning, Support Vector Machines, Constrained Optimization, Projected Gradient Descent, CUDA C/C++, theano.

Abstract. 1. Executive Summary

I implemented SVM natively in theano, and can run entirely on the GPU(s) (through the CUDA C/C++ backend). Solving the *constrained optimization* problem to train a SVM here used *parallel reduce* algorithms; in fact a parallel reduce nested in side a parallel reduce. The work complexity achieved in this case should be of $O(2 \log m)$ where m is the total number of training examples, as opposed to $O(m^2)$ for Quadratic Programming (QP), such as Sequential Minimal Optimization (SMO). Training on the same data set for vehicles as previous work for C/C++ library `libsvm` that uses SMO, `SVM_parallel` (what I call the described implementation here) achieves an accuracy of 95.1% on test data, as opposed to 87.8% for `libsvm`. What I'd like to do in the future is to train and test on larger datasets ($m > 10000$) to test `SVM_parallel` for its promising scalability, and to implement it as the outer layer of a deep neural network (DNN) for "Deep SVM".

2. Motivations and Introductions

Support Vector Machines (SVM) can be used for (binary) classification in supervised learning on labeled data, being able to learn non-linear, higher-dimensional features and to predict a boundary line or discriminator between classes, through the so-called *kernel trick*, which is really to presume a higher-dimensional Hilbert space to represent feature space $\mathcal{F}$ (i.e. $\mathcal{F}$ is a Hilbert space).

I considered the proposition of having, as a final, outer "layer," of a deep neural network (DNN) to be a support vector machine. Could it outperform the same DNN with a sigmoid or softmax function at the final layer?

To my knowledge, there was not a native implementation of SVM on theano, a Python framework for deep learning/DNNs. Being that I sought to speed up learning/computation on the GPU(s) through the theano CUDA backend, it would seem to defeat the GPU speedup advantages if from the very last layer, a large global memory transfer had to occur from the GPU (with the last DNN layer), to a host CPU, *serial*, implementation of SVM. Global memory transfers are prohibitive expensive, for latency, i.e. time-wise, between host CPU and GPUs.[1]

The outline/plan/highlights for this (short) paper is as follows: in

- 3, I review or summarize points in the theory for SVM, give derivations, etc., in which (this has all been done before; I sought to give a concise review)
  - 3.1, I recap basic, elementary concepts motivating the linear discriminator concept and what hyperplanes are, and how they're defined by a *linear* function
  - 3.2 - I continue to review and present derivations for the *Lagrangian*, a Lagrange multiplier problem, we want to minimize, and apply the usual Karush-Kuhn-Tucker (KKT) condition to make progress in deriving the constrained optimization problem we seek to solve,
  - 3.3 kernel trick, with the feature space $\mathcal{F}$ as a Hilbert space, 3.4 slack variables to deal with non-perfectly-separable data, and more derivation that was done before, but made explicitly here
- 4, the *constrained optimization* problem we wish to solve for SVM

- 5, I translate how our constrained optimization problem is to be solved with *projected gradient descent* or "constrained gradient descent" (as the projection operators enforce constraint equalities and inequalities). As noted, this method/algorithm was chosen to utilize the (very useful) `grad` method in the `theano` software package.

  The Eqns. 25, 26 at the end of 5.1 is the crux of the training method considered in this paper and code for SVM and was directly referred to when implemented in code.
  - I also show how to compute, in a numerically stable manner, the intercept b, after λ_i Lagrange multipliers are found, and how to compute predictions $\widehat{y}$, in 5.2, 5.2.1
- 6 the *constrained gradient descent* to solve our constrained optimization problem is implemented and I detail its implementation using software package `theano`, and especially its CUDA backend, so to run solely on the *GPU*. I give the rationale in deploying this constrained gradient descent as opposed to the Sequential Minimal Optimization (SMO) usually used in the predominant SVM software package (in C/C++) `libsvm`. Noteworthy, I also show that **work complexity goes from** $O(m^2)$ **to** $O(2\log(m))$.
- 6.1 briefly tells where the code is made available and the 1-to-1 correspondence between the code and mathematical formulation. I also note that *novel use of theano's reduce within reduce.*
- 7 has *results* that I try to compare with sample datasets used previously, that I've trained as quickly as possible. Look here for the results; better yet, feel free to try the code and jupyter notebook and share benchmarks.[1]

3. Concise Mathematical Review/Summary of the theory for SVM

3.1. Hyperplanes and distances to motivate the linear discriminator concept; Support Vector Machine name.

I'll recap basic, elementary concepts, from Clarke, Fokoue, and Zhang (2009) [5], that motivate the concept of a linear discriminator classifying input data X.

Consider $\theta \in \mathbb{R}^d$, and a linear function y,

$$\begin{aligned} &y : \mathbb{R}^d \to \mathbb{R} \\ &y(x) := \langle \theta, x \rangle + b \end{aligned} \tag{1}$$

Consider a "level set" at real number value $c \in \mathbb{R}$, $H_c(\theta, b)$:

$$H_c(\theta, b) := \{x | y(x) = \langle \theta, x \rangle + b = c\} \tag{2}$$

where $\dim H_c(\theta, b) = d - 1$ is a *hyperplane*.

$\theta \in \mathbb{R}^d$ is the normal vector to this hyperplane, since,

$$\forall\, x^{(i)}, x^{(j)} \in H_c(\theta, b), \text{ then}$$
$$\langle \theta, x^{(i)} \rangle + b = c = \langle \theta, x^{(j)} \rangle + b \Longrightarrow \langle \theta, x^{(i)} - x^{(j)} \rangle = 0$$

and since $x^{(i)} - x^{(j)} \in TH_c(\theta, b)$, i.e. $x^{(i)} - x^{(j)}$ belongs in the tangent space to $H_c(\theta, b)$, $TH_c(\theta, b)$, then θ, in general, is normal to the hyperplane ($\langle \theta, x^{(i)} - x^{(j)} \rangle = 0$).

[1] github:ernestyalumni/MLgrabbag/ML, github:ernestyalumni/MLgrabbag SVM theano.ipynb

Given $z \in \mathbb{R}^d$, what is the distance from z to this hyperplane $H_c(\theta, b)$, $d(z, H_c(\theta, b))$? Consider $z* = z + t\theta \in H_c(\theta, b)$. Then

$$\langle \theta, z^* \rangle + b = c = \langle \theta, z \rangle + t\langle \theta, \theta \rangle + b = c \Longrightarrow t = \frac{c - b - \langle \theta, z \rangle}{\|\theta\|^2}$$

$$\text{and so } d(z, H_c(\theta, b)) = \|t\theta\| = \frac{|\langle \theta, z \rangle + b - c|}{\|\theta\|}$$

Thus, for the perpendicular distance between 2 parallel hyperplanes, $H_c(\theta, b)$, $H_{c'}(\theta, b)$, can be found: choose a pt. from $H_c(\theta, b)$, without loss of generality, s.t. $z = \left(\frac{c-b}{\theta_1}, 0, \dots 0 \right)$, so that

$$\langle \theta, z \rangle + b = c \Longrightarrow \theta_1 z^1 = c - b$$

Then

$$d(H_c(\theta, b), H_{c'}(\theta, b)) = \frac{|\langle \theta, z \rangle + b - c'|}{\|\theta\|} = \frac{|c - b + b - c'|}{\|\theta\|} = \frac{|c - c'|}{\|\theta\|} \tag{3}$$

Given an input (data) domain $\mathcal{X} \subseteq \mathbb{R}^d$, for the case of binary classification, with total number of classes $K = 2$, we can consider representing the outcomes y for each input data example, $X \in \mathbb{R}^d$, in 2 ways:

$$y \in \{-1, 1\} \textbf{ or } \; y \in \{0, 1\} \text{ for } y \in \{0, 1, \dots K - 1\} \, (K = 2) \tag{4}$$

What ends up happening is that the distance between 2 hyperplanes, $c = -1, c' = 1$ vs. $c = 0, c' = 1$, respectively, changes, as $d(H_c(\theta, b), H_{c'}(\theta, b)) = \frac{|c-c'|}{\|\theta\|}$, but its absolute value doesn't matter. What matters is the form of $y : \mathbb{R}^d \to \mathbb{R}$, of Eq. 1 which defines the hyperplane in Eq. 2, notably in θ, b. The lesson is to *be consistent with what the value of y is to define what class X belongs to.* For instance, Bishop (2007) [4] and Clarke, Fokoue, and Zhang (2009) [5] chooses to consider $y \in \{-1, 1\}$, $\forall X$ and I'll do the same here.

The name "support vectors" seems to come from this intuitive notion: $\theta \in \mathbb{R}^d$,b are determined from m input data examples $X^{(i)} \in \mathbb{R}^d$, $\forall i = 1, 2, \dots d$, and $\forall i$, the corresponding class label $y \in \mathbb{Z}$. $\forall X^{(i)} \in \mathbb{R}^d$, imagine attaching normal vectors of the form $t\theta$, $t \in \mathbb{R}$ that extend out to the respective hyperplane, determined by $y^{(i)}$. These imagined vectors "support" the respective hyperplane.

An important takeaway is that the equation defining the hyperplane $H_c(\theta, b)$ in Eq. 2 is *linear.*

3.2. Margins, cost functional or "Lagrangian", dual formulation.

With output, outcome $y \in \{-1, 1\}$, $\forall X$ input data example, the distance between the 2 hyperplanes, which are level sets of $c = -1, c' = 1$, is

$$\frac{2}{\|\theta\|}$$

The method of SVM seeks to maximize this distance, also known as "margin", to make margins as big as possible.

Clearly, this is equivalent to minimizing $\frac{1}{2}\|\theta\|^2$, with $\frac{1}{2}$ multiplication factor chosen, without loss of generality, to make taking derivatives of θ easier.

But we also have the following constraints. We want to have a "margin" of $\frac{2}{\|\theta\|}$ between the hyperplanes that'll separate the input data examples $X^{(i)}$, $\forall i = 1, 2, \dots m$, for different classes, in this binary classification class, of those with

$y^{(i)} \in \{-1, 1\}$, and so those $X^{(i)}$'s will "fall far away" from this "margin" and remain within its corresponding hyperplane $H_{c'=1}(\theta, b)$ or $H_{c=1}(\theta, b)$, thus defining these inequalities:

$$y^{(i)}(\langle \theta, x^{(i)} \rangle + b) - 1 \geq 0 \qquad \forall\, i = 1, \dots, m \tag{5}$$

So we want to find

$$\theta \in \mathbb{R}^d \backslash \{0\},\ b \in \mathbb{R}$$

s.t.

$$y^{(i)}(\langle \theta, x^{(i)} \rangle + b) - 1 \geq 0 \qquad \forall\, i = 1, \dots, m$$

where $\frac{2}{\|\theta\|}$ is maximized, or equivalently, defining the so-called *objective function* $f_\theta(\theta, b)$, minimize $f_\theta(\theta, b)$:

$$f_\theta(\theta, b) := \frac{1}{2}\|\theta\|^2 \qquad \text{(objective function)} \tag{6}$$

Consider then this cost functional, also known as the "Lagrangian", which we want to *minimize.*

(7)

$$\mathcal{L}((\theta, b), \lambda) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle \theta, x^{(i)} \rangle + b - 1) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle \theta, x^{(i)} \rangle + b) + \sum_{i=1}^m \lambda_i$$

Of note, we introduced *Lagrangian multipliers* λ_i, $\forall i \in 1, 2 \dots m$, $\lambda_i \in \mathbb{R}$, to account for each of the constraints given in Eq. 5.

The Karush-Kuhn-Tucker (KKT) condition tells us that (θ, b) makes $\mathcal{L}$ a minimum for a certain λ (and that these λ_i's exist), and that these relations hold:[2], [3]:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta_j} &= 0 = \theta_j - \sum_{i=1}^m \lambda_i y^{(i)} x_j^{(i)} \qquad j = 1, 2 \dots d \\ \frac{\partial \mathcal{L}}{\partial b} &= 0 = -\sum_{i=1}^m \lambda_i y^{(i)} \end{aligned} \tag{8}$$

and

$$\lambda_i \geq 0 \qquad \forall\, i = 1, 2, \dots m \tag{9}$$

,

$$\sum_{j=1}^m \lambda_i y^{(i)}(\langle \theta, x^{(i)} \rangle) + b - 1) = 0 \tag{10}$$

$\forall\, i = 1, 2, \dots m$, we want input data example $X^{(i)}$ to be "far away" from the boundary line, or, i.e. to give enough "margin" from the other class's hyperplane, and so in general, $(\langle \theta, x^{(i)} \rangle) - b - 1)$ will be non-zero in Eq. 10. So this condition is equivalently

$$\sum_{j=1}^m \lambda_i y^{(i)} = 0 \tag{11}$$

It's interesting to see that the step in taking the partial derivatives of $\mathcal{L}$ in Eq. 8 is analogous to the construction/computation of dual "conjugate" variables, conjugate momentum, in physics.

Notice then that

$$\begin{aligned} \frac{1}{2}\|\theta\|^2 = \frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)}y^{(j)}\langle x^{(i)}, x^{(j)}\rangle \text{ and} \\ \sum_{i=1}^m \lambda_i y^{(i)}(\langle\theta, x^{(i)}\rangle + b) = \sum_{i=1}^m \lambda_i y^{(i)}(\sum_{j=1}^m \lambda_j y^{(j)}\langle x^{(j)}, x^{(i)}\rangle) \end{aligned} \tag{12}$$

$$\implies \mathcal{L}((\theta, b), \lambda) = -\frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)}y^{(j)}\langle x^{(i)}, x^{(j)}\rangle + \sum_{i=1}^m \lambda_i \tag{13}$$

3.3. **So-called "Kernel trick"; feature space is a Hilbert space.** The so-called "feature space" $\mathcal{F}$ is a Hilbert space $\mathcal{H}$, $\Phi : \mathbb{R}^d \to \mathcal{H}$, equipped with inner product

$$\langle\Phi(x), \Phi(y)\rangle = K(x, y) \tag{14}$$

with $K : \mathbb{K}^d \times \mathbb{K}^d \to \mathbb{K}^K$ being called the kernel function. Recall that the feature space $\mathcal{F}$ had been introduced to represent the process of preprocessing input data X. For example, given a single input data example, $X = (X_1, \dots, X_d) \in \mathbb{R}^d$, maybe we'd want to consider polynomial features, linear combinations of various orders of monomials X_iX_j or $X_i^2X_j$, and so on. Then Φ represents the map from X to all these features.

As both a pedantic remark and academic question, I had denoted $\mathbb{K}$ to be, in general, a *field* - (very familiar) examples of fields are $\mathbb{K} = \mathbb{R}, \mathbb{C}, \mathbb{Z}$, the real numbers, complex numbers, integers, respectively. Many times, input data that we receive could take on discrete values, meaning $X^{(i)} \in \mathbb{Z}$. Would there be issues, in proving existence, continuity, and differentiability, throughout derivations for these SVM algorithms, if the underlying field $\mathbb{K}$ is not the real number line $\mathbb{R}$?

Nevertheless, the essense of the kernel trick is this: the explicit form of Φ need *not be known*, nor even the space $\mathcal{H}$. Only the kernal function K form needs to be guessed at.

And so even if we now have to modify our Eq. 7 to account for this preprocessing map Φ, applied first to our training data $X^{(i)}$, we essentially still have the same form, formally.

Keep in mind the whole point of this nonlinear preprocessing map Φ - we want to keep the linear discrimination procedure with the weight, or parameter θ, and intercept b, being this linear model on the feature space (Hilbert space) F. We're *linear* in $\mathcal{F}$. But we're *nonlinear* in the input data $X = \{X^{(1)}, \dots X^{(m)}\}$'s domain.

So,

$$\mathcal{L}((\theta,b),\lambda) = \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle\theta,\Phi(x^{(i)})\rangle + b - 1) =$$

$$= \frac{1}{2}\|\theta\|^2 - \sum_{j=1}^m \lambda_i y^{(i)}(\langle\theta, x^{(i)}\rangle + b) + \sum_{i=1}^m \lambda_i \text{ and so}$$

$$\frac{\partial\mathcal{L}}{\partial\theta_j} = 0 = \theta_j - \sum_{i=1}^m \lambda_i y^{(i)}\Phi(x)_j^{(i)} \tag{15}$$

$$\Longrightarrow$$

$$\mathcal{L}((\theta,b),\lambda) = -\frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)}y^{(j)}\langle\Phi(x^{(i)}),\Phi(x^{(j)})\rangle + \sum_{i=1}^m \lambda_i = \mathcal{L}(X,y,\lambda)$$

Note that we'll now want to *maximize* this dual formulation $\mathcal{L}(X,y,\lambda)$.

3.4. **Dealing with Errors, (non-negative) slack variables, dealing with not-necessarily perfectly separable data.** First, "loosen the strict constraint" $y^{(i)}(\langle\theta,x^{(i)}\rangle + b) \geq 1$ by introducing *non-negative* slack variables ξ_i, $i = 1\dots m$,

$$y^{(i)}(\langle\theta,x^{(i)}\rangle - b) \geq 1 - \xi_i, \qquad \forall\, i = 1,2,\dots m \tag{16}$$

Simply add ξ to the objective function to implement penalty (for "too much slack"), with a "regularization" constant C (in analogy to regularization in the linear regression or logistic regression classifier methods):

$$f_0(\theta,b,\xi) = \frac{1}{2}\|\theta\|^2 + C\sum_{i=1}^m \xi_i \tag{17}$$

So then the total Lagrangian becomes

$$\mathcal{L}(\theta,b,\xi,\lambda,\mu) = \frac{1}{2}\|\theta\|^2 + C\sum_{i=1}^m \xi_i - \sum_{i=1}^m \lambda_i(y^{(i)}(\langle\theta,x^{(i)}\rangle + b) - 1 + \xi_i) - \sum_{i=1}^m \mu_i\xi_i \tag{18}$$

where the constraint is turned into a Lagrange-multiplier type relation:

$$\xi_i \geq 0 \Longrightarrow \mu_i(\xi_i - 0) \qquad \forall\, i = 1,\dots m \tag{19}$$

$-\mu_i\xi_i$ is indeed a valid cost (penalty) functional (if $\xi_i < 0$, $-\mu_i\xi_i > 0$, and there's more penalty as ξ_i gets more negative. I understood this cost or penalty accounting, given an *inequality constraint*, from reading notes from here, http://www.pitt.edu/~jrclass/opt/notes4.pdf).

If we "turn the crank" and take partial derivatives of $\mathcal{L}$, with respect to ξ_i, finding its "conjugate momentum dual", we'll actually see that $\mathcal{L}$ has *no* dependence on

ξ_i:

$$\frac{\partial \mathcal{L}}{\partial \xi_i} = C - \lambda_i - \mu_i = 0$$

$$\begin{aligned} \text{since} \quad & C - \lambda_i = \mu_i \qquad \Longrightarrow C \geq \lambda_i \\ & \mu_i \geq 0 \text{ is given} \\ & \qquad \Longrightarrow \end{aligned} \tag{20}$$

$$\mathcal{L}(\theta, b, \xi, \lambda, \mu) = \frac{1}{2}\|\theta\|^2 - \sum_{i=1}^m \lambda_i (y^{(i)}(\langle \theta, \Phi(x^{(i)})\rangle) + \sum_{i=1}^m \lambda_i = \mathcal{L}((\theta, b), \lambda)$$

ξ, μ no longer appear in the dual Lagrangian, $\mathcal{L}(X, y, \lambda)$, which we want to *maximize*, nor in the so-called "primal" Lagrangian, $\mathcal{L}((\theta, b), \lambda)$.

4. Dual Formulation

Denoting $W(\lambda) := -\mathcal{L}(X, y, \lambda)$,

$$\boxed{\begin{aligned} \text{minimize} \qquad & W(\lambda) = -\sum_{i=1}^m \lambda_i + \frac{1}{2}\sum_{i,j=1}^m \lambda_i \lambda_j y^{(i)} y^{(j)} K(x^{(i)}, x^{(j)}) \\ \text{s.t.} \quad & \sum_{i=1}^m \lambda_i y^{(i)} = 0 \\ & 0 \leq \lambda_i \leq C \qquad \forall\, i = 1, 2 \dots m \end{aligned}} \tag{21}$$

At this point, Eq. 21 is what I could consider the "theoretical gold" version. Further modification of this formulation are really to efficiently implement this on the computer (or microprocessor!). But the schemes should respect this "gold" version and compute what this is and say.

5. Constrained Optimization

Wotao Yin's notes had a terse, but to-the-point, survey/summary of optimization, in particular nonlinear optimization with inequality constraints, for his courses 273a and Math 164, Algorithms for constrained optimization. In both course notes, the material is "taken from the textbook Chong-Zak, 4th. Ed." So we'll refer to Chong and Zak (2013) [3].

From Ch. 22 "Algorithms for Constrained Optimization", 2nd. Ed., pp. 439, Sec. 22.2 "Projections", consider $\Omega \subset \mathbb{R}^d$, with

$$\Omega = \{\mathbf{x} | l_i \leq x_i \leq u_i,\, i = 1 \dots d\}$$

Let us denote $\Pi \equiv$ projection operator. Let us mathematically formulate how projection operator Π maps a point $\mathbf{x} \in \mathbb{R}^d$ onto the subset $\Omega \subset \mathbb{R}^d$ defined above:

$$\forall\, \mathbf{x} \in \mathbb{R}^d,\; y := \Pi[x] \in \mathbb{R}^d$$

$$y_i \equiv \begin{cases} u_i & \text{if } x_i > u_i \\ x_i & \text{if } l_i \leq x_i \leq u_i \\ l_i & \text{if } x_i < l_i \end{cases} \tag{22}$$

5.1. **Projected Gradient descent.** We want to minimize $W(\lambda)$ in Eq. 21. The software package `theano` provides the graph-generating method `grad`, which automatically computes the symbolic gradient of a scalar-valued function of symbolic (theano) variables. This `grad` has been very useful for automating the computation of the so-called "back-propagation" step of machine learning/deep learning.

We would like to reuse this useful theano method for SVM. Therefore I sought out a solution to our constrained optimization problem that'll involve computing gradients at each iteration, but subject to our constraint equality and inequalities.

We already know how to deal with constraint *inequalities* via the projection operator in Eq. 22. And note that this can be simply implemented in Python/theano with a `if/else` statement(s) and `theano.tensor.switch`, respectively.

To implement the constraint *equality*, $\sum_{i=0}^m \lambda_i y^{(i)} = 0$, consider the orthogonal projector matrix (operator)

$$\mathbf{P} := \mathbf{1}_{\mathbb{R}^d} - A^T(AA^T)^{-1}A \tag{23}$$

with A being a transformation from $\mathbb{R}^d$ to $\mathbb{R}^m$, i.e. $A : \mathbb{R}^d \to \mathbb{R}^m$, and where $Ax = b$ is the constraint equality (written in its most general form) [3].

So for where $\Omega = \{X | AX = b\}$, if $m = 1$, then

$$\text{Proj}_\Omega(\mathbf{y}) = \mathbf{y} - \frac{\mathbf{a}_1^T\mathbf{y} - b}{\|\mathbf{a}_1\|^2}\mathbf{a}_1$$

If $m > 1$, then

$$\text{Proj}_\Omega(\mathbf{y}) = (1_{\mathbb{R}^d} - A^T(AA^T)^{-1}A)\mathbf{y} + A^T(AA^T)^{-1}\mathbf{b}$$

For the linear (equality) constraint

$$\sum_{i=1}^m \lambda_i y^{(i)} = 0$$

we have

$$\mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)}=0}(\mathbf{y}) = \left(\mathbf{y} - \frac{\sum_{i=1}^m y^{(i)}(\mathbf{y})_i}{\sum_{i=1}^m (y^{(i)})^2}(y^{(i)})\mathbf{e}_i \right) \tag{24}$$

In summary,

$$\boxed{\begin{gathered} \text{we seek to minimize} \\ W(\lambda) = -\sum_{i=1}^m \lambda_i + \frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)}y^{(j)}K(X^{(i)}, X^{(j)}) \\ \text{by iterating } t = 0, 1, \dots, \text{ as such:} \\ \lambda_i'(t+1) := \lambda_i(t) - \alpha \text{grad} W(\lambda) \\ \lambda_i''(t+1) := \mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)}=0}(\lambda_i'(t+1)) \\ \lambda_i(t+1) := \Pi_{0\leq\lambda_i\leq C}(\lambda_i''(t+1)) \end{gathered}} \tag{25}$$

where

$$\boxed{\begin{aligned} &\mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)}=0}(\lambda_i'(t+1)) = \lambda_i'(t+1) - \frac{\sum_{i=1}^m y^{(i)}\lambda_i'(t+1)}{\sum_{i=1}^m (y^{(i)})^2} y^{(i)} \\ &\Pi_{0\leq \lambda_i \leq C}(\lambda_i''(t+1)) = \begin{cases} C & \text{if } \lambda_i''(t+1) > C \\ \lambda_i''(t+1) & \text{if } 0 \leq \lambda_i''(t+1) \leq C \\ 0 & \text{if } \lambda_i'(t+1) < 0 \end{cases} \end{aligned}} \tag{26}$$

The α parameter is the analogue to the *learning rate* of gradient descent and will need to be tuned.

5.2. **Computing b, the intercept, with a good algebra tip: multiply both sides by the denominator.** Bishop (2007) [4], on pp. 330 of Ch. 7, Sparse Kernel Machines, gave a very good (it resolved possible numerical instabilities) prescription on how to compute the intercept b, given λ, which would then give us the function that can make predictions $\widehat{y}$ on input data example $X^{(i)} \in X$. It's worth expounding upon here.

For any support vector (Bishop called it a support vector; what I think it's equivalent to is that we've trained on our training set $(X,y)^{\text{train}}$, and this is 1 of the training examples) $X^{(i)}$, $i = 1\dots m$,

$$y^{(i)} f(X^{(i)}) = 1 \tag{27}$$

. Then using

$$\begin{aligned} & f(x) := \sum_{i=1}^m y^{(i)} \lambda_i^* K(X^{(i)}, x) + b \\ & \Longrightarrow y^{(i)} \left(\sum_{j=1}^m y^{(j)} \lambda_j^* K(X^{(j)}, X^{(i)}) + b \right) = 1 \end{aligned} \tag{28}$$

Although we can solve this equation for b with algebra/arithmetic for our arbitrarily chosen support vector, it's numerically more stable to 1st. multiply through by $y^{(i)}$, using $(y^{(i)})^2 = 1$, and then averaging over all support vectors.

$$\begin{aligned} & \sum_{j=1}^m y^{(j)} \lambda_j^* K(X^{(j)}, X^{(i)}) + b = y^{(i)} \\ & \Longrightarrow b = \frac{1}{m} \left(\sum_{i=1}^m y^{(i)} - \sum_{i,j=1}^m y^{(j)} \lambda_j^* K(X^{(j)}, X^{(i)}) \right) \end{aligned} \tag{29}$$

5.2.1. *Prediction (with SVM).* Compute predictions with this formula: [5]

$$\begin{aligned} & \widehat{y}(X) = \sum_{i=1}^m y^{(i)} \lambda_i^* K(X^{(i)}, X) + b^* \\ & \widehat{y} : \mathbb{R}^d \to \{0, 1, \dots, K-1\} \quad \text{(with } K = 2 \text{ for binary classification)} \end{aligned} \tag{30}$$

6. Constrained Gradient Descent (Implementation)

From Eqns. 25, 26, with the algorithm or iterative, computational steps that we should take mathematically formulated (clearly), I had sought out to implement these steps using `theano` and on the GPU, in the hopes of speeding up computation and developing a method that can scale with m input data examples.

Take a look at this double summation term in Eqn. 25 for $W(\lambda)$:

$$\begin{aligned} f_1(\lambda) := \frac{1}{2}\sum_{i,j=1}^m \lambda_i\lambda_j y^{(i)}y^{(j)}K(X^{(i)},X^{(j)}) = \\ = \frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m \lambda_i y^{(i)}K(X^{(i)},X^{(j)})\lambda_j y^{(j)} = (\mathbf{q}^{(i)})^T K(X^{(i)},X^{(j)})\mathbf{q}^{(j)} \end{aligned} \tag{31}$$

Quadratic programming (denoted "QP" in computer science literature) is essentially trying to put the calculation of double sums, such as the above, into quadratic form, as in the very last equality. Techniques for efficient calculation, on the CPU, of this quadratic form, after reformulation of the original problem, make up QP.

The prevailing software package used for SVM, written in C/C++, that also underlies SVM module for sci-kit learn (`sklearn`) [7] is `libsvm` [8]. `libsvm` employs the method of Sequential Minimal Optimization (SMO) [9]. The main advantage of SMO is that only 2 λ_i's, Lagrange multipliers, are considered in the working set at each stage in time and the optimal solution is computed analytically at this point.

For instance, suppose we are considering 2 Lagrange multipliers λ_1 and λ_2. We first compute the optimal value changing λ_2 only. Then, using the inequality constraints $0 \leq \lambda_1, \lambda_2 \leq C$, and equality constraint $\sum_{i=1}^m \lambda_i y^{(i)} = 0$ (but adapted to the fact that we're only changing 2 λ_i's), we can analytically compute the other λ_1.

The (serial) computation in C/C++ of this analytical problem at this single step for SMO is fast. However, for the fitting for large data sets (large m), the fit time complexity is more than quadratic with the number of examples m, which makes it difficult to scale to datasets of more than a couple of 10000 examples ($m > 10000$) [2] [7].

Instead, I considered the idea behind *All-pairs N-body* algorithm of Nyland, Harris, and Prins (2007) in Ch. 31 of **GPU Gems 3** [6]. It was also explained in Udacity's CS344 with Owens and Luebke [1] [3].

Look again at Eq. 31, f_1, which clearly requires m^2 fetches, or reads, for $\lambda_i\lambda_j y^{(i)}y^{(j)}K(X^{(i)},X^{(j)})$ term and m^2 computations, for each (i,j) pairs (and there are m^2 total pairs). It could also help to imagine a $m \times m$ matrix:

$$\begin{matrix} i=1,j=1 & i=1,j=2 & \dots & i=1,j=m \\ i=2,j=1 & i=2,j=2 & \dots & i=2,j=m \\ \vdots & \vdots & \ddots & \dots \\ i=m,j=1 & i=m,j=2 & \dots & i=m,j=m \end{matrix}$$

and observing that $\forall\, i = 1,2,\dots m$, we're doing m computations for j and needing to fetch m values for each λ_j, y_j, $X^{(j)}$, and so on.

[2] sklearn.svm.SVC

[3] Quiz: All Pairs N-Body

Consider this computation: for a given, single $i \in \{1, 2, \dots m\}$, define

$$f_{1i}(\lambda) := \frac{1}{2} \sum_{j=1}^m \lambda_j y^{(j)} K(X^{(i)}, X^{(j)}) \tag{32}$$

For this step, we'll only need to do m fetches for the $\lambda_j, y^{(j)}, X^{(j)}$ values, and $X^{(i)}$ value will be fetched once. As this is a summation over a potentially large vector (m can be big), this looks like a good case/candidate for the usage of parallel *reduce* algorithm. The work complexity of parallel reduce is $O(\log m)$ [1][4]. Theano has an implementation of reduce in `theano.reduce`.

Once all m f_{1i}'s are obtained, for $i = 1, 2, \dots m$, then parallel reduce can be used again (especially if m is large!). Also, empirically, I found that using `theano.reduce` again helped to circumvent the problem of the maximum recursion limit for Python [5], which is inherent with Python (cf. `import sys  sys.getrecursionlimit()`). In practice, above about 10000 recursions, the Python script fails with run-time errors.

Nevertheless, in this second (parallel) reduce step, we are doing

$$f_1 = \sum_{i=1}^m \lambda_i y^{(i)} f_{1i}(\lambda)$$

with m fetches of values for $\lambda_i, y^{(i)}$. The work complexity here for this reduce step is again $O(\log(m))$

Thus, we are doing, for 2 (parallel) reduces, $2m$ fetches (or reads), for each λ_i or $y^{(i)}$ or $X^{(i)}$.

The total work complexity is $O(2\log(m))$.

Likewise, for the computation of the intercept b in Eqn. 29, after minimizing $W(\lambda)$ by varying λ, I also employed parallel reduce via `theano.reduce` (but only once for the single sum) and for the prediction step for $\widehat{y}$ in Eq. 30

6.1. **Code (theano/Python script), jupyter notebook accompanying code.**

SVM is implemented as described above, in particular Eqns. 25, 26, in the Python class `SVM_parallel`. The default kernel function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$ is the radial basis function, which takes the form of a gaussian, is first implemented and I can implement other kernel functions easily, as a Python function object and Python class member, in the future.

Take note that for the (currently) implemented radial basis function, Python function (object) `rbf` in `SVM.py` of github:ernestyalumni/MLgrabbag/ML, what's formulated is this:

$$K(X^{(i)}, X^{(j)}) = \exp\left(- \frac{\| X^{(i)} - X^{(j)} \|^2}{2\sigma^2} \right) \tag{33}$$

with $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$.

Take a look at the $\sigma \in \mathbb{R}$ parameter in Eq. 33. σ is analogous to the variance of a Gaussian (normal) distribution. For other implementations, notably `libsvm` and sci-kit learn, they use this form of the radial basis function:

$$K(X^{(i)}, X^{(j)}) = \exp\left(-\gamma \| X^{(i)} - X^{(j)} \|^2 \right)$$

[4]Step Complexity of Parallel Reduce - Intro to Parallel Programming, Udacity
[5]max recusion limit #689

So γ parameter here is equivalent to σ:

$$\gamma = \frac{1}{2\sigma^2}$$

While this redefinition makes no change to the formulation above, this is something to note when when using `libsvm`, sci-kit learn, or `SVM.py` here when manually inputting the parameters to train models.

The theano/Python code follows directly from Eqns. 25, 26 and is in the `/ML` subfolder of the github repository `MLgrabbag` [6], in `SVM.py`. Wherever a summation is seen in the mathematical formulation, `theano.reduce` is used.

In the `SVM_parallel` Python class method `build_W`, I code a `theano.reduce` inside a `theano.reduce` and show it's possible to be done. This represents, both formally and the parallel reduction on the GPU, the double summation that we sought to compute in 25 for $W(\lambda)$.

The jupyter notebook `SVM_theano.ipynb` in the same github repository steps through how I developed and used `SVM_parallel`, training it on a number of sample datasets. Because of the interactivity of jupyter notebook, I invite others to explore and play with the notebook if further clarification on `SVM_parallel`, or how to use it, is needed [7]

7. Immediate Results from training on sample datasets

7.1. **Real-World Examples.** I trained a SVM on 2 of the real-world data sets provided by Hsu, Chang, and Lin [10], one for astroparticles and another for vehicles, using, for hardware, a NVIDIA GeForce GTX 980 Ti. Checking the computational graph generated by theano (using `theano.function.maker.fgraph.toposort()`), `nvidia-smi -l 2` (monitoring real-time GPU usage), and the (usual, in Utilities) CPU resources System Monitor.

I will copy the results from Hsu, Chang, and Lin [10] for comparison. The accuracy measure is determined from the given *test* data, *not* on the training data (which is part of good machine learning and scientific practice).

Applications	# training data	# testing data	# features	# classes
Astroparticle[8]	3089	4000	4	2
Vehicle[9]	1243	41	21	2

Applications	$C =$	$\gamma =$	Accuracy by `libsvm`
Astroparticle	2.0	2.0	96.9%
Vehicle	128.0	0.125	87.8%

Table 1: Sample Dataset Problem characteristics and accuracy performance [10].

Applications	$C =$	$\sigma =$	$\alpha =$	# iterations
Astroparticle	2.0	0.30	0.001	15
Vehicle	128.0	2.0	0.001	20

Applications	Time to train (on GTX 980Ti)	Accuracy by `SVM_parallel`
Astroparticle	1h 7min 18s	96.1%
Vehicle	14min 54s	95.1%

Table 2: Results of training on Sample Datasets with `SVM_parallel`

[6] github:ernestyalumni/MLgrabbag
[7] github:ernestyalumni/MLgrabbag SVM theano.ipynb

The very last result testing on the test data for vehicles is promising for `SVM_parallel`. At this point, I would invite others to suggest sample and real-world datasets to train and test on, using `SVM_parallel`, as I also try to find other datasets, and add onto the jupyter notebook SVM theano.ipynb on github. It'd be interesting to vary the *number of training examples*, to find a dataset with more than 10000 ($m > 10000$) examples and see how `SVM_parallel` can scale with large data sets (indeed, for $m > 10000$, the SVM would have $m > 10000$ support vectors in the model), and vary the *number of features* (whether SVM does better with large or small number of features, relative to m).

8. Conclusions/Summary/Dictionary between Math and Code

I had reviewed the motivation and derivations for SVM.

What's novel is that, given the GPU(s), I implemented *constrained gradient descent* or *projected gradient descent*, for training models, instead of Quadratic Programming, that computes a quadratic form (to tackle the double summation in the dual formulation), through SMO, as used before (e.g. `libsvm`, sci-kit learn). Its (*constrained gradient descent* or its implementation here `SVM_parallel`) work complexity is $O(2 \log m)$, as opposed to $O(m^2)$. This was achieved by using theano's reduce, inside a reduce.

Its (i.e. `SVM_parallel`) promising to be scalable to large datasets ($m > 10000$). I seek to find large datasets to train and test on and are appropriate for binary classification, and invite others to make suggestions or play with the code and jupyter notebook itself.

I'll provide a 1-to-1 dictionary here between the mathematical formulation and the Python code. As a note on software engineering, object-oriented programming (OOP) and how to code classes, I had sought to identify (make isomorphisms) and design Python classes and function objects with 1-to-1 correspondence to the mathematical formulation. The hope is that it would allow other developers to rapidly make progress in improving upon the code or to rapidly understand its usage and apply it as they'd like to see fit.

we seek to minimize

$$W(\lambda) = -\sum_{i=1}^m \lambda_i + \frac{1}{2}\sum_{i,j=1}^m \lambda_i \lambda_j y^{(i)} y^{(j)} K(X^{(i)}, X^{(j)}) \qquad \texttt{SVM\_parallel.build\_W}$$

by iterating $t = 0, 1, \dots,$ as such:

`SVM_parallel.train_mode_full(max_iters=250)`

$$\begin{aligned} \lambda_i'(t+1) &:= \lambda_i(t) - \alpha \text{grad} W(\lambda) \\ \lambda_i''(t+1) &:= \mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)} = 0}(\lambda_i'(t+1)) \qquad \texttt{SVM\_parallel.build\_update} \\ \lambda_i(t+1) &:= \Pi_{0 \le \lambda_i \le C}(\lambda_i''(t+1)) \end{aligned}$$

where

$$\mathbf{P}_{\sum_{i=1}^m \lambda_i y^{(i)}=0}(\lambda_i'(t+1)) = \lambda_i'(t+1) - \frac{\sum_{i=1}^m y^{(i)}\lambda_i'(t+1)}{\sum_{i=1}^m (y^{(i)})^2} y^{(i)}$$

$$\Longleftrightarrow$$

`updatelambda_mult=updatelambda_mult-T.dot(y,updatelambda_mult)/T.dot(y,y)*y`
in `SVM.build_update`

$$\Pi_{0\leq \lambda_i \leq C}(\lambda_i''(t+1)) = \begin{cases} C & \text{if } \lambda_i''(t+1) > C \\ \lambda_i''(t+1) & \text{if } 0 \leq \lambda_i''(t+1) \leq C \\ 0 & \text{if } \lambda_i'(t+1) < 0 \end{cases}$$

$$\Longleftrightarrow$$

`updatelambda_mult=T.switch(T.lt(C,updatelambda_mult),C,updatelambda_mult)`
in `SVM.build_update`
`updatelambda_mult=T.switch(T.lt(updatelambda_mult,lower_bound),`
`lower_bound,updatelambda_mult)`
in `SVM.build_update`

Finally, to tie it back into my original motivation, now that SVM is natively implemented in theano, it would be interesting to try to develop (and of course find appropriate datasets to train and test on) a DNN that will have as its "outer" or last layer to be a SVM. Since SVM is now part of the theano computational graph, optimization (the so-called "backpropagation" step) will be done automatically and simply with theano's `grad`, on all the parameters or "weights" of the entire model.

References

[1] John Owens. David Luebki. *Intro to Parallel Programming. CS344.* **Udacity**

[2] Thomas Nowak. "Implementation and Evaluation of a Support Vector Machine on an 8-bit Microcontroller." Univ.Ass. Dipl.-Ing. Dr.techn. Wilfried Elmenreich Institut für Technische Informatik Fakultät für Informatik Technische Universität Wien. Juli 2008. https://www.lri.fr/~nowak/misc/bakk.pdf

[3] Edwin K. P. Chong and Stanislaw H. Zak. **An Introduction to Optimization**. 4th Edition. Wiley. (January 14, 2013). ISBN-13: 978-1118279014

[4] Christopher M. Bishop. **Pattern Recognition and Machine Learning** (Information Science and Statistics). Springer (October 1, 2007). ISBN-13: 978-0387310732

[5] Bertrand Clarke, Ernest Fokoue, Hao Helen Zhang. **Principles and Theory for Data Mining and Machine Learning** (Springer Series in Statistics) Springer; 2009 edition (July 30, 2009). ISBN-13: 978-0387981345

[6] Hubert Nguyen. **GPU Gems 3**. Addison-Wesley Professional (August 12, 2007). ISBN-13: 978-0321515261. Also made available in its entirety online at https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_pref01.html

[7] Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, **JMLR 12**, pp. 2825-2830, 2011.

[8] C.-C. Chang and C.-J. Lin. *LIBSVM : a library for support vector machines.* **ACM Transactions on Intelligent Systems and Technology**, 2:27:1–27:27, 2011.

[9] J. Platt. *Fast training of support vector machines using sequential minimal optimization.* In A. Smola B. Schölkopf, C. Burges, editor, **Advances in Kernel Methods: Support Vector Learning**. MIT Press, Cambridge, MA, 1998.

[10] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. *A Practical Guide to Support Vector Classification.* http://www.ee.columbia.edu/~sfchang/course/spr/papers/svm-practical-guide.pdf