Dual Solver for the Multiplicative Kernel Structural SVM

Kun He, Boston University
hekun@cs.bu.edu
January 5, 2016

1. Introduction

This manuscript describes the implementation of a mini-batch dual solver for the multiplicative kernel structural SVM used in [1]. The solver is written from scratch (except for wrapping around a QP solver), and uses dual coordinate ascent style updates, similar to SMO [2, 3, 4], SDCA [5], and D. Ramanan's linear structural SVM solver [6]. The use of a mini-batch update strategy resulted in a 10x speed up over batch solutions (see [1]).

2. Preliminaries

First let's revisit the setup in [1]. We are given a training set $S = \{(x_i, y_i)\}_{i=1}^n$, where each training pattern $x_i \in \mathcal{X}$ is an image, and each training label $y_i = (B_i, \theta_i) \in \mathcal{Y}$ encodes the location (bounding box B) and pose (angle θ) of an object in that image. For simultaneous localization and pose estimation, our goal is to learn a scoring function f such that $y_i = \arg\max_{y\in\mathcal{Y}} f(x_i, y)$.

The scoring function f is parameterized as $f(x, y) = \langle \mathbf{w}, \Psi(x, y)\rangle$, where $\mathbf{w}$ is a parameter vector, and $\Psi(x, y)$ is a joint feature map. The inner product $\langle\cdot,\cdot\rangle$ is defined in a reproducing kernel Hilbert space $\mathcal{H}$, instantiated by a joint kernel function: $K(x, y, x', y') = \langle\Psi(x, y), \Psi(x', y')\rangle$. In [1], the joint kernel function is restricted to a multiplicative form, *i.e.*

$$K(x, y, x', y') = K_s(\Phi(x, B), \Phi(x', B')) \cdot K_p(\theta, \theta') \tag{1}$$

$$= \Phi(x, B)^\top \Phi(x', B') \cdot \exp\left(-\gamma\,\angle(\theta, \theta')^2\right) \tag{2}$$

Here, $\Phi(x, B)$ represents the feature vector extracted from the image region inside bounding box B in image x, and $\angle(\cdot,\cdot)$ is the angular distance. K_s is linear for efficiency considerations, and K_p is a RBF kernel.

2.1. SVM Primal

The learning formulation is a n-slack structural SVM [7]:

$$\min_{\mathbf{w},\xi} \frac{1}{2}\|\mathbf{w}\|_{\mathcal{H}}^2 + \frac{C}{n}\sum_{i=1}^n \xi_i \tag{3}$$

$$s.t.\ \langle\mathbf{w}, \Psi(x_i, y_i)\rangle - \langle\mathbf{w}, \Psi(x_i, y)\rangle \geq \Delta(y_i, y) - \xi_i,\ \ \forall i, \forall y \in \mathcal{Y} \tag{4}$$

where the task loss term $\Delta(y_i, y)$ encodes the penalty of predicting y instead of y_i, and C is a weighting parameter.

Note that we slightly depart from [7] and drop the explicit nonnegativity constraints on the slack variables, and allow y to freely range over $\mathcal{Y}$ in (4). But in fact this is equivalent to [7]'s formulation: in (4), assuming $\Delta(y_i, y_i) = 0$, setting $y = y_i$ gives the constraint $\xi_i \geq 0$. Therefore the nonnegativity constraints are naturally encoded in this formulation.

2.2. SVM Dual

For notational brevity, we use a shorthand:

$$\delta\Psi(i,y) \triangleq \Psi(x_i,y_i) - \Psi(x_i,y). \tag{5}$$

And it is easy to verify that

$$\langle \delta\Psi(i,y), \delta\Psi(j,y') \rangle = K(x_i,y_i,x_j,y_j) - K(x_i,y,x_j,y_j) - K(x_i,y_i,x_j,y') + K(x_i,y,x_j,y'). \tag{6}$$

Assigning a nonnegative dual variable $\alpha_{i,y}$ to each constraint (4) in the primal, and applying Lagrangian duality, the dual program can be derived as:

$$\max_{\boldsymbol{\alpha}\geq\mathbf{0}} \; -\frac{1}{2}\sum_{i,j=1}^{n}\sum_{y,y'\in\mathcal{Y}} \alpha_{i,y}\alpha_{j,y'}\langle \delta\Psi(i,y), \delta\Psi(j,y') \rangle + \sum_{i=1}^{n}\sum_{y\in\mathcal{Y}} \alpha_{i,y}\Delta(y_i,y) \tag{7}$$

$$s.t. \quad \sum_{y\in\mathcal{Y}} \alpha_{i,y} = \frac{C}{n}, \forall i \tag{8}$$

and the parameter vector (conceptually, since Ψ may not be explicit) is reconstructed as

$$\mathbf{w} = \sum_{i=1}^{n}\sum_{y\in\mathcal{Y}} \alpha_{i,y}\delta\Psi(i,y). \tag{9}$$

Remark: the sums over $y \in \mathcal{Y}$ might be problematic in the case of a continuous label space $\mathcal{Y}$, as the number of such y's is infinite. In the actual realization of this program we always use a discretized (and thus finite) version of the $\mathcal{Y}$ space, *e.g.* using 16 equally spaced viewpoint angles (see page 7 in [1]).

3. Mini-Batch Dual Coordinate Ascent

As usual, we use a cutting plane algorithm for optimizing the dual program. However, a batch cutting plane algorithm (*e.g.* the 1-slack reformulation [8] that we had tried) usually has undesirable scaling properties. Therefore a mini-batch algorithm was proposed in [1] to speed up learning. The idea is to incrementally generate cutting planes by traversing the training set in mini-batches, and update the model after each mini-batch is processed. Since the updates are applied to the dual variables $\boldsymbol{\alpha}$, this is essentially a dual coordinate ascent strategy.

The sketch of the algorithm: in iteration t,

1. sample a mini-batch of training pairs $\{(x_s, y_s)|s \in S_t\}$,
2. generate and cache cutting planes for $\forall s \in S_t$ using the current model $\mathbf{w}_{t-1}$,
3. update the dual program with all cached cutting planes and resolve,
4. update the cache to retain the active cutting planes.

3.1. Caching Cutting Planes with Working Sets

By *cutting plane*, we refer to a difference vector $\delta\Psi(i,y)$ as defined in (5) for some (i,y). Of course, we cannot explicitly construct $\delta\Psi(i,y)$. However, since we only need kernel evaluations, it suffices to store the quantities needed for kernel evaluations involving $\delta\Psi(i,y)$. Recall that

$$\delta\Psi(i,y) = \Psi(x_i,y_i) - \Psi(x_i,y) = \Psi(x_i,(B_i,\theta_i)) - \Psi(x_i,(B,\theta)), \tag{10}$$

it can be verified that caching the following four items is enough: $\{\Phi(x_i, B_i), \Phi(x_i, B), \theta_i, \theta\}$. In the remainder of this manuscript we will continue to refer to $\delta\Psi(i, y)$ for notational convenience, but readers should be aware of its actual representation.

We use a *working set*, denoted $\mathcal{W}_i$, for each training pair (x_i, y_i), to cache the generated cutting planes during learning. Conceptually, we write $\mathcal{W}_i = \{\delta\Psi(i, y) | \alpha_{i,y} > 0, y \neq y_i\}, \forall i$. Then, at any point during learning, the set of active cutting planes (*i.e.* support vectors) is the union of all the working sets. Further space saving is possible by creating a lookup table for feature vectors Φ indexed by (i, B). This way, each $\Phi(x_i, B_i), \forall i$ is cached at most once.

As is common practice, in our implementation removing inactive cutting planes from the working sets (step 4 of algorithm sketch) is only done occasionally, as constraints may re-enter the cache after being removed. A heuristic, that we borrowed from [8]'s implementation, is to check the working sets every D iterations, and remove cutting planes that have been inactive for at least L iterations. A reasonable choice can be $D = 20, L = 10$.

3.2. Generating Cutting Planes

Given (intermediate) model $\mathbf{w}$ and a tolerance parameter $\epsilon > 0$, constraints of the form (4) that are violated more than ϵ (cutting planes) are found via loss-augmented inference, and the corresponding $\delta\Psi(i, y)$'s are added to the working sets. Please refer to [1] for details of loss-augmented inference.

Remark: there is no restriction that only one cutting plane be generated for a training pair in each iteration. In fact, generating multiple *divserse* cutting planes in each iteration helps structural SVM training [9]. This is accomodated in our solver implementation.

3.3. Mini-Batch Sampling

Different strategies exist. We have tried uniform sampling, non-uniform (importance weighted) sampling, and sequentially going through a randomly permuted training set. The last strategy is what we ultimately chose; before each pass over the training set (epoch), we randomly permute the training examples. In [1]'s experiments we found 2 epochs is enough to train a good model. But of course, this is data-dependent.

3.4. Updating Dual Variables

Suppose at iteration t, we have sampled a mini-batch S_t and have generated a set of cutting planes $\{\delta\Psi(s, y_s^t) | \forall s \in S_t\}$ (for simplicity, assuming one cutting plane per training pair). Now we need to update the dual variables. Let $\boldsymbol{\alpha}(S_t) = \{\alpha_{s, y_s^t} | \forall s \in S_t\}$. A true dual coordinate ascent (more precisely, dual *block* coordinate ascent) solution would be to fix the values of all other dual variables, and only update $\boldsymbol{\alpha}(S_t)$, for which closed-form solution is likely available. An example where the batch size is 1 is Sequential Minimal Optimization (SMO), used in [2, 3, 4] and others.

We implemented SMO-style closed-form updates for $\boldsymbol{\alpha}(S_t)$. However, in our experiments, we found that when the problem size is not very large, updating *all* dual variables (by resolving the entire QP) after each mini-batch is actually efficient. The reason is that loss-augmented inference dominates training time in practice (more than 95% in our experiments, and many vision applications). This allowed us to simply resolve the entire QP after each mini-batch. The immediate benefit is that coding complexity is reduced (only need to call the QP solver); also, a potential benefit is faster convergence since in each iteration, updates are made to all dual variables instead of those in a single block.

Remark: in an online setting (*e.g.* [4]) or where loss-augmented inference is efficient enough, doing closed-form SMO updates may prove to be beneficial.

4. Algorithm Pseudocode

Now we are finally ready to describe the algorithm pseudocode. Algorithm 1 is the overall flow.

Algorithm 1 Dual solver for kernel structural SVM

```
 1: procedure SolveDual($S = \{1, \dots, n\}, \epsilon, C, D, L$)
 2:     $\mathcal{W}_i \leftarrow \phi,\ i = 1, \dots, n$                                ▷ init working sets
 3:     $t \leftarrow 0$
 4:     repeat
 5:         $t \leftarrow t + 1$
 6:         $\boldsymbol{\delta\Psi} \leftarrow \cup_{i=1}^{n}\{\delta\Psi(i, y)|y \in \mathcal{W}_i\}$                ▷ collect cutting planes
 7:         $\boldsymbol{\Delta} \leftarrow \cup_{i=1}^{n}\{\Delta(y_i, y)|y \in \mathcal{W}_i\}$       ▷ and their corresponding loss values
 8:         $(\boldsymbol{\alpha}, \boldsymbol{\xi}) =$ SolveQP($\boldsymbol{\delta\Psi}, \boldsymbol{\Delta}, \mathcal{W}, C$)                ▷ construct QP as in (7)(8)
 9:         if mod($t, D$) $= 0$ then
10:             $\mathcal{W}_i =$ RemoveInactive($\mathcal{W}_i, L$), $i = 1, \dots, n$
11:         end if
12:         $S_t =$ SampleMiniBatch($S, t$)
13:         $P_t =$ FindCuttingPlanes($\boldsymbol{\alpha}, \boldsymbol{\delta\Psi}, S_t, \boldsymbol{\xi}, \epsilon$)          ▷ $P_t$ is a set of $(i, y)$ pairs
14:         $\mathcal{W}_i \leftarrow \mathcal{W}_i \cup \{y|(i, y) \in P_t\}, \forall i \in S_t$                ▷ expand working sets
15:     until no constraint violated more than $\epsilon$ can be found
16: end procedure
```

Note that in Algorithm 1, again for notational convenience, we slightly modified the definition of working sets to only store the "violating" labels returned by `FindCuttingPlanes`, and treated $\delta\Psi$ as an oracle that can be freely queried by providing (i, y) pairs. However in reality, as discussed before, actual feature vectors need to be cached.

Next, the key subroutine of constructing and solving QP. We used the QP solver from [10]. The primal slack variables $\boldsymbol{\xi}$ need to be recovered as a result of solving the QP, since `FindCuttingPlanes` will need them to determine the amount of constraint violations. Also, note that the constraint in the actual QP should be an inequality as opposed to the equality in (8), since at any point the y's cached in a working set $\mathcal{W}_i$ is an incomplete enumeration of the entire $\mathcal{Y}$ space.

Algorithm 2 Constructing and solving QP

```
1: procedure SolveQP($\boldsymbol{\delta\Psi}, \boldsymbol{\Delta}, \mathcal{W}, C$)
2:     $\mathbf{G}_{(i,y),(j,y')} \leftarrow \langle\delta\Psi(i, y), \delta\Psi(j, y')\rangle, \forall\delta\Psi(i, y), \delta\Psi(j, y') \in \boldsymbol{\delta\Psi}, (i, y) \neq (j, y')$   ▷ Gram matrix
3:     $\boldsymbol{\alpha} = \arg\min_{\boldsymbol{\alpha}\geq\mathbf{0}} \frac{1}{2}\boldsymbol{\alpha}^\top\mathbf{G}\boldsymbol{\alpha} - \boldsymbol{\Delta}^\top\boldsymbol{\alpha},\ \ s.t. \sum_{y\in\mathcal{W}_i}\alpha_{i,y} \leq \frac{C}{n}, \forall i$      ▷ note the $\leq$ constraint
4:     for $i = 1$ to $n$ do
5:         $\xi_i = \max\{\max_{y\in\mathcal{W}_i}\{\boldsymbol{\Delta}_{i,y} - \sum_{(j,y')}\mathbf{G}_{(i,y),(j,y')} \cdot \alpha_{j,y'}\}, 0\}$
6:     end for
7: end procedure
```

References

[1] Kun He, Leonid Sigal, and Stan Sclaroff. Parameterizing object detectors in the continuous pose space. In *Proc. of the European Conference on Computer Vision (ECCV)*, volume 8692, pages 450–465, 2014. 1, 2, 3

[2] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*. MIT Press, January 1998. 1, 3

[3] Antoine Bordes, Léon Bottou, Patrick Gallinari, and Jason Weston. Solving multiclass support vector machines with LaRank. In *Proceedings of the 24th international conference on Machine learning*, pages 89–96. ACM, 2007. 1, 3

[4] Sam Hare, Amir Saffari, and Philip HS Torr. Struck: Structured output tracking with kernels. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 263–270. IEEE, 2011. 1, 3

[5] Shai Shalev-Shwartz and Tong Zhang. Stochastic dual coordinate ascent methods for regularized loss. *The Journal of Machine Learning Research*, 14(1):567–599, 2013. 1

[6] Deva Ramanan. Dual coordinate solvers for large-scale structural SVMs. *arXiv preprint arXiv:1312.1743*, 2013. 1

[7] Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. Large margin methods for structured and interdependent output variables. In *Journal of Machine Learning Research*, pages 1453–1484, 2005. 1

[8] Thorsten Joachims, Thomas Finley, and Chun-Nam John Yu. Cutting-plane training of structural SVMs. *Machine Learning*, 77(1):27–59, 2009. 2, 3

[9] Abner Guzman-Rivera, Pushmeet Kohli, and Dhruv Batra. Divmcuts: Faster training of structural SVMs with diverse M-best cutting-planes. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 316–324, 2013. 3

[10] Vojtech Franc and Václav Hlavác. A novel algorithm for learning support vector machines with structured output spaces. 2006. 4