Imposing Hard Constraints on Deep Networks: Promises and Limitations

Pablo Márquez-Neila Mathieu Salzmann Pascal Fua
École Polytechnique Fédérale de Lausanne (EPFL)
firstname.secondname@epfl.ch

Abstract

Imposing constraints on the output of a Deep Neural Net is one way to improve the quality of its predictions while loosening the requirements for labeled training data. Such constraints are usually imposed as soft constraints by adding new terms to the loss function that is minimized during training. An alternative is to impose them as hard constraints, which has a number of theoretical benefits but has not been explored so far due to the perceived intractability of the problem.

In this paper, we show that imposing hard constraints can in fact be done in a computationally feasible way and delivers reasonable results. However, the theoretical benefits do not materialize and the resulting technique is no better than existing ones relying on soft constraints. We analyze the reasons for this and hope to spur other researchers into proposing better solutions.

1. Introduction

Deep Neural Networks (DNNs) have become fundamental Computer Vision tools but can only give their full measure when enough training data is available. When there is not enough, good performance can still be obtained by applying well-established approaches, such as performing data augmentation, generating synthetic data, or imposing constraints on the network's output. Such constraints often are regularization constraints that act as a form of weak supervision. Sometimes, they are also used to make domain-specific knowledge explicitly available to the training algorithm to leverage unlabeled data. For example, symmetry constraints are introduced in the recent CNN approach to human pose estimation of [24] to regularize the prediction. Such constraints improve the quality of the DNN's predictions while loosening the requirements for labeled training data.

In the Deep Learning context, constraints are usually treated as *soft* ones and imposed by adding extra penalty terms to the standard loss function. While conceptually simple, this approach has two major drawbacks. First, it makes it necessary to wisely choose the relative importance of the different terms in the loss function, which is not easy. Second, this gives no guarantee that the constraints will be satisfied in practice. It has been shown that, in many cases, replacing the soft constraints by hard ones addresses both of these problems [5, 8]: The optimizer automatically chooses the relative weights and the constraints end up being satisfied to machine precision if a feasible solution exists. As a result, the algorithms relying on hard constraints are typically more robust and simpler to deploy. In this study, we investigate whether the same holds in the Deep Learning context.

There is a pervasive belief in the community that imposing hard constraints on Deep Nets is impractical [14] because typical networks involve millions of free parameters that need to be learned, which tends to overwhelm standard constrained optimization algorithms. Our contribution is to show that this is not the real problem and that, by using a Krylov subspace approach, it is in fact possible to solve the very large linear systems that these algorithms generate. As a result, our method, while slow, returns meaningful results.

However, somewhat surprisingly, we observed that imposing soft constraints instead of hard ones yields even better results, while being far less computationally demanding. Having studied the reasons for this counter-intuitive behavior, we have concluded that it arises from the fact that it is hard to guarantee that the linearized constraints used during the optimization are independent. This, in turn, opens perspectives on how to overcome the problem and eventually enable us to take full advantage of the power of hard constraints in the framework of Deep Learning.

2. Related Work

Given a labeled training set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i), 1 \leq i \leq N\}$ of N pairs of input vectors $\mathbf{x}_i$ and output vectors $\mathbf{y}_i$, Deep Learning can be understood as finding a mapping $\phi(\cdot, \mathbf{w})$ such that $\phi(\mathbf{x}_i, \mathbf{w}) \approx \mathbf{y}_i$ for all i, with $\mathbf{w}$ an N_P dimensional vector of parameters to be learned. This is usually achieved using Stochastic Gradient Descent (SGD) to find $\mathbf{w}^* = \arg\min_{\mathbf{w}} R(\mathbf{w})$, with $R(\mathbf{w}) = \frac{1}{N}\sum_i L(\phi(\mathbf{x}_i, \mathbf{w}), \mathbf{y}_i)$. R approximates the expected value

of the loss L, and is also known as the *risk* function.

A standard way to bias the result of this minimization is to add penalty terms to the risk function, for example to regularize the computation or introduce prior knowledge. Constraints imposed in this way are known as *soft* constraints because the computation yields a parameter vector $\mathbf{w}^*$ that achieves a compromise between minimizing the risk and satisfying the constraints. An alternative to using soft constraints is to use *hard* ones, which in our context means solving

$$\mathbf{w}^* = \min_{\mathbf{w}} R(\mathbf{w}) \ , \ \text{s.t.} \ C_k(\mathbf{w}) = 0 \ \text{for} \ 1 \leq k \leq N_c \ , \tag{1}$$

where the $C_k(\cdot)$ are the N_c constraints we want to enforce.

In traditional optimization, it is well known that minimizing an objective function that includes many penalty terms yields poor convergence properties [5, 8]: First, the optimizer is likely to minimize the constraint terms while ignoring the remaining terms of the objective function. Second, if one tries to enforce several constraints of different natures, the penalty terms are unlikely to be commensurate and one has to face the difficult problem of adequately weighing the various constraints.

Constrained optimization techniques offer a solution to both problems, essentially by also optimizing for the weights of the constraint terms as the optimization proceeds. In our field, this has been used to train a kernelized latent variable model to impose equality and inequality constraints [20] and to show that a Gaussian Process can be made to satisfy linear and quadratic constraints [19]. Similar techniques have also been proposed to constrain the output of Neural Nets before they became deep [17, 23]. However, the constrained optimization techniques used in these papers would not be applicable to DNNs. This is because they involve solving large linear systems of equations, even for moderate size problems. When dealing with the millions of parameters that define typical Deep Learning models, the numerical techniques at the heart of these approaches would have to handle matrices whose dimensions would also be that big. This would indeed "make a direct optimization of the constraints difficult", as stated in [15], if not downright impossible on available hardware.

In fact, the approach of [15] is the only one we know of that tries to address this dimensionality problem when imposing hard constraints on a Deep Network. In this image segmentation work, the constraints are linear and used to synthesize training labelings from the network output, which avoids having to explicitly impose the constraints while minimizing the loss function. This an effective but very specialized solution. By contrast, the problem we address in this paper is that of enforcing completely generic non-linear constraints.

3. Formalization

As discussed in Section 2, given a labeled training dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$ and a Deep Network architecture ϕ with parameters $\mathbf{w}$, training a Deep Net amounts to finding the parameter $\mathbf{w}^*$ that minimizes the risk function $R(\mathbf{w})$. In other words, $\mathbf{w}^*$ should ideally be such that $\forall i \leq N, \phi(\mathbf{x}_i, \mathbf{w}^*) \approx y_i$.

A specificity of Deep Learning is that it rarely, if ever, makes sense to impose hard constraints on the $\mathbf{w}$ parameters themselves, except for normalization that can usually be achieved more simply. Here, we therefore focus on the problem of imposing constraints on the network outputs. For example, in the experiment section we will consider 3D human body pose estimation from images. In this scenario the $\mathbf{x}$ vectors are the images and the $\mathbf{y}$ vectors represent the predicted locations of the person's joints. Therefore meaningful constraints are constraints on the relative predicted positions of the various joints. We will refer to them as *Data Dependent* constraints because they are expressed in the output space of the network, and thus explicitly take into account the input vectors.

Formally, let us consider a set of N_C such constraints $\{C_j(\cdot)\}_{j=1}^{N_C}$. We seek to enforce all of these constraints on each input vector. However, accurately predicting the output vectors of the elements of $\mathcal{D}$ already implicitly satisfies the constraints, assuming that the training samples satisfy them. Therefore, following [20], we introduce a set of unlabeled points $\mathcal{C} = \{\mathbf{x}'_k\}_{k=1}^{|\mathcal{C}|}$, which may or may not share elements with $\mathcal{D}$. The hard-constraint optimization problem of Eq. 1 can then be formulated as solving

$$\min_{\mathbf{w}} R(\mathbf{w}) \quad \text{s.t.} \ C_{jk}(\mathbf{w}) = 0 \quad \forall j \leq N_C, \forall k \leq |\mathcal{C}| \ , \tag{2}$$

where $C_{jk}(\mathbf{w}) = C_j(\phi(\mathbf{x}'_k; \mathbf{w}))$.

The corresponding soft-constraint problem typically becomes solving

$$\min_{\mathbf{w}} R(\mathbf{w}) + \sum_j \lambda_j \left(\sum_k C_{jk}(\mathbf{w})^2 \right) \ , \tag{3}$$

where the λ_j parameters are positive scalars that control the relative contribution of each constraint. This is a compromise between minimizing the loss and satisfying the constraints. Too small a value of λ_j will result in the corresponding constraint being ignored, and too large a one in the other terms, especially the loss, being ignored.

The soft-constraint problem of Eq. 3 can be solved using techniques that have now become standard in the Deep Learning literature, such as SGD with momentum or using the Adam solver [10]. Even though constrained SGD methods have been proposed [7], they have not been investigated in the Deep Learning context where the dimension of the

vector $\mathbf{w}$ is very large, which is what we will do in the following section.

4. Dealing with Millions of Variables

In this section, we first introduce standard Lagrangian methods to constrained optimization. Unfortunately, they cannot be directly used for our problem because (i) the parameter vector $\mathbf{w}$ of Eq. 2 has a large dimensionality, often in the order of millions or tens of millions, (ii) the constraints are not always guaranteed to be independent and might even be slightly incompatible, which introduces numerical instabilities, and (iii) the training datasets $\mathcal{D}$ and $\mathcal{C}$ of Eq. 2 are usually large as well. In the remainder of this section, we address each one of these issues in turn. In this work, we focus on equality constraints, but we will argue below that our approach could naturally extend to inequality ones.

4.1. Karush-Kuhn-Tucker Conditions

An effective way to solve a constrained optimization problem such as the one of Eq. 2 is to find a stationary point of the Lagrangian

$$\begin{aligned} &\min_{\mathbf{w}} \max_{\Lambda} \mathcal{L}(\mathbf{w}, \Lambda), \\ &\text{with } \mathcal{L}(\mathbf{w}, \Lambda) = R(\mathbf{w}) + \Lambda^T \mathbf{C}(\mathbf{w}), \end{aligned} \tag{4}$$

where $\mathcal{L}$ is the Lagrangian and Λ the $(|\mathcal{C}| \cdot N_C)$-dimensional vector of *Lagrange multipliers* [5]. They play the same role as the λ_j of Eq. 3 but do not need to be set by hand. Since neither the loss nor the constraints are convex or linear, given an initial value $\mathbf{w}_0$ of $\mathbf{w}$, one computes an increment $\mathrm{d}\mathbf{w}$ and the value of Λ by solving a linearized version of Eq. 4, replacing $\mathbf{w}$ by $\mathbf{w} + \mathrm{d}\mathbf{w}$, and iterates.

If the loss L is a generic differentiable function, the increment $\mathrm{d}\mathbf{w}$ is normally computed at each iteration by solving the linear system

$$\begin{bmatrix} \eta I & \frac{\partial \mathbf{C}}{\partial \mathbf{w}}^T \\ \frac{\partial \mathbf{C}}{\partial \mathbf{w}} & 0 \end{bmatrix} \begin{bmatrix} \mathrm{d}\mathbf{w} \\ \Lambda \end{bmatrix} = \begin{bmatrix} -\frac{\partial R}{\partial \mathbf{w}} \\ -C(\mathbf{w}) \end{bmatrix} , \tag{5}$$

which amounts to projected gradient descent. The projection is performed onto the hyperplanes of linearized constraints. The linear system of Eq. 5 is derived from the Karush-Kuhn-Tucker (KKT) conditions that $\mathrm{d}\mathbf{w}$ and Λ must satisfy.

When the risk R is a sum of squared residuals, which is typical in regression problems, it is generally more effective to replace the projected gradient by Gauss-Newton (GN) or Levenberg-Marquardt (LM). The iteration scheme remains the same but the KKT conditions become

$$\begin{bmatrix} \mathbf{J}^T\mathbf{J} + \eta I & \frac{\partial \mathbf{C}}{\partial \mathbf{w}}^T \\ \frac{\partial \mathbf{C}}{\partial \mathbf{w}} & 0 \end{bmatrix} \begin{bmatrix} \mathrm{d}\mathbf{w} \\ \Lambda \end{bmatrix} = \begin{bmatrix} -\mathbf{J}^T\mathbf{r}(\mathbf{w}_t) \\ -C(\mathbf{w}_t) \end{bmatrix}, \tag{6}$$

where $\mathbf{r}$ is the vector of residuals and $\mathbf{J} = \frac{\partial \mathbf{r}}{\partial \mathbf{w}}$ its Jacobian matrix evaluated at $\mathbf{w}_t$. The full derivation can be found in [6].

4.2. Krylov Subspace Methods

To perform the minimization, we must solve at each iteration a linear system of the form $\mathbf{B}\mathbf{v} = \mathbf{b}$ where the exact form of $\mathbf{B}$ and $\mathbf{b}$ depends on whether we enforce the KKT conditions of Eq. 5 or 6. In either case, the immense dimensionality of $\mathbf{w}$ precludes explicitly computing or storing the matrix $\mathbf{B}$. Instead, we solve the linear system using a Krylov subspace method [11].

Krylov subspace methods solve linear systems by iteratively computing elements of the base of the Krylov subspace and finding approximate solutions of the linear system in that subspace. The main advantage of these methods is that they do not need an explicit representation of $\mathbf{B}$. Instead, the user has to provide a function that receives a vector $\mathbf{v}$ and returns the matrix-vector product $\mathbf{B}\mathbf{v}$. This is usually much faster and less memory consuming than explicitly computing $\mathbf{B}$.

Computing matrix-vector products with the matrices defined in Eqs. 5 and 6 involves products of Jacobian matrices with vectors, which are essentially directional derivatives. The *Pearlmutter trick* [16, 21] is a well-known technique that computes Jacobian-times-vector and vector-times-Jacobian products leveraging the backpropagation procedures common in neural networks. We call *R-op* to the Jacobian-times-vector operation, and we will write $\mathrm{Rop}(\mathbf{f}, \mathbf{w}, \mathbf{v})$ to express the backpropagation procedure that computes the directional derivative of the multi-dimensional function $\mathbf{f}$ with respect to $\mathbf{w}$ evaluated in the direction $\mathbf{v}$. That is,

$$\mathrm{Rop}(\mathbf{f}, \mathbf{w}, \mathbf{v}) \equiv \frac{\partial \mathbf{f}}{\partial \mathbf{w}} \mathbf{v}. \tag{7}$$

Similarly, we call *L-op* the backpropagation procedure that computes the *adjoint directional derivative*, *i.e.*, the vector-times-Jacobian product. That is,

$$\mathrm{Lop}(\mathbf{f}, \mathbf{w}, \mathbf{v}) \equiv \mathbf{v}^T \frac{\partial \mathbf{f}}{\partial \mathbf{w}}. \tag{8}$$

R-op and *L-op* are available as GPU operators in many deep learning packages such as *Theano* [1].

Given that *R-op* and *L-op* are available, a Krylov subspace method can solve the linear systems of Eq. 5 and Eq. 6 using Algorithms 1 and 2, respectively. Algorithm 1 computes the matrix-vector product for the matrix of Eq. 5, and Algorithm 2 for the matrix of Eq. 6.

Algorithm 1 Compute $\mathbf{B}\mathbf{v}$ to solve Eq. 5

Input: Vector $\mathbf{v}$

Output: Product $\mathbf{B}\mathbf{v}$

$\mathbf{v}_1, \mathbf{v}_2 \leftarrow$ Split $\mathbf{v}$ in two at position N_P
$\mathbf{a} \leftarrow \eta\mathbf{v}_1 + \mathrm{Lop}(\mathbf{C}, \mathbf{w}, \mathbf{v}_2)$
$\mathbf{b} \leftarrow \mathrm{Rop}(\mathbf{C}, \mathbf{w}, \mathbf{v}_1)$
return Concatenation of $\mathbf{a}$ and $\mathbf{b}$
end

Algorithm 2 Compute $\mathbf{Bv}$ to solve Eq. 6
Input: Vector $\mathbf{v}$
Output: Product $\mathbf{Bv}$
$\mathbf{v}_1, \mathbf{v}_2 \leftarrow$ Split $\mathbf{v}$ in two at position N_P
$\mathbf{Jv} \leftarrow \mathrm{Rop}(\mathbf{r}, \mathbf{w}, \mathbf{v}_1)$
$\mathbf{JJv} \leftarrow \mathrm{Lop}(\mathbf{r}, \mathbf{w}, \mathbf{Jv})$
$\mathbf{a} \leftarrow \mathbf{JJv} + \eta\mathbf{v}_1 + \mathrm{Lop}(\mathbf{C}, \mathbf{w}, \mathbf{v}_2)$
$\mathbf{b} \leftarrow \mathrm{Rop}(\mathbf{C}, \mathbf{w}, \mathbf{v}_1)$
return Concatenation of $\mathbf{a}$ and $\mathbf{b}$
end

4.3. Krylov Subspace Methods for Ill-conditioned Constraints

Although Krylov subspace methods allow us to solve high-dimensional linear systems, the solution might still suffer from numerical instabilities. In particular, the matrices of Eqs. 5 and 6 are usually singular or ill-conditioned as a consequence of incompatible or repeated constraints, which typically arise when linearizing constraints for a large number of samples. As shown in Table 1, different Krylov subspace methods impose different conditions on the matrix $\mathbf{B}$, and not all of them can handle ill-conditioned matrices. Since our matrices are squared, symmetric, not positive-definite, and, crucially, ill-conditioned, we chose the *minimal residual method with QLP decomposition* (MINRES-QLP) [3], an improved version of the older (MINRES) [12], as our Krylov subspace method.

4.4. Stochastic Active Constraints

While Krylov subspace methods reduce the memory required to solve Eq. 5, the size of the system still depends on the number of constraints imposed, which is in the order of the number of training samples. A simple method for reducing this size consists of selecting a subset $\hat{\mathcal{C}} \subset \mathcal{C}$ of *active constraints* from the unlabeled dataset at each iteration, in a similar way as stochastic gradient descent does with the labeled dataset.

In some of our experiments, instead of randomly sampling $\hat{\mathcal{C}}$, we look for a subset of elements with the largest violation of the constraint functions. That is,

$$\hat{\mathcal{C}} = \underset{\hat{\mathcal{C}} \subset \mathcal{C}}{\arg\max} \sum_{\mathbf{x}' \in \hat{\mathcal{C}}} \underset{j}{\mathrm{median}} \, |C_j(\phi(\mathbf{x}'; \mathbf{w}_t))| \quad \text{s.t.} \quad |\hat{\mathcal{C}}| < N_{\hat{\mathcal{C}}}. \tag{9}$$

This corresponds to *hard sample mining* for constraints. It is easily extended to *inequality* constraints by treating them as equality constraints when they are violated and ignoring them otherwise. The active constraints are then chosen among the violated ones.

4.5. Constrained Adam

The step computed in Eq. 5 is a projected gradient. Pure gradient descent has known issues with DNNs: it is sensitive to the chosen learning rate, slow and prone to fall in bad local minima. Methods like momentum gradient descent solve these problems by averaging the gradient over several iterations, and others like Adagrad [4] or Adadelta [22] compute a parameter-wise learning rate based on the previous updates of every parameter. Adam [10] combines momentum with a parameter-wise learning rate.

We extend our constrained training method with an Adam-like step. At every iteration, we update first and second order moment vectors $\mathbf{m}$ and $\mathbf{v}$ as

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1) \frac{\partial R}{\partial \mathbf{w}}, \tag{10}$$

$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2) \left(\frac{\partial R}{\partial \mathbf{w}} \right)^{\circ 2}, \tag{11}$$

for given hyperparameters β_1 and β_2 and partial derivatives evaluated at $\mathbf{w}_t$. β_1 and β_2 control the exponential decay rates of the vectors $\mathbf{m}$ and $\mathbf{v}$. We set them to 0.9 and 0.999 respectively, as proposed in [10]. The linear system of the constrained version of Adam uses the vector $\mathbf{m}$ as the current gradient and $\sqrt{\mathbf{v}}$ as the parameter-wise learning rate. This yields

$$\begin{bmatrix} \eta f \, \mathrm{diag}\left(\sqrt{\mathbf{v}_{t+1}} + \epsilon\right) & \frac{\partial \mathbf{C}}{\partial \mathbf{w}}^T \\ \frac{\partial \mathbf{C}}{\partial \mathbf{w}} & 0 \end{bmatrix} \begin{bmatrix} \mathrm{d}\mathbf{w} \\ \Lambda \end{bmatrix} = \begin{bmatrix} -\mathbf{m}_{t+1} \\ -C(\mathbf{w}_t) \end{bmatrix}, \tag{12}$$

with the correction factor $f = \dfrac{\sqrt{1 - \beta_2^{t+1}}}{1 - \beta_1^{t+1}}$. Algorithm 1 can be easily modified to compute matrix-vector products with this constrained Adam matrix.

5. Experiments

To compare the respective merits of soft and hard constraints when used in conjunction with Deep Networks, we used the well-known and important problem of 3D human pose estimation from a single view. We performed our experiments on the *Walking* sequence of the *Human3.6m* dataset [2, 9]. It consists of 162008 color images that were cropped to 128×128 pixels. The ground-truth data for each image corresponds to the 3D coordinates of 17 joints of the human body as shown if Figure 1. We split the dataset into 80%-20% subsets for training and testing, respectively.

To define the network ϕ, we used an architecture that is a composition of convolutional layers, max-pooling, ReLU

	Squared	Symmetric	Pos-definite	Full rank
Conjugate gradient	Yes	Yes	Yes	Yes
Symmetric LQ [12]	Yes	Yes	No	Yes
Minimal residual [12]	Yes	Yes	No	No
Minimal residual + QLP [3]	Yes	Yes	No	No
Generalized minimal residual [18]	Yes	No	No	No
Biconjugate gradient	No	N/A	N/A	Yes

Table 1. Krylov subspace methods for solving a linear system $\mathbf{Bv} = \mathbf{b}$ and their requirements for the matrix $\mathbf{B}$.

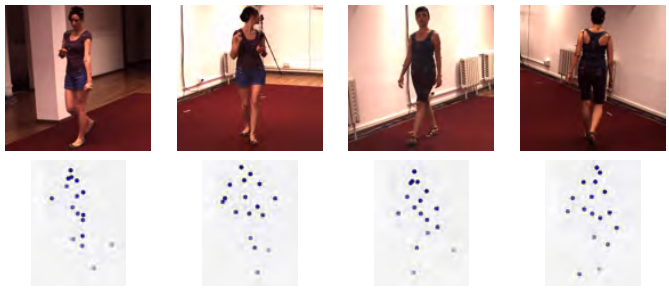


Figure 1. Samples from the Human3.6m dataset. Our models are trained to map from 128×128 color images, as those of the top row, to the 3D pose of 17 joints, as shown in the bottom row.

non-linearities and fully-connected layers in the following order:

$$\begin{aligned} \phi \quad = \quad & \mathcal{A}_{51} \circ \mathcal{R} \circ \mathcal{A}_{1024} \circ \mathcal{R} \circ \mathcal{A}_{1024} \circ \mathcal{C}_{8,3\times3} \\ & \circ \mathcal{R} \circ \mathcal{P}_{2\times2} \circ \mathcal{C}_{8,5\times5} \circ \mathcal{R} \circ \mathcal{P}_{4\times4} \circ \mathcal{C}_{8,5\times5}, \end{aligned} \tag{13}$$

where $\mathcal{C}_{a,b}$ is a convolutional layer of a filters with size b, $\mathcal{P}_a$ is a max-pooling layer with block size a, $\mathcal{R}$ is the ReLU non-linearity and $\mathcal{A}_a$ is an affine transformation, or fully connected layer, with a outputs. Layers of type $\mathcal{C}$ and $\mathcal{A}$ are parameteric, but we omitted the parameters for notational simplicity. Even though this architecture is not state-of-the-art anymore, it still gives reasonably good results while remaining simple enough to perform numerous experiments. Given that are we not interested in absolute performance but in comparing performances obtained using different approaches to imposing constraints, this is what we need.

We took the supervised loss L to be the squared distance between the ground-truth and the prediction for each joint. This can be expressed as

$$L(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{17 \cdot 3} \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2 \,, \tag{14}$$

where each $\mathbf{y}_i$ represents the true 3D position of one the 17 joints and $\hat{\mathbf{y}}_i$ the position predicted by the network from the corresponding image.

We first trained the network ϕ by minimizing this loss without imposing either hard or soft constraints. We ran 100 epochs of the Adam solver [10] with a minibatch size of 128. We take the model that reached the smallest error on the validation set as our *unconstrained* model.

We then defined six symmetry constraints: Equal lengths of left and right arms, forearms, legs and calves, equal distance from the two shoulders to the spine, and equal distance of the two hips to the spine. They are different from those of [24] but similar in spirit. The constraint functions are written as

$$\begin{aligned} C_{jk} \equiv C_j(\hat{\mathbf{y}}_k) \quad = \quad & \|\hat{\mathbf{y}}_{k,\mathit{joint}(j,1)} - \hat{\mathbf{y}}_{k,\mathit{joint}(j,2)}\| \\ & - \|\hat{\mathbf{y}}_{k,\mathit{joint}(j,3)} - \hat{\mathbf{y}}_{k,\mathit{joint}(j,4)}\| \,, \end{aligned} \tag{15}$$

where j and k are the constraint and the sample indices, respectively. $\hat{\mathbf{y}}_{k,\mathit{joint}(j,m)}$ is the predicted 3D position of the joint $\mathit{joint}(j,m)$ for data sample k, and $\mathit{joint}(\cdot,\cdot)$ is an auxiliary function indexing the joints involved in every constraint. Its complete definition is given in Table 2. A constraint C_{jk} is therefore met when the distance between joints $\mathit{joint}(j,1)$ and $\mathit{joint}(j,2)$ is equal to the distance between $\mathit{joint}(j,3)$ and $\mathit{joint}(j,4)$.

We trained the following constrained models in several different ways:

Soft-SGD: Standard stochastic gradient descent to minimize the loss of Eq. 3. We tried different values of the λ_j parameters of Eq. 3, which we take to all be equal to each other, and of the learning rate η.

Soft-Adam: Same as **Soft-SGD**, but using the Adam solver instead. In this case the learning rate was fixed to $\eta = 10^{-3}$ because Adam is very insensitive to that choice.

Hard-SGD: Our method using Eq. 5 solved using MINRES-QLP. We tried different parameters for the learning rate η.

Hard-Adam: Same as **Hard-SGD**, but using our Adam version of Eq. 12.

All constrained models were trained on the same training dataset $\mathcal{D}$ we used to learn the unconstrained model, using again minibatches of 128 samples. Similarly, we used the same dataset $\mathcal{C}$ on which we enforced the constraints for all the constrained models. There, we used either random

	$m=1$	$m=2$	$m=3$	$m=4$
$j=1$	left shoulder	left elbow	right shoulder	right elbow
$j=2$	left elbow	left hand	right elbow	right hand
$j=3$	left hip	left knee	right hip	right knee
$j=4$	left knee	left heel	right knee	right heel
$j=5$	chest	left shoulder	chest	right shoulder
$j=6$	pelvis	left hip	pelvis	right hip

Table 2. Joints for the symmetry constraint functions of Eq. 15. Every cell contains the value *joint*(j, m).

minibatches of 128 samples from $\mathcal{C}$ or performed hard constraint mining according to Eq. 9, keeping only 16 samples at a time. We used two metrics for evaluation purposes:

- **Prediction error**. Mean of the Euclidean distances between the predictions of the joint positions and the real positions, averaged over the whole validation dataset:

$$\text{error}(\mathcal{Y}, \hat{\mathcal{Y}}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{17}\sum_{m=1}^{17}\|\mathbf{y}_{i,m} - \hat{\mathbf{y}}_{i,m}\|.$$

- **Median constraint violation**. Median of the absolute values of all C_{jk}.

We report our comparative results in Fig. 2. **Soft-Adam** yields the best results, closely followed by **Hard-SGD**. Generally speaking, the Hard-constraint methods perform similarly to the soft-constraint methods in both metrics, instead of being better as we had hoped. In particular, the hard-constraint methods do not enforce the constraints perfectly, not even on the training data. This is because we can only use a subset of the constraints at each iteration, as explained in Section 4.4. As a result, constraints that were satisfied at one iteration can stop being satisfied at the next.

The parameter-wise adaptive learning rate of Adam leads to convergence even when using high values of λ without causing numerical instabilities, which makes it relatively easy to choose λ. Conversely, **Soft-SGD** is very sensitive to the choice of both the learning rate and λ because it lacks the adaptability of Adam. Surprisingly, Adam does not perform well with hard-constraint methods. We speculate that this is a consequence of the Adam momentum not dealing well with the projection onto the linearized constraint surface at each iteration.

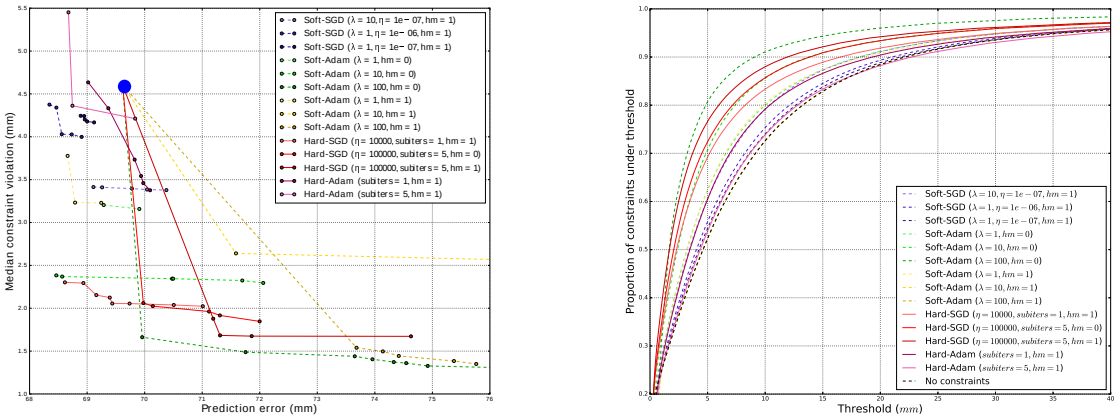
We implemented all the methods in Python with Theano [1] using the MINRES-QLP implementation from [13], and ran the experiments on an nVidia Titan Z. **Soft-SGD** and **Soft-Adam** required between 0.8 and 1.6 seconds per iteration (s/it) respectively, while **Hard-SGD** required between 3 s/it and 32 s/it, depending on the properties of the linear system solved at each update. **Hard-Adam** ran at 103 s/it. As expected, hard-constraint techniques are on average between 10 and 100 times slower than their soft-constraint counterparts.

6. Discussion

The human pose estimation experiments described in the previous section indicate that learning with hard constraints is computationally tractable, if more expensive than with soft constraints, and yields exploitable results. However, it does not guarantee either perfect satisfaction of the constraints, not even in the training data, or better performance than using soft constraints in conjunction with the Adam solver [10], which is not what we expected to happen when we started working on this problem.

6.1. Interpretation

This could of course be a fluke of the specific network architecture and constraint set we used in our experiments. In this section, we argue here that this not the case and points to a fundamental problem with imposing hard constraints on Deep Nets that needs to be addressed: Lagrangian methods applied to non-linear loss and constraint functions involve repeatedly linearizing and solving the KKT conditions of Eqs. 5 or 6, which means solving linear systems to project the potential solution vectors onto the intersection of a set of hyperplanes. The Krylov subspace approach we introduced in Section 4.3 has made those computationally tractable despite the enormous size of the linear systems while the MINRES-QLP approach [3] to solving them delivers numerical stability even though the matrices are ill-conditioned. However, we are still left with the problem that we cannot enforce all the constraints all the time and had to resort to using a randomly chosen subset of constraints at each iteration. As already observed, this means that constraints that are satisfied at one iteration can stop being satisfied at the next. The problem is compounded by the fact that the constraints $C_j(f(\mathbf{x}'_k; \mathbf{w}))$ of Eq. 2 are *data-dependent*, meaning they depend not only on the parameter vector $\mathbf{w}$ but also of the data vector $\mathbf{x}'$ at which they are evaluated. This means that we cannot guarantee that their linearizations are linearly independent from each other, which makes the KKT systems even more ill-conditioned, making the work of MINRES-QLP even more difficult. We believe this to be the root-cause of our difficulties.



(a) (b)

Figure 2. Comparison of soft and hard constraint methods for human pose estimation. (a) Comparison of average errors vs. median of absolute values of the constraints. Every dot represents a model. The large blue dot represents the unconstrained model. Lines connect models obtained with the same configuration but different number of training iterations. (b) Proportion of constraints met vs. threshold for different methods. In both plots the line style indicates the class of the method: solid lines for hard constraints and dashed lines for soft constraints.

6.2. Demonstration on a Synthetic Example

To demonstrate this, we now introduce a simple synthetic problem with mildly incompatible constraints, but with as many variables as in the previous Deep Nets. We then show that it gives rise to the same behavior as the one we reported in Section 5, that is, treating the constraints as hard constraints can be made to work but, in the end, treating them as soft constraints remains preferable if we have to work with different constraint subsets at each iteration.

Let us consider the hard-constraint problem of solving

$$\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}-\mathbf{x}_0\|^2 \text{ s. t. } \|\mathbf{w}-\mathbf{c}_i\|-10=0,\ 1\leq i\leq 200 \tag{16}$$

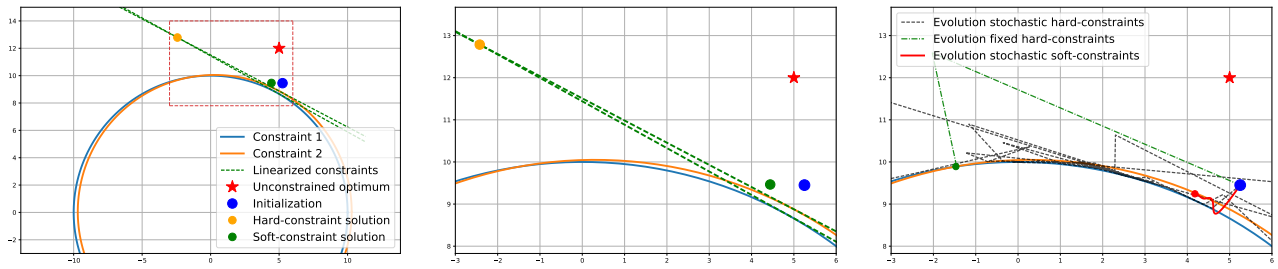
and the corresponding soft-constraint one of solving

$$\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}-\mathbf{x}_0\|^2 + \lambda \sum_{1\leq i\leq 200} \left(\|\mathbf{w}-\mathbf{c}_i\|-10\right)^2 , \tag{17}$$

when $\mathbf{w}$, $\mathbf{x}_0$, and the $\mathbf{c}_i$ are vectors of dimension d. In other words, we look for a vector $\mathbf{w}$ that is closest to a fixed vector $\mathbf{x}_0$, while being at the intersection of 200 different hyperspheres. Each one is centered at $\mathbf{c}_i$, which we take to be normally distributed around the origin with a variance of 0.01 that is two order of magnitude smaller than the hypersphere radius. In other words, all these constraints are very similar but slightly incompatible. For the soft-constraint problem, we take λ to be 100.

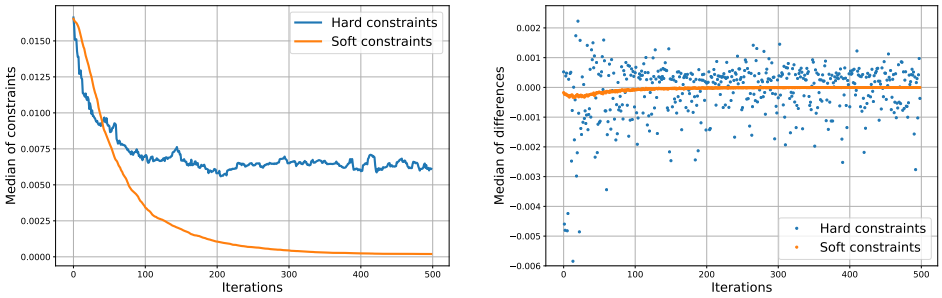
Fig. 3(a,b) depicts a single iteration where only two constraints are active and $d = 2$. We choose these two constraints to be very similar but slightly incompatible. Linearizing them therefore results in hyperplanes—lines in this case—that are almost parallel and intersect far away. In a sense, this a worst-case scenario for hard-constraints, with the linear manifolds intersecting far from the original constraints. As a result, imposing the KKT conditions by solving the linear system of Eq. 5 sends the candidate solution towards the intersection of the two hyperplanes and far away from the real constraint surfaces. If we repeated this operation using the *same* two constraints, $\mathbf{w}$ would eventually converge to the one feasible point at the intersection of the two constraint circles nevertheless. However, as discussed in Section 4.4, the set of active constraints changes at each iteration, resulting in an erratic behavior of $\mathbf{w}$ as the optimization progresses, as shown in Fig. 3(c).

To simulate a problem of a size roughly comparable to that of a Deep Learning one, we solved the hard- and soft-constraint problems of Eq. 17 with $d = 1e6$. To enforce the hard constraints, we used the method of Eq. 5. For the soft ones, we used stochastic gradient descent without momentum. In both cases we used a subset of 20 randomly chosen active constraints at every iteration. Fig. 4(a) depicts the resulting evolution of the median of the absolute value of the 200 constraints over 500 iterations. In this simple experiment, not only do soft constraints perform much better than hard constraints, but their behavior is smoother and more stable. To quantify how common it is for KKT updates to make the constraints *less* well satisfied as opposed to better satisfied, which is what they are designed to do, we computed the value of the active constraints before and after updating the parameters. In Fig. 4(b) we plot the differences



(a) (b) (c)

Figure 3. Synthetic experiment with only 2 active constraints. (a) One iteration starting from the blue dot and leading to either the green dot when using soft constraints or the yellow one when using hard constraints. (b) Zoomed in version of (a). (c) The optimization path for many iterations starting from the blue dot, when using soft constraints (red line), the same two hard constraints all the time (green dotted line), and different pairs of constraints at every iteration (back dotted line). The latter is totally erratic.



(a) (b)

Figure 4. Hypersphere experiment of Eq. 16 and 17. (a) Median of the absolute value of all the 200 constraint functions at every iteration. (b) Differences between the absolute value of the *active* constraints before and after the update step at every iteration. A positive value indicates that the update step deteriorated the quality of the active constraints. Note that these are values of the original constraints, as opposed to the linearized ones.

of the median of the absolute value of the constraints before and after updating. Negative values therefore indicate that the active constraints are better satisfied after update than before. Conversely positive values denote a degradation. In the soft-constrain case, there is initially a steady improvement until the algorithm stabilizes. By contrast, in the hard-constraint case, the behavior is far more unpredictable and chaotic, while eventually yielding a similar value of the loss.

To quantify how common it is for KKT updates to make the constraints *less* well satisfied as opposed to better satisfied, which is what they are designed to do, we computed the value of the active constraints before and after updating the parameters. In Fig. 4(b) we plot the differences of the median of the absolute value of the constraints before and after updating. Negative values therefore indicate that the active constraints are better satisfied after update than before. Conversely positive values denote a degradation. In the soft-constrain case, there is initially a steady improvement until the algorithm stabilizes. By contrast, in the hard-constraint case the behavior is far more unpredictable and chaotic, while eventually yielding a similar value of the loss.

7. Conclusion

We have shown that it is practical train a Deep Network architecture while imposing hard constraints on its output. To this end, we have developed a Lagrangian method that relies on a Krylov subspace approach to solving the resulting very large linear systems. Unfortunately, our experiments have shown that our approach performs well, but not better than using a soft-constraint approach as we had hoped.

We attribute this negative result the fact that, to keep the computational cost within reasonable limits, we had to choose a new subset of active constraints at every iteration instead of using all constraints all the time, thus forgetting the effect of all previously used ones. This is clearly suboptimal as successful hard-constraint methods do not normally remove elements from the pool of active constraint, at least when dealing when equality constraints instead of inequality ones. Implementing this in the Deep Network context is far from trivial, but we hope that this paper might inspire other researchers to look into it.

References

[1] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun 2010. Oral Presentation.

[2] C. S. Catalin Ionescu, Fuxin Li. Latent structured models for human pose estimation. In *International Conference on Computer Vision*, 2011.

[3] S.-C. T. Choi, C. C. Paige, and M. A. Saunders. MINRES-QLP: A Krylov subspace method for indefinite or singular symmetric systems. *SIAM Journal on Scientific Computing*, 33(4):1810–1836, 2011.

[4] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. Technical Report UCB/EECS-2010-24, EECS Department, University of California, Berkeley, Mar 2010.

[5] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 1987.

[6] P. Fua, V. Aydin, R. Urtasun, and M. Salzmann. Least-Squares Minimization Under Constraints. Technical report, EPFL, 2010.

[7] G. Gasso, A. Pappaioannou, M. Spivak, and L. Bottou. Batch and Online Learning Algorithms for Nonconvex Neyman-Pearson Classification. *ACM Transactions on Intelligent Systems and Technology*, 2(3), 2011.

[8] P. Gill, W. Murray, and M. Wright. *Practical Optimization*. Academic Press, 1981.

[9] C. Ionescu, D. Papava, V. Olaru, and C. Sminchisescu. Human3.6m: Large scale datasets and predictive methods for 3d human sensing in natural environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(7):1325–1339, jul 2014.

[10] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2014.

[11] A. N. Krylov. On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined. *News of Academy of Sciences of the USSR*, 1931.

[12] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12(4):617–629, 1975.

[13] R. Pascanu. Theano optimize. `https://github.com/pascanur/theano_optimize`, 2013.

[14] D. Pathak, P. Krahenbuhl, and T. Darrell. Constrained Convolutional Neural Networks for Weakly Supervised Segmentation. In *Conference on Computer Vision and Pattern Recognition*, 2015.

[15] D. Pathak, P. Krhenbhl, and T. Darrell. Constrained convolutional neural networks for weakly supervised segmentation. In *IEEE International Conference on Computer Vision (ICCV)*, pages 1796–1804, Dec 2015.

[16] B. A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6:147–160, 1994.

[17] J. C. Platt and A. H. Barr. Constrained differential optimization. Technical report, California Institute of Technology Pasadena, 1988.

[18] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[19] M. Salzmann and R. Urtasun. Implicitly Constrained Gaussian Process Regression for Monocular Non-Rigid Pose Estimation. In *Advances in Neural Information Processing Systems*, December 2010.

[20] A. Varol, M. Salzmann, P. Fua, and R. Urtasun. A Constrained Latent Variable Model. In *Conference on Computer Vision and Pattern Recognition*, 2012.

[21] O. Vinyals and D. Povey. Krylov subspace descent for deep learning. In N. D. Lawrence and M. A. Girolami, editors, *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics (AISTATS-12)*, volume 22, pages 1261–1268, 2012.

[22] M. D. Zeiler. Adadelta: An adaptive learning rate method. *CoRR*, 2012.

[23] S. Zhang and A. Constantinides. Lagrange programming neural networks. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(7):441–452, 1992.

[24] T. Zhou, M. Brown, N. Snavely, and D. Lowe. Unsupervised Learning of Depth and Ego-Motion from Video. In *Conference on Computer Vision and Pattern Recognition*, 2017.