Explicitly Imposing Constraints in Deep Networks via Conditional Gradients Gives Improved Generalization and Faster Convergence

Sathya N. Ravi, Tuan Dinh, Vishnu Suresh Lokhande, Vikas Singh
{ravi5@wisc.edu}, {tuandinh, lokhande, vsingh}@cs.wisc.edu

Abstract

A number of results have recently demonstrated the benefits of incorporating various constraints when training deep architectures in vision and machine learning. The advantages range from guarantees for statistical generalization to better accuracy to compression. But support for general constraints within widely used libraries remains scarce and their broader deployment within many applications that can benefit from them remains under-explored. Part of the reason is that Stochastic gradient descent (SGD), the workhorse for training deep neural networks, *does not natively deal with constraints with global scope very well*. In this paper, we revisit a classical first order scheme from numerical optimization, Conditional Gradients (CG), that has, thus far had limited applicability in training deep models. We show via rigorous analysis how various constraints can be naturally handled by modifications of this algorithm. We provide convergence guarantees and show a suite of *immediate benefits* that are possible — from training ResNets with fewer layers but better accuracy simply by substituting in our version of CG to faster training of GANs with 50% fewer epochs in image inpainting applications to provably better generalization guarantees using efficiently implementable forms of recently proposed regularizers.

Introduction

The learning or fitting problem in deep neural networks in the supervised setting is often expressed as the following stochastic optimization problem,

$$\min_{W} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \mathcal{L}(W;(x,y)) \tag{1}$$

where $W = W_1 \times \cdots \times W_l$ denotes the Cartesian product of the weight matrices of the network with l layers that we seek to learn from the data (x, y) sampled from the underlying distribution $\mathcal{D}$. Here, x can be thought of the "features" (or predictor variables) of the data and y denotes the "labels" (or the response variable). The variable W parameterizes the function that predicts the labels given the features whose accuracy is measured using the loss function $\mathcal{L}$. For simplicity, the specification above is intentionally agnostic of the activation function we use between the layers and the specific network architecture. Most common instantiations of the above task are non-convex but results in the last 5 years show that good minimizers can be found via SGD and its variants. Recent results have also explored the interplay between the overparameterization of the network, its degrees of freedom and issues related to global optimality (Soudry and Carmon 2016).

Regularizers. Independent of the architecture we choose to deploy for a given task, one may often want to impose additional constraints or regularizers, pertinent to the application domain of interest. In fact, the use of task specific constraints to improve the behavioral performance of *neural networks*, both from a computational and statistical perspective, has a long history dating back at least to the 1980s (Platt and Barr 1988; Zhang and Constantinides 1992). These ideas are being revisited (Rudd, Di Muro, and Ferrari 2014) motivated by generalization, convergence or simply as a strategy for compression (Cheng and others 2017). However, using constraints on the types of architectures that are common in modern AI problems is still being actively researched by various groups. For example, (Mikolov and others 2014) demonstrated that training Recurrent Networks can be accelerated by constraining a part of the recurrent matrix to be close to identity. Sparsity and low-rank encouraging constraints have shown promise in a number of settings (Tai and others 2015). In an interesting paper, (Pathak, Krahenbuhl, and Darrell 2015) showed that linear constraints on the output layer improves the accuracy on a semantic image segmentation task. (Márquez-Neila, Salzmann, and Fua 2017) showed that hard constraints on the output layer yield competitive results on the pose estimation task and (Oktay and others 2017) used anatomical constraints for cardiac image analysis. This suggests that while there are some results demonstrating the value of *specific* constraints for *specific* problems, the development is still in a nascent stage. It is, therefore, not surprising that the existing software libraries for deep learning (DL) offer little to no support for hard constraints. For example, Keras only offers support for simple bound constraints.

Optimization Schemes. Let us momentarily set aside the issue of constraints and discuss the choice of the optimization schemes that are in use today. There is little doubt that SGD algorithms dominate the landscape of DL problems in AI. Instead of evaluating the loss and the gradient over the full training set, SGD computes the gradient of the parameters using a few training examples. It mitigates the cost of running back propagation over the full dataset and comes with various guarantees as well (Hardt, Recht, and Singer 2016). The reader will notice that part of the reason that constraints have not been intensively explored may have to do with the interplay between constraints and the SGD algorithm (Márquez-Neila, Salzmann, and Fua 2017). While some regularizers and "local" constraints are easily handled within SGD, some others require a great deal of care and can adversely affect convergence and practical runtime (Bengio 2012). There are also a broad range of constraints where SGD is *unlikely to work well based on theoretical results known today* — and it remains an open question in optimization (Johnson and Zhang 2013). We note that algorithms other than the standard SGD have remained a constant focus of research in the community since they offer many theoretical advantages that can also be easily translated to practice (Dauphin, de Vries, and Bengio 2015). These include adaptive sub-gradient methods such as Adagrad, the RMSprop algorithm, and various adaptive schemes for choosing learning rate including momentum based methods (Goodfellow, Bengio, and Courville 2016). However, notice that these methods only impose constraints in a "local" fashion since the computational cost of imposing global constraints using SGD-based methods becomes extremely high (Pathak, Krahenbuhl, and Darrell 2015).

Do we need to impose constraints?

The question of why constraints are needed for statistical learning models in vision and machine learning can be restated in terms of the need for regularization while learning models. Recall that regularization schemes in one form or another go nearly as far back as the study of fitting models to observations of data (Wahba 1990). Broadly speaking, such schemes can be divided into two related categories: algebraic and statistical. The **first** category may refer to problems that are otherwise not possible or difficult to solve, also known as ill-posed problems (Tikhonov, Goncharsky, and Bloch 1987). For example, without introducing some additional piece of information, it is not possible to solve a linear system of equations $Ax = b$ in which the number of observations (rows of A) is less than the number of degrees of freedom (columns of A). In the **second** category, one may use regularization as a way of "explaining" data using simple hypotheses rather than complex ones, for example, the minimum description length principle (Rissanen 1985). The rationale is that, complex hypotheses are less likely to be accurate on the unobserved samples since we need more data to train complex models. Recent developments on the theoretical side of DL showed that imposing simple but *global* constraints on the parameter space is an effective way of analyzing the sample complexity and generalization error (Neyshabur, Tomioka, and Srebro 2015). Hence we seek to solve,

$$\min_{W} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \mathcal{L}(W;(x,y)) + \mu R(W) \tag{2}$$

where $R(\cdot)$ is a suitable regularization function for a fixed $\mu > 0$. We usually assume that $R(\cdot)$ is simple, in the sense that the gradient can be computed efficiently. Using the Lagrangian interpretation, Problem (2) is the same as the following constrained formulation,

$$\min_{W} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \mathcal{L}(W;(x,y)) \text{ s.t. } R(W) \leq \lambda \tag{3}$$

where $\lambda > 0$. Note that when the loss function $\mathcal{L}$ is convex, both the above problems are equivalent in the sense that given $\mu > 0$ in (2), there exists a $\lambda > 0$ in (3) such that the *optimal solutions to both the problems coincide* (see Sec 1.2 in (Bach and others 2012)). In practice, both λ and μ are chosen by standard procedures such as cross validation.

Finding Pareto Optimal Solutions: On the other hand, when the loss function is nonconvex as is typically the case in DL, formulation (3) is more powerful than (2). Let us see why.

For a fixed $\lambda > 0$, there might be solutions W_λ^* of (3) for which there exists *no* $\mu > 0$ such that $W_\lambda^* = W_\mu^*$ whereas any solution of problem (2) can be obtained for some μ in (3) (Section 4.7 in (Boyd and Vandenberghe 2004)). It turns out that it is easier to understand this phenomenon through Multiobjective Optimization (MO). In MO, care has to be taken to even define the notion of optimality of feasible points (let alone computing them efficiently). Among various notions of optimality, we will now argue that Pareto optimality is the most suited for our goal.

Recall that our goal is to find W's that achieve low training error and are at the same time "simple" (as measured by R). In this context, a Pareto optimal solution is a point W such that none of $\mathcal{L}(W)$ or $R(W)$ in (3) or (2) can be made better without making the other worse, thus capturing the essence of overfitting effectively. In practice, there are many algorithms to find Pareto optimal solutions and this is where problem (3) dominates (2). Specifically, formulation (2) falls under the category of "scalarization" technique whereas (3) is ϵ-constrained technique. It is well known that when the problem is nonconvex, ϵ-constrained technique yields pareto optimal solutions whereas scalarization technique does not (Boyd and Vandenberghe 2004)!

Finally, we should note that even when the problems (2) and (3) are equivalent, in practice, algorithms that are used to solve them can be very different.

Our contributions: We show that many interesting global constraints of interest in AI can be explicitly imposed using a classical technique, Conditional Gradient (CG) that has not been deployed much at all in deep learning. We analyze the theoretical aspects of this proposal in detail. On the application side, specifically, we explore and analyze the performance of our CG algorithm with a specific focus on training deep models on the constrained formulation shown in (3). Progressively, we go from cases where there is no (or negligible) loss of both accuracy and training time to scenarios where this procedure shines and offers significant benefits in performance, runtime and generalization. Our experiments indicate that: (i) **with less than** 50% **#-parameters**, we improve ResNet accuracy by 25% (from 8% to 6% test error), and (ii) GANs can be trained in nearly a **third of the computational time** achieving the same or better qualitative performance on an image inpainting task.

First Order Methods: Two Representatives

To setup the stage for our development we first discuss the two broad strategies that are used to solve problems of the form shown in (3). **First**, a natural extension of gradient descent (GD) also known as Projected GD (PGD) may be used. Intuitively, we take a gradient step and then compute the point that is closest to the feasible set defined by the regularization function. Hence, at each iteration PGD requires the solution of the following optimization problem or the so-called Projection operator,

$$W_{t+1}^{PGD} \leftarrow \arg \min_{W:R(W)\leq\lambda} \frac{1}{2}\|W-(W_t-\eta g_t)\|_F^2 \quad (4)$$

where $\|W\|_F$ is the Frobenius norm of W, g_t is (an estimate of) the gradient of $\mathcal{L}$ at W_t and η is the step size. In practice, we compute g_t by using only a few training samples (or minibatch) and running backpropagation. Note that the objective $\mathbb{E}_{(x,y)\sim\mathcal{D}}\mathcal{L}(W)$ is smooth in W for any probability distribution $\mathcal{D}$ with a density function and is commonly referred to as *stochastic smoothing*. Hence, for our descriptions, we will assume that the derivative is well defined. Furthermore, when there are no constraints, (4) is simply the standard SGD method that requires optimizing a quadratic function on the feasible set. So, the main bottleneck in explicitly imposing constraints with PGD is the complexity of solving (4). Even though many $R(\cdot)$ do admit an efficient procedure in theory, using them for applications in training deep models has been a challenge since they may be complicated or not easily amenable to a GPU implementation (Taylor and others 2016; Frerix and others 2017).

So, a natural question to ask is whether there are methods that are faster in the following sense: can we solve simpler problems at each iteration and also explicitly impose the constraints effectively? An assertive answer is provided by a scheme that falls under the **second** general category of first order methods: the Conditional Gradient (CG) algorithm (Reddi and others 2016). Recall that CG methods solve the following *linear minimization problem at each iteration* instead of a quadratic one

$$s_t \in \arg\min_W g_t^T W \text{ s.t. } R(W)\leq\lambda \quad (5)$$

and update $W_{t+1}^{CG} \leftarrow \eta W_t + (1-\eta)s_t$. While both PGD and CG guarantee convergence with mild conditions on η, it may be the case (as we will see shortly) that problems of the form (5) can be *much simpler* than the form in (4) and hence highly suitable for training deep learning models. An additional bonus is that CG algorithms also offer nice space complexity guarantees that are also attainable in practice, making it a very promising choice for explicitly *constrained* training of deep models.

Remark 1. Note that in order for CG algorithm (5) to be well defined, we need the feasible set to be bounded whereas this is not required for PGD (4).

To that end, we will see how the CG algorithm behaves for the regularization constraints that are commonly used.

Remark 2. Note that although the loss function $\mathbb{E}\mathcal{L}(W)$ is nonconvex, the constraints that we need for almost all applications are convex, hence, all our algorithms are guaranteed to converge by design (Lacoste-Julien 2016; Reddi and others 2016) to a stationary point.

Categorizing "Generic" Constraints for CG

We now describe how a broad basket of "generic" constraints broadly used in literature, can be arranged into a hierarchy — ranging from cases where a CG scheme is perfect and expected to yield wide-ranging improvements to situations where the performance is only satisfactory and additional technical development is warranted. For example, the ℓ_1-norm is often used to induce sparsity. The nuclear norm (sum of singular values) is used to induce a low rank regularization, often for compression and/or speed-up reasons.

So, how do we know which constraints when imposed using CG are likely to work well? In order to analyze the qualitative nature of constraints suitable for CG algorithm, we categorize the constraints into three categories based on how the updates will, computationally, and learning-wise, compare to a SGD update.

Category 1 constraints are excellent

We categorize constraints as Category 1 if both the SGD and CG updates take a similar form algebraically. The reason we call this category "excellent" is because it is easy to transfer the empirical knowledge that we obtained in the unconstrained setting, specifically, learning and dropout rates to the regime where we want to explicitly impose these additional constraints. Here, we see that we get quantifiable improvements in terms of both computation and learning.

Two types of generic constraints fall into this category: **1)** the Frobenius norm and **2)** the Nuclear norm (Ruder 2017). We will now see how we can solve (4) and (5) by comparing and contrasting them.

Frobenius Norm. When $R(\cdot)$ is the Frobenius norm, it is easy to see that (4) corresponds to the following,

$$W_{t+1}^{PGD} = \begin{cases} W_t-\eta g_t & \text{if } \|W_t-\eta g_t\|_F \leq \lambda \\ \lambda \cdot \frac{W_t-\eta g_t}{\|W_t-\eta g_t\|_F} & \text{otherwise,} \end{cases} \quad (6)$$

and (5) corresponds to $s_t = -\lambda \frac{g_t}{\|g_t\|_F}$ which implies that,

$$W_{t+1}^{CG} = W_t - (1-\eta)\left(W_t + \lambda\frac{g_t}{\|g_t\|_F}\right). \quad (7)$$

It is easy to see that both the update rules essentially take the same amount of calculation which can be easily done while performing a backpropagation step. So, the actual change in any existing implementation will be minimal but CG will automatically offer an important advantage, notably **scale invariance**, which several recent papers have found to be advantageous (Lacoste-Julien and Jaggi 2015).

Nuclear norm. On the other hand, when $R(\cdot)$ is the nuclear norm, the situation where we use CG (versus not) is quite different. All known projection (or proximal) algorithms require computing at each iteration the **full singular value decomposition** of W, which in the case of deep learning methods becomes restrictive (Recht, Fazel, and Parrilo 2010). In contrast, CG only requires computing the top-1 singular vector of W which can be done easily and efficiently on a GPU via the power method (Jaggi 2013). Hence in this case, if the number of edges in the network is $|E|$ we get a *near-quadratic speed up*, i.e., from $O(|E|^3)$ for PGD to $O(|E|\log|E|)$ making it practically implementable (Golub and Van Loan 2012) in the very large scale settings (Yu, Zhang, and Schuurmans 2017). Furthermore, it is interesting to observe that the rank of W after running T iterations of CG is at most T which implies that we need to only store $2T$ vectors instead of the whole matrix W making it a viable solution for deployment on devices with memory constraints (Howard and others 2017). The main takeaway is that, since projections are computationally expensive, **projected SGD is not a viable option in practice**.

Category 2 constraints are potentially good

As we saw earlier, CG algorithms are always *at least* as efficient as the PGD updates: in general, any constraint that can be imposed using the PGD algorithm can also be imposed by CG algorithm, if not faster. Hence, generic constraints are defined to be Category 2 constraints for CG if the empirical knowledge cannot be easily transferred from PGD. Two classical norms that fall into this category: $\|W\|_1$ and $\|W\|_\infty$. For example, PGD on the ℓ_1 ball can be done in linear time (see (Duchi and others 2008)) and for $\|W\|_\infty$ using gradient clipping (Boyd and Vandenberghe 2004). So, let us evaluate the CG step (5) for the constraint $\|W\|_1 \leq \lambda$ which corresponds to,

$$s_t^j = \begin{cases} -\lambda & \text{if } j^* = \arg\max_j \left|g_t^j\right| \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

That is, we assign $-\lambda$ to the coordinate of the gradient g_t that has the maximum magnitude in the gradient matrix which corresponds to a *deterministic* dropout regularization in which at each iteration we only update *one* edge of the network. While this might not be necessarily bad, it is now common knowledge that a high dropout rate (i.e., updating very few weights at each iteration) leads to underfitting or in other words, the network tends to need a longer training time (Srivastava and others 2014). Similarly, the update step (5) for CG algorithm with $\|W\|_\infty$ takes the following form,

$$s_t^j = \begin{cases} +\lambda & \text{if } g_t^j < 0 \\ -\lambda & \text{otherwise.} \end{cases} \tag{9}$$

In this case, the CG update uses only the sign of the gradient and does not use the magnitude at all. In both cases, one issue is that information about the gradients is not used by the standard form of the algorithm making it not so efficient for practical purposes. Interestingly, even though the update rules in (8) and (9) use extreme ways of using the gradient information, we can, in fact, use a group norm type penalty to model the trade-off. Recent work shows that there are very efficient procedures to solve the corresponding CG updates as well (5).

Remark 3. The main takeaway from the discussion is that Category 2 constraints surprisingly unifies many regularization techniques that are traditionally used in DL in a more methodical way.

Category 3 constraints need more work

There is one class of regularization norms that do not nicely fall in either of the above categories, but is used in several problems in vision: the Total Variation (TV) norm. TV norm is widely used in denoising algorithms to promote smoothness of the estimated sharp image (Chambolle and Lions 1997). The TV norm on an image I is defined as a certain type of norm of its discrete gradient field $(\nabla_i I(\cdot), \nabla_j I(\cdot))$ i.e.,

$$\|I\|_{TV}^p := ((\|\nabla_i I\|_p + \|\nabla_j I\|_p)^p \,. \tag{10}$$

Note that for $p \in \{1, 2\}$, this corresponds to the classical anisotropic and isotropic TV norm respectively. Motivated by the above idea, we can now define the TV norm of a Feed Forward Deep Network. TV norm, as the name suggests, captures the notion of balanced networks, shown to make the network more stable (Neyshabur, Salakhutdinov, and Srebro 2015). Let A be the incidence matrix of the network: the rows of A are indexed by the nodes and the columns are indexed by the (directed) edges such that each column contains exactly two nonzero entries: a $+1, -1$ in the rows corresponding to the starting node u and ending node v respectively. Let us also consider the weight matrix of the network as a vector (for simplicity) indexed in the same order as the columns of A. Then, the TV norm of the deep neural network is,

$$\|W\|_{TV} := \|AW\|_p. \tag{11}$$

It turns out that when $R(W) = \|W\|_{TV}$, PGD is **not** trivial to solve and requires special schemes (Fadili and Peyré 2011) with runtime complexity of $O(n^4)$ where n is the number of nodes — impractical for most deep learning applications. In contrast, CG iterations only require a special form of maximum flow computation which can be done efficiently (Harchaoui, Juditsky, and Nemirovski 2015).

Lemma 4. *An ϵ-approximate CG step* (5) *can be computed in $O(1/\epsilon)$ time (independent of dimensions of A).*

Proof. (Sketch) We show that the problem is equivalent to solving the dual of a specific linear program which can be efficiently done using (Johnson and Zhang 2013). □

Remark 5. The above discussion suggests that conceptually, Category 3 constraints can be incorporated and will immensely benefit from CG methods. However, unlike Category 1-2 constraints, it requires specialized implementations to solve subproblems from (11) which are not currently available in popular libraries. So, additional work is needed before broad utilization may be possible.

Path Norm Constraints in Deep Learning

So far, we only covered constraints that were already in use in vision/machine learning and recently, some attempts (Márquez-Neila, Salzmann, and Fua 2017) were made to utilize them in deep networks. Now, we review a new notion of regularization, introduced recently, that has its roots primarily in deep learning (Neyshabur, Salakhutdinov, and Srebro 2015). We will first see the definition and explain some of the properties that this type of constraint captures.

Algorithm 1 Path-CG iterations

```
Pick a starting point $W_0 : \|W_0\|_\pi \leq \lambda$ and $\eta \in (0,1)$.
for $t = 0, 1, 2, \cdots, T$ iterations do
    for $j = 0, 1, 2, \cdots, l$ layers do
        $g \leftarrow$ gradient of edges from $j-1$ to $j$ layer.
        Compute $\gamma_e \; \forall \; e$ from $j-1$ to $j$ layer (eq (13))
        Set $s_t^j \leftarrow \arg\min_W g^T W$ s.t. $\|\Gamma W\|_2 \leq \lambda$ (eq(14))
        Update $W_{t+1}^j \leftarrow \eta W_t^j + (1-\eta) s_t^j$
    end for
end for
```

Definition 6. (Neyshabur, Salakhutdinov, and Srebro 2015) The ℓ_2-path regularizer is defined as :

$$\|W\|_\pi^2 = \sum_{v_{in}[i] \xrightarrow{e_1} v_1 \xrightarrow{e_2} \cdots v_{out}[j]} \left| \prod_{k=1}^{l} W_{e_k} \right|^2 . \tag{12}$$

Here π denotes the set of paths, v_{in} corresponds to a node in the input layer, e_i corresponds to an edge between a node $(i-1)$-th layer and i-th layer that lies in the path between v_{in} and v_{out} in the output layer. Therefore, the path norm measures *norm of all possible paths π in the network up to the output layer*.

Why do we need path norm? One of the basic properties of ReLu (Rectified Linear Units) is that it is *scaling invariant* in the following way: multiplying the weights of incoming edges to a node i by a positive constant and dividing the outgoing edges from the same node i does not change $\mathcal{L}$ for any (x, y). Hence, an update scheme that is *scaling invariant* will significantly increase the training speed. Furthermore, the authors in (Neyshabur, Salakhutdinov, and Srebro 2015) showed how path regularization converges to optimal solutions that can generalize better compared to the usual SGD updates — so apart from computational benefits, there are clear statistical generalization advantages too.

How do we incorporate the path norm constraint? Recall from Remark 1 that the feasible set has to be bounded, so that the step (5) is well defined. Unfortunately, this is not the case with the path norm. To see this, consider a simple line graph with weights W_1 and W_2. In this case, there is only one path and the path norm constraint is $W_1^2 W_2^2 \leq 1$ which is clearly unbounded. Further, we are not aware of an efficient procedure to compute the projection for higher dimensions since there is no known efficient separation oracle. Interestingly, we take advantage of the fact that if we fix W_1, then the feasible set is bounded. This intuition can be generalized, that is, we can update one layer at a time which we will describe now precisely.

Path-CG Algorithm: In order to simplify the presentation, we will assume that there are no biases noting that the procedure can be easily extended to the case when we have individual bias for every node. Let us fix a layer j and the vectorized weight matrix of that layer be W that we want to update and as usual, g corresponds to the gradient. Let the number of nodes in the $(j-1)$ and j-th layers be n_1 and n_2 respectively. For each edge between these two layers we will compute the scaling factors γ_e defined as,

$$\gamma_e = \sum_{v_{in}[i] \cdots \xrightarrow{e} \cdots v_{out}[j]} \left| \prod_{e_k \neq e} W_{e_k} \right|^2 . \tag{13}$$

Intuitively, γ_e computes the norm of all paths that pass through the edge e excluding the weight of e. This can be efficiently done using Dynamic Programming in time $O(l)$ where l is the number of layers. Consequently, the computation of path norm also satisfies the same runtime, see (Neyshabur, Salakhutdinov, and Srebro 2015) for more details. Now, observe that the path norm constraint when all of the other layers are fixed reduces to solving the following problem,

$$\min_W g^T W \text{ s.t. } \|\Gamma W\|_2 \leq \lambda \tag{14}$$

where Γ is a diagonal matrix with $\Gamma_{e,e} = \gamma_e$, see (13). Hence, we can see that the problem again reduces to a simple rescaling and then normalization as seen for the Frobenius norm in (7) and repeat for each layer.

Remark 7. The starting point W_0 such that $\|W_0\|_\pi \leq \lambda$ can be chosen simply by randomly assigning the weights from the Normal Distribution with mean 0.

Complexity of Path-CG 1: From the above discussion, our full algorithm is given in Algorithm 1. The main computational complexity in Path-CG comes from computing the matrix Γ for each layer, but as we described earlier, this can be done *by* backpropagation – $O(1)$ flops per example. Hence, the complexity of Path-CG for running T iterations is essentially $O(lBT)$ where B is a size of the mini-batch.

Scale invariance of Path-CG 1: Note that CG algorithms satisfy a much general property called as Affine Invariance (Jaggi 2013), which implies that it is also scale invariant. Scale invariance makes our algorithm more efficient (in wall clock time) since it avoids exploring parameters that correspond to the same prediction function.

Experimental Evaluation

We present experimental results on three different case studies to support our basic premise and theoretical findings in the earlier sections: constraints can be easily handled with our CG algorithm in the context of Deep Learning while preserving the empirical performance of the models. **The first set** of experiments is designed to show how simple/generic constraints can be easily incorporated in existing deep learning models to get both faster training times and better accuracy while reducing the #-layers using the ResNet architecture. **The second set** of experiments is to evaluate our Path-CG algorithm. The goal is to show that Path-CG is much more stable than the Path-SGD algorithm in (Neyshabur, Salakhutdinov, and Srebro 2015), implying lower generalization error of the model. **In the third set** of experiments we show that GANs (Generative Adversarial Networks) can be trained faster using the CG algorithm and that the training tends to be stable. To validate this, we test the performance of the GAN on image inpainting. Since CG algorithm maintains a solution that is a convex combination of all previous

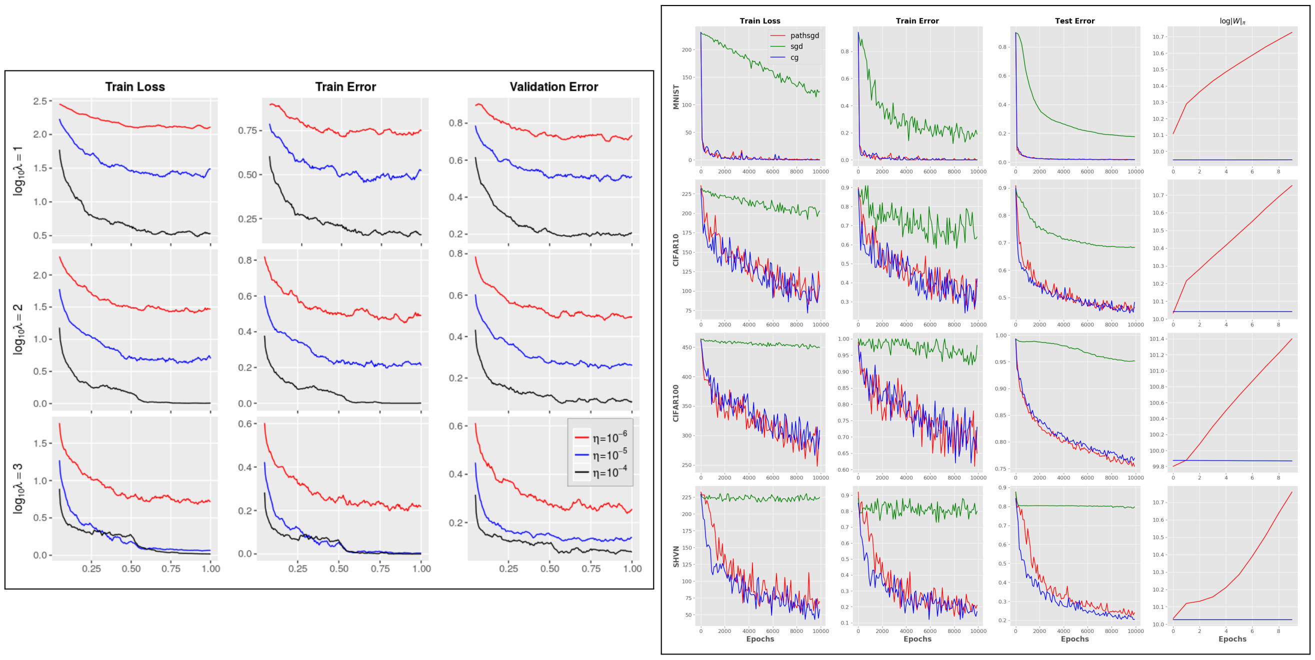


Figure 1: (Left) Performance of CG on ResNet-32 on CIFAR10 dataset (x-axis denotes the fraction of T): as λ increases, the training error, loss value and test error all start to decrease simultaneously. (Right) Performance of Path-CG vs SGD on a 2-layer fully connected network on four datasets (x-axis denotes the #iterations). Observe that across all datasets, Path-CG is much faster than SGD (first three columns). Last column shows that SGD is not stable with respect to the path norm.

iterates, hence to decrease the effect of random initialization, the training scheme consists of two phases: (i) burn-in phase in which the CG algorithm is run with a constant stepsize; (ii) decay phase in which the stepsize is decaying according to $1/t$. This makes sure that the effect of randomness from the initialization is diminished. We use 1 epoch for the burn-in phase, hence we can conclude that the algorithm is guaranteed to converge to a stationary point (Lacoste-Julien 2016).

Improve ResNets using Conditional Gradients

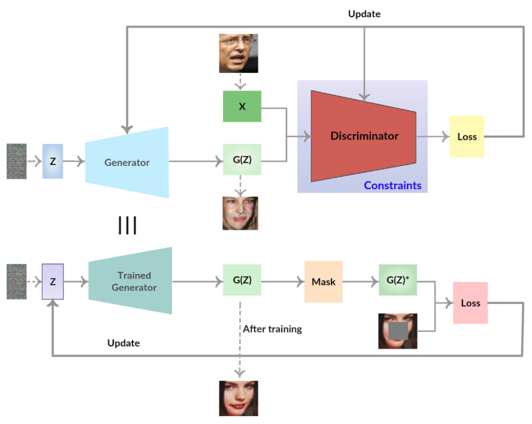
We start with the problem of image classification, detection and localization. For these tasks, one of the best performing architectures are variants of the Deep Residual Networks (ResNet) (He and others 2016). For our purposes, to analyze the performance of CG algorithm, we used the shallower variant of ResNet, namely ResNet-32 (32 hidden layers) architecture and trained on the CIFAR10 (Krizhevsky, Sutskever, and Hinton 2012) dataset. ResNet-32 consists of 5 residual blocks and 2 fully connected, one each at the input and output layers. Each residual block consists of 2 convolution, ReLu (Rectified Linear units), and batch normalization layers, see (He and others 2016) for more details. CIFAR10 dataset contains 60000 color images of size 32×32 with 10 different categories/labels. Hence, the network contains approximately $0.46M$ parameters.

To make the discussion clear, we present results for the case where the total Frobenius norm of the network parameters is constrained to be less than λ and trained using the CG algorithm. To see the effect of the parameters λ and step sizes η on the model, we ran 80000 iterations, see Figure 1. The plots essentially show that if λ is chosen reasonably big, then the accuracy of CG is very close to the accuracy of ResNet-164 (5.46% top-1 test error, see (He and others 2016)) that has **many more parameters** (approximately 5 times!). In practice, since λ is a constraint parameter, we can initially choose λ to be small and gradually increase it, thus avoiding complicated grid search procedures.Thus, figure 1 shows that CG can be used to improve the performance of *existing architectures* by appropriately choosing constraints (see supplement for more experiments).

Takeaway: *CG offers fewer parameters and higher accuracy on a standard network with no additional change.*

Path-CG vs Path-SGD: Which is better?

In this case study, the goal is to compare Path-CG with the Path-SGD algorithm (Neyshabur, Salakhutdinov, and Srebro 2015) in terms of both accuracy and stability of the algorithm. To that end, we considered image classification problem with a path norm constraint on the network: $\|W_t\|_p \leq \lambda$ for varying λ as before. We train a simple feed-forward network which consists of 2 fully-connected hidden layers with 4000 units each, followed by the output layer with 10 nodes. We used ReLu nonlinearity as the activation function and cross entropy as the loss, see (Neyshabur, Salakhutdinov, and Srebro 2015) for more details.
We performed experiments on 4 standard datasets for image classification: MNIST, CIFAR (10,100) (Krizhevsky, Sutskever, and Hinton 2012) and finally color images of house numbers from SVHN dataset (Netzer and others).



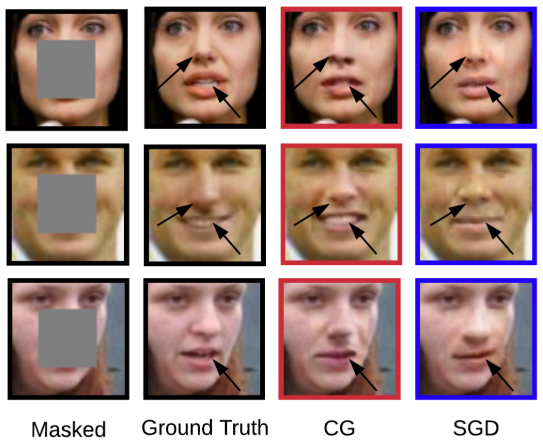


Figure 2: **Left:** Illustrates the task of image inpainting overall pipeline. **Right:** CG-trained DC-GAN performs as good as (or better than) SGD-based DC-GAN but with 50% epochs.

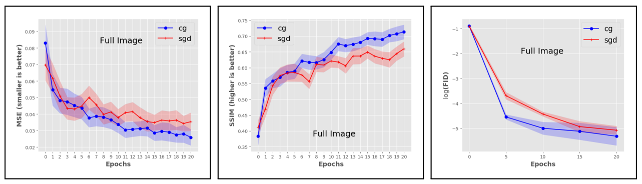


Figure 3: From left: show MSE/SSIM/FID on the **full** image.

Figure 1 (right) shows the result for $\lambda = 10^{-5}$ (after tuning), it can achieve the same accuracy as that of Path-SGD.

Path-CG has one main advantage over Path-SGD: Our results in the supplement show that Path-CG is more stable while the path norm of Path-SGD algorithm increases rapidly. This shows that Path-SGD *does not* effectively regularize the path norm whereas Path-CG keeps the path norm less than λ as expected.

Takeaway: *All statistical benefits of path norm are possible via CG while being computationally stable.*

Image Inpainting using Conditional Gradients

Finally, we illustrate the ability of our CG framework on an exciting and recent application of image inpainting using Generative Adversarial Networks (GANs). We now briefly explain the overall experimental setup. GANs using game theoretic notions can be defined as a system of 2 neural networks called Generator and the Discriminator competing with each other in a zero-sum game (Arora and others 2017).

Image inpainting/completion can be performed using the following two steps (Amos): (i) Train a standard GAN as a normal image generation task, and (ii) use the trained generator and then tune the noise that gives the best output. Hence, our hypothesis is that if the generator is trained well, then the follow-up task of image inpainting benefits automatically.

Train DC-GAN faster for better image inpainting: We used the state of the art DC-GAN architecture in our experiments and we impose a Frobenius norm constraint on the parameters but *only* on the Discriminator to avoid mode collapse issues and trained using the CG algorithm. In order to verify the performance of the CG algorithm, we used 2 standard face image datasets from CelebA and LWF and conducted two experiments: trained on the CelebA dataset with LFW being the test dataset and vice-versa. We found that the generator generates very high quality images after being trained with LFW images in comparison to the original DC-GAN *in just* 10 *epochs* (reducing the computational **cost by** 50%). Quantitatively, we provide numerical evidence in Figure 3 with 2 intrinsic metrics viz., **Structural Similarity (SSim), Mean Squared Error (MSE)** and 1 extrinsic metric, **Frechet Inception Distance (FID)**. All the three metrics are standard in GAN literature. We calculated the intrinsic metrics *after* the image completion phase. We can see that on *all the three metrics*, CG outperforms SGD clearly.

Takeaway: *GANs can be trained faster with no change in accuracy.*

Conclusions

The main emphasis of our work is to provide evidence supporting three distinct but related threads: (i) global constraints are relevant in the context of training deep models in vision and machine learning; (ii) the lack of support for global constraints in existing libraries like Keras and Tensorflow (Abadi and others 2016) may be because of the complex interplay between constraints and SGD which we have shown can be side-stepped, to a great extent, using CG; and (iii) constraints can be easily incorporated with negligible to small changes to existing implementations. We provide empirical results on *three* different case studies to support our claims, and conjecture that a broad variety of other problems will immediately benefit by viewing them through the lens of CG algorithms. Our analysis and experiments suggest concrete ways in which one may realize improvements, in both generalization and runtime, by substituting in CG schemes in deep learning models. Tensorflow code for all our experiments will be made available in Github.

Acknowledgements This work is supported by NSF CAREER RI 1252725 (VS), and UW CPCP (U54 AI117924).

References

Abadi, M., et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*.

Amos, B. Image Completion with Deep Learning in TensorFlow. http://bamos.github.io/2016/08/09/deep-completion. Accessed: [09/05/2018].

Arora, S., et al. 2017. Generalization and equilibrium in generative adversarial nets (gans). In *ICML*.

Bach, F., et al. 2012. Optimization with sparsity-inducing penalties. *Foundations and Trends® in Machine Learning*.

Bengio, Y. 2012. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*.

Boyd, S., and Vandenberghe, L. 2004. *Convex optimization*.

Chambolle, A., and Lions, P.-L. 1997. Image recovery via total variation minimization and related problems. *Numerische Mathematik*.

Cheng, Y., et al. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv:1710.09282*.

Dauphin, Y.; de Vries, H.; and Bengio, Y. 2015. Equilibrated adaptive learning rates for non-convex optimization. In *NIPS*.

Duchi, J., et al. 2008. Efficient projections onto the l 1-ball for learning in high dimensions. In *ICML*.

Fadili, J. M., and Peyré, G. 2011. Total variation projection with first order schemes. *IEEE Transactions on Image Processing*.

Frerix, T., et al. 2017. Proximal backpropagation. *arXiv:1706.04638*.

Golub, G. H., and Van Loan, C. F. 2012. *Matrix computations*.

Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep Learning*.

Harchaoui, Z.; Juditsky, A.; and Nemirovski, A. 2015. Conditional gradient algorithms for norm-regularized smooth convex optimization. *Mathematical Programming*.

Hardt, M.; Recht, B.; and Singer, Y. 2016. Train faster, generalize better: Stability of stochastic gradient descent. In *ICML*.

He, K., et al. 2016. Deep residual learning for image recognition. In *CVPR*.

Howard, A. G., et al. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*.

Jaggi, M. 2013. Revisiting frank-wolfe: projection-free sparse convex optimization. In *ICML*.

Johnson, R., and Zhang, T. 2013. Accelerating stochastic gradient descent using predictive variance reduction. In *NIPS*.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS*.

Lacoste-Julien, S., and Jaggi, M. 2015. On the global linear convergence of frank-wolfe optimization variants. In *NIPS*.

Lacoste-Julien, S. 2016. Convergence rate of frank-wolfe for non-convex objectives. *arXiv:1607.00345*.

Márquez-Neila, P.; Salzmann, M.; and Fua, P. 2017. Imposing hard constraints on deep networks: Promises and limitations. *arXiv:1706.02025*.

Mikolov, T., et al. 2014. Learning longer memory in recurrent neural networks. *arXiv:1412.7753*.

Netzer, Y., et al. Reading digits in natural images with unsupervised feature learning.

Neyshabur, B.; Salakhutdinov, R. R.; and Srebro, N. 2015. Path-sgd: Path-normalized optimization in deep neural networks. In *NIPS*.

Neyshabur, B.; Tomioka, R.; and Srebro, N. 2015. Norm-based capacity control in neural networks. In *COLT*.

Oktay, O., et al. 2017. Anatomically constrained neural networks (acnn): Application to cardiac image enhancement and segmentation. *arXiv:1705.08302*.

Pathak, D.; Krahenbuhl, P.; and Darrell, T. 2015. Constrained convolutional neural networks for weakly supervised segmentation. In *ICCV*.

Platt, J. C., and Barr, A. H. 1988. Constrained differential optimization. In *NIPS*.

Recht, B.; Fazel, M.; and Parrilo, P. A. 2010. Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization. *SIAM review*.

Reddi, S. J., et al. 2016. Stochastic frank-wolfe methods for nonconvex optimization. In *54th Annual Allerton Conference*.

Rissanen, J. 1985. *Minimum description length principle*.

Rudd, K.; Di Muro, G.; and Ferrari, S. 2014. A constrained backpropagation approach for the adaptive solution of partial differential equations. *IEEE transactions on neural networks and learning systems*.

Ruder, S. 2017. An overview of multi-task learning in deep neural networks. *arXiv:1706.05098*.

Soudry, D., and Carmon, Y. 2016. No bad local minima: Data independent training error guarantees for multilayer neural networks. *arXiv:1605.08361*.

Srivastava, N., et al. 2014. Dropout: a simple way to prevent neural networks from overfitting. *JMLR*.

Tai, C., et al. 2015. Convolutional neural networks with low-rank regularization. *arXiv:1511.06067*.

Taylor, G., et al. 2016. Training neural networks without gradients: A scalable admm approach. In *ICML*.

Tikhonov, A. N.; Goncharsky, A.; and Bloch, M. 1987. *Ill-posed problems in the natural sciences*.

Wahba, G. 1990. *Spline models for observational data*. SIAM.

Yu, Y.; Zhang, X.; and Schuurmans, D. 2017. Generalized conditional gradient for sparse estimation. *The Journal of Machine Learning Research*.

Zhang, S., and Constantinides, A. 1992. Lagrange programming neural networks. *IEEE Transactions on Circuits and Systems II*.