

Geometry of Optimization and Implicit Regularization in Deep Learning

This survey chapter was done as a part of Intel Collaborative Research institute for Computational Intelligence (ICRI-CI) “Why & When Deep Learning works – looking inside Deep Learning” compendium with the generous support of ICRI-CI.

Behnam Neyshabur

BNEYSHABUR@TTIC.EDU

*Toyota Technological Institute at Chicago
Chicago, IL 60637, USA*

Ryota Tomioka

RYOTO@MICROSOFT.COM

*MSR Cambridge
Cambridge, CB1 2FB, UK*

Ruslan Salakhutdinov

RSALAKHU@CS.CMU.EDU

*School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA*

Nathan Srebro

NATI@TTIC.EDU

*Toyota Technological Institute at Chicago
Chicago, IL 60637, USA*

Editor:

Abstract

We argue that the optimization plays a crucial role in generalization of deep learning models through implicit regularization. We do this by demonstrating that generalization ability is not controlled by network size but rather by some other implicit control. We then demonstrate how changing the empirical optimization procedure can improve generalization, even if actual optimization quality is not affected. We do so by studying the geometry of the parameter space of deep networks, and devising an optimization algorithm attuned to this geometry.

Keywords: Deep Learning, Implicit Regularization, Geometry of Optimization, Path-norm, Path-SGD

1. Introduction

Central to any form of learning is an inductive bias that induces some sort of capacity control (i.e. restricts or encourages predictors to be “simple” in some way), which in turn allows for generalization. The success of learning then depends on how well the inductive bias captures reality (i.e. how expressive is the hypothesis class of “simple” predictors) relative to the capacity induced, as well as on the computational complexity of fitting a “simple” predictor to the training data.

Let us consider learning with feed-forward networks from this perspective. If we search for the weights minimizing the training error, we are essentially considering the hypothesis class of predictors representable with different weight vectors, typically for some fixed architecture. Capacity is then controlled by the size (number of weights) of the network¹. Our justification for using such networks is then that many interesting and realistic functions can be represented by not-too-large (and hence bounded capacity) feed-forward networks. Indeed, in many cases we can show how specific architectures can capture desired behaviors. More broadly, any $O(T)$ time computable function can be captured by an $O(T^2)$ sized network, and so the expressive power of such networks is indeed great (Sipser, 2006, Theorem 9.25).

At the same time, we also know that learning even moderately sized networks is computationally intractable—not only is it NP-hard to minimize the empirical error, even with only three hidden units, but it is hard to learn small feed-forward networks using *any* learning method (subject to cryptographic assumptions). That is, even for binary classification using a network with a single hidden layer and a logarithmic (in the input size) number of hidden units, and even if we know the true targets are *exactly* captured by such a small network, there is likely no efficient algorithm that can ensure error better than $1/2$ (Sherstov, 2006; Daniely et al., 2014)—not if the algorithm tries to fit such a network, not even if it tries to fit a much larger network, and in fact no matter how the algorithm represents predictors. And so, merely knowing that some not-too-large architecture is excellent in expressing reality does *not* explain why we are able to learn using it, nor using an even larger network. Why is it then that we succeed in learning using multilayer feed-forward networks? Can we identify a property that makes them possible to learn? An alternative inductive bias?

In section 2, we make our first steps at shedding light on this question by going back to our understanding of network size as the capacity control at play. Our main observation, based on empirical experimentation with single-hidden-layer networks of increasing size (increasing number of hidden units), is that size does *not* behave as a capacity control parameter, and in fact there must be some other, implicit, capacity control at play. We suggest that this hidden capacity control might be the real inductive bias when learning with deep networks.

Revisiting the choice of gradient descent, we recall that optimization is inherently tied to a choice of geometry or measure of distance, norm or divergence. Gradient descent for example is tied to the ℓ_2 norm as it is the steepest descent with respect to ℓ_2 norm in the parameter space, while coordinate descent corresponds to steepest descent with respect to the ℓ_1 norm and exp-gradient (multiplicative weight) updates is tied to an entropic divergence. Moreover, at least when the objective function is convex, convergence behavior is tied to the corresponding norms or potentials. For example, with gradient descent, or SGD, convergence speeds depend on the ℓ_2 norm of the optimum. The norm or divergence can be viewed as a regularizer for the updates. There is therefore also a strong link between regularization for optimization and regularization for learning: optimization may provide implicit regularization in terms of its corresponding geometry, and for ideal optimization performance the optimization geometry should be aligned with inductive bias driving the learning (Srebro et al., 2011).

Is the ℓ_2 geometry on the weights the appropriate geometry for the space of deep networks? Or can we suggest a geometry with more desirable properties that would enable faster optimization and

1. The exact correspondence depends on the activation function—for hard thresholding activation the pseudo-dimension, and hence sample complexity, scales as $O(S \log S)$, where S is the number of weights in the network. With sigmoidal activation it is between $\Omega(S^2)$ and $O(S^4)$ (Anthony and Bartlett, 1999).

perhaps also better implicit regularization? As suggested above, this question is also linked to the choice of an appropriate regularizer for deep networks.

Focusing on networks with RELU activations in this section, we observe that scaling down the incoming edges to a hidden unit and scaling up the outgoing edges by the same factor yields an equivalent network computing the same function. Since predictions are invariant to such rescalings, it is natural to seek a geometry, and corresponding optimization method, that is similarly invariant.

We consider here a geometry inspired by max-norm regularization (regularizing the maximum norm of incoming weights into any unit) which seems to provide a better inductive bias compared to the ℓ_2 norm (weight decay) (Goodfellow et al., 2013; Srivastava et al., 2014). But to achieve rescaling invariance, we use not the max-norm itself, but rather the minimum max-norm over all rescalings of the weights. We discuss how this measure can be expressed as a “path regularizer” and can be computed efficiently.

We therefore suggest a novel optimization method, Path-SGD, that is an approximate steepest descent method with respect to path regularization. Path-SGD is rescaling-invariant and we demonstrate that Path-SGD outperforms gradient descent and AdaGrad for classifications tasks on several benchmark datasets. This again demonstrates the importance of implicit regularization that is introduced by optimization.

This summary paper combines material previously presented by the authors at the 3rd International Conference on Learning Representations (ICLR), the 28th Conference on Learning Theory (COLT) and Advances in Neural Information Processing Systems (NIPS) 28, as well as Intel Collaborative Research Institutes retreats (Neyshabur et al., 2015a,b,c).

Notations A feedforward neural network that computes a function $f : \mathbb{R}^D \rightarrow \mathbb{R}^C$ can be represented by a directed acyclic graph (DAG) $G(V, E)$ with D input nodes $v_{in}[1], \dots, v_{in}[D] \in V$, C output nodes $v_{out}[1], \dots, v_{out}[C] \in V$, weights $w : E \rightarrow \mathbb{R}$ and an activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ that is applied on the internal nodes (hidden units). We denote the function computed by this network as $f_{G,w,\sigma}$. In this paper we focus on RELU (REctified Linear Unit) activation function $\sigma_{\text{RELU}}(x) = \max\{0, x\}$. We refer to the depth d of the network which is the length of the longest directed path in G . For any $0 \leq i \leq d$, we define V_{in}^i to be the set of vertices with longest path of length i to an input unit and V_{out}^i is defined similarly for paths to output units. In layered networks $V_{in}^i = V_{out}^{d-i}$ is the set of hidden units in a hidden layer i .

2. Implicit Regularization

Consider training a feed-forward network by finding the weights minimizing the training error. Specifically, we will consider a network with D real-valued inputs $\mathbf{x} = (x[1], \dots, x[D])$, a single hidden layer with H rectified linear units, and C outputs $y[1], \dots, y[k]$ where the weights are learned by minimizing a (truncated) soft-max cross entropy loss² on n labeled training examples. The total number of weights is then $H(C + D)$.

2. When using soft-max cross-entropy, the loss is never exactly zero for correct predictions with finite margins/confidences. Instead, if the data is separable, in order to minimize the loss the weights need to be scaled up toward infinity and the cross entropy loss goes to zero, and a global minimum is never attained. In order to be able to say that we are actually reaching a zero loss solution, and hence a global minimum, we use a slightly modified soft-max which does not noticeably change the results in practice. This truncated loss returns the same exact value for wrong predictions or correct prediction with confidences less than a threshold but returns zero for correct predictions with large enough margins: Let $\{s_i\}_{i=1}^k$ be the scores for k possible labels and c be the correct labels. Then the soft-max cross-entropy loss can be written as $\ell(s, c) = \ln \sum_i \exp(s_i - s_c)$ but we instead use the differentiable

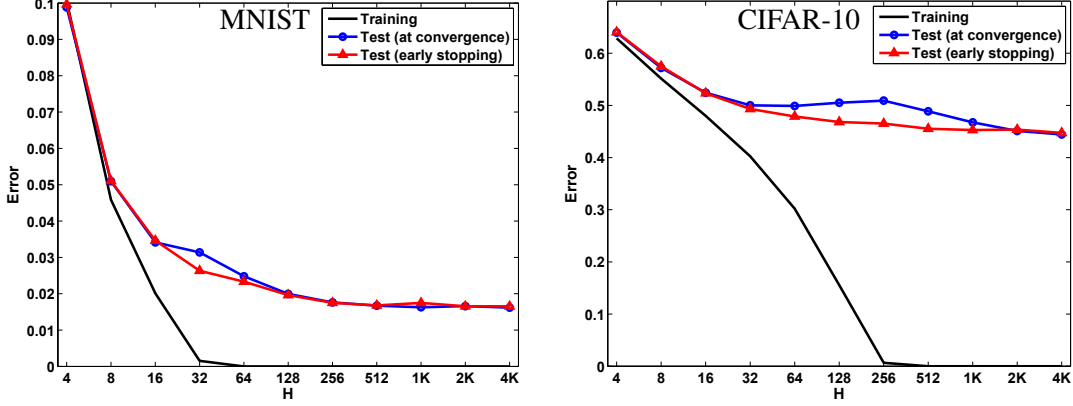


Figure 1: The training error and the test error based on different stopping criteria when 2-layer NNs with different number of hidden units are trained on MNIST and CIFAR-10. Images in both datasets are downsampled to 100 pixels. The size of the training set is 50000 for MNIST and 40000 for CIFAR-10. The early stopping is based on the error on a validation set (separate from the training and test sets) of size 10000. The training was done using stochastic gradient descent with momentum and mini-batches of size 100. The network was initialized with weights generated randomly from the Gaussian distribution. The initial step size and momentum were set to 0.1 and 0.5 respectively. After each epoch, we used the update rule $\mu^{(t+1)} = 0.99\mu^{(t)}$ for the step size and $m^{(t+1)} = \min\{0.9, m^{(t)} + 0.02\}$ for the momentum.

What happens to the training and test errors when we increase the network size H ? The training error will necessarily decrease. The test error might initially decrease as the approximation error is reduced and the network is better able to capture the targets. However, as the size increases further, we lose our capacity control and generalization ability, and should start overfitting. This is the classic approximation-estimation tradeoff behavior.

Consider, however, the results shown in Figure 1, where we trained networks of increasing size on the MNIST and CIFAR-10 datasets. Training was done using stochastic gradient descent with momentum and diminishing step sizes, on the training error and without any explicit regularization. As expected, both training and test error initially decrease. More surprising is that if we increase the size of the network past the size required to achieve zero training error, the test error continues decreasing! This behavior is not at all predicted by, and even contrary to, viewing learning as fitting a hypothesis class controlled by network size. For example for MNIST, 32 units are enough to attain zero training error. When we allow more units, the network is not fitting the training data any better, but the estimation error, and hence the generalization error, should increase with the increase in capacity. However, the test error goes down. In fact, as we add more and more parameters, even beyond the number of training examples, the generalization error does not go up.

loss function $\hat{\ell}(s, c) = \ln \sum_i f(s_i - s_c)$ where $f(x) = \exp(x)$ for $x \geq -11$ and $f(x) = \exp(-11)[x + 13]_+^2/4$ otherwise. Therefore, we only deviate from the soft-max cross-entropy when the margin is more than 11, at which point the effect of this deviation is negligible (we always have $|\ell(s, c) - \hat{\ell}(s, c)| \leq 0.000003k$)—if there are any actual errors the behavior on them would completely dominate correct examples with margin over 11, and if there are no errors we are just capping the amount by which we need to scale up the weights.

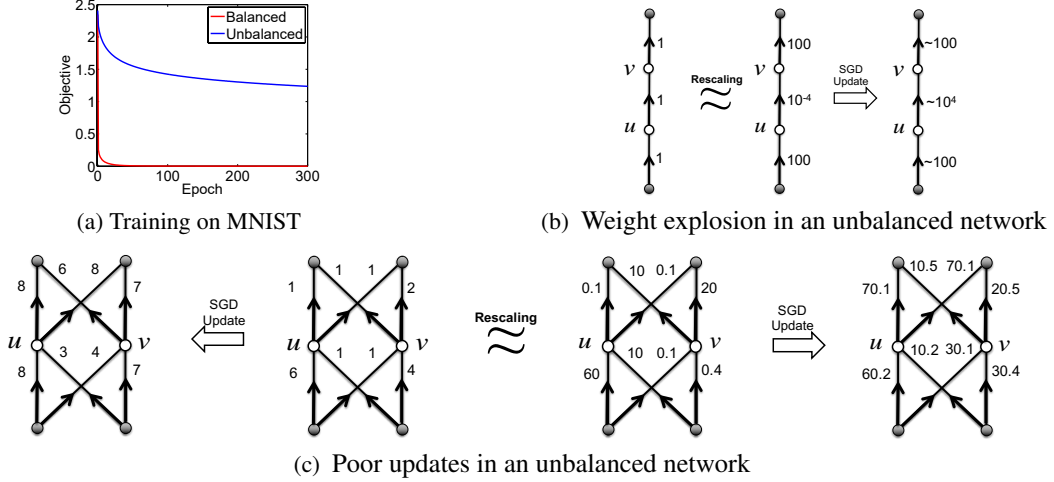


Figure 2: (a): Evolution of the cross-entropy error function when training a feed-forward network on MNIST with two hidden layers, each containing 4000 hidden units. The unbalanced initialization (blue curve) is generated by applying a sequence of rescaling functions on the balanced initializations (red curve). (b): Updates for a simple case where the input is $x = 1$, thresholds are set to zero (constant), the stepsize is 1, and the gradient with respect to output is $\delta = -1$. (c): Updated network for the case where the input is $x = (1, 1)$, thresholds are set to zero (constant), the stepsize is 1, and the gradient with respect to output is $\delta = (-1, -1)$.

What is happening here? A possible explanation is that the optimization is introducing some implicit regularization. That is, we are implicitly trying to find a solution with small “complexity”, for some notion of complexity, perhaps norm. This can explain why we do not overfit even when the number of parameters is huge. Furthermore, increasing the number of units might allow for solutions that actually have lower “complexity”, and thus generalization better. Perhaps an ideal then would be an infinite network controlled only through this hidden complexity.

We want to emphasize that we are not including any explicit regularization, neither as an explicit penalty term nor by modifying optimization through, e.g., drop-outs, weight decay, or with one-pass stochastic methods. We are using a stochastic method, but we are running it to convergence—we achieve zero surrogate loss and zero training error. In fact, we also tried training using batch conjugate gradient descent and observed almost identical behavior. But it seems that even still, we are not getting to some random global minimum—indeed for large networks the vast majority of the many global minima of the training error would horribly overfit. Instead, the optimization is directing us toward a “low complexity” global minimum.

We have argued that the implicit regularization is due to the optimization. It is therefore expected that different optimization methods introduce different implicit regularizations which leads to different generalization properties. In an attempt to find an optimization method with better generalization properties, we recall that the optimization is also tied to a choice of geometry/distance measure in the parameter space. We look into the desirable properties of a geometry for neural networks and suggest an optimization algorithm that is tied to that geometry.

3. The Geometry of Optimization: Rescaling and Unbalanceness

In this section, we look at the behavior of the Euclidean geometry under rescaling and unbalanceness. One of the special properties of RELU activation function is non-negative homogeneity. That is, for any scalar $c \geq 0$ and any $x \in \mathbb{R}$, we have $\sigma_{\text{RELU}}(c \cdot x) = c \cdot \sigma_{\text{RELU}}(x)$. This interesting property allows the network to be rescaled without changing the function computed by the network. We define the *rescaling function* $\rho_{c,v}(w)$, such that given the weights of the network $w : E \rightarrow \mathbb{R}$, a constant $c > 0$, and a node v , the rescaling function multiplies the incoming edges and divides the outgoing edges of v by c . That is, $\rho_{c,v}(w)$ maps w to the weights $\tilde{w}$ for the rescaled network, where for any $(u_1 \rightarrow u_2) \in E$:

$$\tilde{w}_{(u_1 \rightarrow u_2)} = \begin{cases} c \cdot w_{(u_1 \rightarrow u_2)} & u_2 = v, \\ \frac{1}{c} w_{(u_1 \rightarrow u_2)} & u_1 = v, \\ w_{(u_1 \rightarrow u_2)} & \text{otherwise.} \end{cases} \quad (1)$$

It is easy to see that the rescaled network computes the same function, i.e. $f_{G,w,\sigma_{\text{RELU}}} = f_{G,\rho_{c,v}(w),\sigma_{\text{RELU}}}$. We say that the two networks with weights w and $\tilde{w}$ are *rescaling equivalent* denoted by $w \sim \tilde{w}$ if and only if one of them can be transformed to another by applying a sequence of rescaling functions $\rho_{c,v}$.

Given a training set $\mathcal{S} = \{(x_1, y_n), \dots, (x_n, y_n)\}$, our goal is to minimize the following objective function:

$$L(w) = \frac{1}{n} \sum_{i=1}^n \ell(f_w(x_i), y_i). \quad (2)$$

Let $w^{(t)}$ be the weights at step t of the optimization. We consider update step of the following form $w^{(t+1)} = w^{(t)} + \Delta w^{(t+1)}$. For example, for gradient descent, we have $\Delta w^{(t+1)} = -\eta \nabla L(w^{(t)})$, where η is the step-size. In the stochastic setting, such as SGD or mini-batch gradient descent, we calculate the gradient on a small subset of the training set.

Since *rescaling equivalent* networks compute the same function, it is desirable to have an update rule that is not affected by rescaling. We call an optimization method *rescaling invariant* if the updates of rescaling equivalent networks are rescaling equivalent. That is, if we start at either one of the two rescaling equivalent weight vectors $\tilde{w}^{(0)} \sim w^{(0)}$, after applying t update steps separately on $\tilde{w}^{(0)}$ and $w^{(0)}$, they will remain rescaling equivalent and we have $\tilde{w}^{(t)} \sim w^{(t)}$.

Unfortunately, gradient descent is *not* rescaling invariant. The main problem with the gradient updates is that scaling down the weights of an edge will also scale up the gradient which, as we see later, is exactly the opposite of what is expected from a rescaling invariant update.

Furthermore, gradient descent performs very poorly on “unbalanced” networks. We say that a network is *balanced* if the norm of incoming weights to different units are roughly the same or within a small range. For example, Figure a shows a huge gap in the performance of SGD initialized with a randomly generated balanced network $w^{(0)}$, when training on MNIST, compared to a network initialized with unbalanced weights $\tilde{w}^{(0)}$. Here $\tilde{w}^{(0)}$ is generated by applying a sequence of random rescaling functions on $w^{(0)}$ (and therefore $w^{(0)} \sim \tilde{w}^{(0)}$).

In an unbalanced network, gradient descent updates could blow up the smaller weights, while keeping the larger weights almost unchanged. This is illustrated in Figure b. If this were the only issue, one could scale down all the weights after each update. However, in an unbalanced network, the relative changes in the weights are also very different compared to a balanced network. For

example, Figure c shows how two rescaling equivalent networks could end up computing a very different function after only a single update.

4. Magnitude/Scale measures for deep networks

Following Neyshabur et al. (2015b), we consider the grouping of weights going into each node of the network. This forms the following generic group-norm type regularizer, parametrized by $1 \leq p, q \leq \infty$:

$$\mu_{p,q}(w) = \left(\sum_{v \in V} \left(\sum_{(u \rightarrow v) \in E} |w_{(u \rightarrow v)}|^p \right)^{q/p} \right)^{1/q}. \quad (3)$$

Two simple cases of above group-norm are $p = q = 1$ and $p = q = 2$ that correspond to overall ℓ_1 regularization and weight decay respectively. Another form of regularization that is shown to be very effective in RELU networks is the max-norm regularization, which is the maximum over all units of norm of incoming edge to the unit³ (Goodfellow et al., 2013; Srivastava et al., 2014). The max-norm correspond to “per-unit” regularization when we set $q = \infty$ in equation (4) and can be written in the following form:

$$\mu_{p,\infty}(w) = \sup_{v \in V} \left(\sum_{(u \rightarrow v) \in E} |w_{(u \rightarrow v)}|^p \right)^{1/p} \quad (4)$$

Weight decay is probably the most commonly used regularizer. On the other hand, per-unit regularization might not seem ideal as it is very extreme in the sense that the value of regularizer corresponds to the highest value among all nodes. However, the situation is very different for networks with RELU activations (and other activation functions with non-negative homogeneity property). In these cases, per-unit ℓ_2 regularization has shown to be very effective (Srivastava et al., 2014). The main reason could be because RELU networks can be rebalanced in such a way that all hidden units have the same norm. Hence, per-unit regularization will not be a crude measure anymore.

Since $\mu_{p,\infty}$ is not rescaling invariant and the values of the scale measure are different for rescaling equivalent networks, it is desirable to look for the minimum value of a regularizer among all rescaling equivalent networks. Surprisingly, for a feed-forward network, the minimum ℓ_p per-unit regularizer among all rescaling equivalent networks can be efficiently computed by a single forward step. To see this, we consider the vector $\pi(w)$, the *path vector*, where the number of coordinates of $\pi(w)$ is equal to the total number of paths from the input to output units and each coordinate of $\pi(w)$ is the equal to the product of weights along a path from an input nodes to an output node. The ℓ_p -path regularizer is then defined as the ℓ_p norm of $\pi(w)$ (Neyshabur et al., 2015b):

$$\phi_p(w) = \|\pi(w)\|_p = \left(\sum_{v_{in}[i] \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \dots \xrightarrow{e_d} v_{out}[j]} \left| \prod_{k=1}^d w_{e_k} \right|^p \right)^{1/p} \quad (5)$$

3. This definition of max-norm is a bit different than the one used in the context of matrix factorization (Srebro and Shraibman, 2005). The later is similar to the minimum upper bound over ℓ_2 norm of both outgoing edges from the input units and incoming edges to the output units in a two layer feed-forward network.

The following Lemma establishes that the ℓ_p -path regularizer corresponds to the minimum over all equivalent networks of the per-unit ℓ_p norm:

Lemma 4.1 (Neyshabur et al. (2015b)) $\phi_p(w) = \min_{\tilde{w} \sim w} \left(\mu_{p,\infty}(\tilde{w}) \right)^d$

The definition (5) of the ℓ_p -path regularizer involves an exponential number of terms. But it can be computed efficiently by dynamic programming in a single forward step using the following equivalent form as nested sums:

$$\phi_p(w) = \left(\sum_{(v_{d-1} \rightarrow v_{\text{out}}[j]) \in E} |w_{(v_{d-1} \rightarrow v_{\text{out}}[j])}|^p \sum_{(v_{d-2} \rightarrow v_{d-1}) \in E} \cdots \sum_{(v_{\text{in}}[i] \rightarrow v_1) \in E} |w_{(v_{\text{in}}[i] \rightarrow v_1)}|^p \right)^{1/p}$$

A straightforward consequence of Lemma 4.1 is that the ℓ_p path-regularizer ϕ_p is invariant to rescaling, i.e. for any $\tilde{w} \sim w$, $\phi_p(\tilde{w}) = \phi_p(w)$.

5. Path-SGD: An Approximate Path-Regularized Steepest Descent

Motivated by empirical performance of max-norm regularization and the fact that path-regularizer is invariant to rescaling, we are interested in deriving the steepest descent direction with respect to the path regularizer $\phi_p(w)$:

$$\begin{aligned} w^{(t+1)} &= \arg \min_w \eta \langle \nabla L(w^{(t)}), w \rangle + \frac{1}{2} \left\| \pi(w) - \pi(w^{(t)}) \right\|_p^2 \\ &= \arg \min_w \eta \langle \nabla L(w^{(t)}), w \rangle + \left(\sum_{v_{\text{in}}[i] \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \dots \xrightarrow{e_d} v_{\text{out}}[j]} \left(\prod_{k=1}^d w_{e_k} - \prod_{k=1}^d w_{e_k}^{(t)} \right)^p \right)^{2/p} \\ &= \arg \min_w J^{(t)}(w) \end{aligned} \tag{6}$$

The steepest descent step (6) is hard to calculate exactly. Instead, we will update each coordinate w_e independently (and synchronously) based on (6). That is:

$$w_e^{(t+1)} = \arg \min_{w_e} J^{(t)}(w) \quad \text{s.t. } \forall_{e' \neq e} w_{e'} = w_{e'}^{(t)} \tag{7}$$

Taking the partial derivative with respect to w_e and setting it to zero we obtain:

$$0 = \eta \frac{\partial L}{\partial w_e}(w^{(t)}) - (w_e - w_e^{(t)}) \left(\sum_{v_{\text{in}}[i] \dots \xrightarrow{e} \dots v_{\text{out}}[j]} \prod_{e_k \neq e} |w_{e_k}^{(t)}|^p \right)^{2/p}$$

where $v_{\text{in}}[i] \dots \xrightarrow{e} \dots v_{\text{out}}[j]$ denotes the paths from any input unit i to any output unit j that includes e . Solving for w_e gives us the following update rule:

$$\hat{w}_e^{(t+1)} = w_e^{(t)} - \frac{\eta}{\gamma_p(w^{(t)}, e)} \frac{\partial L}{\partial w}(w^{(t)}) \tag{8}$$

where $\gamma_p(w, e)$ is given as

$$\gamma_p(w, e) = \left(\sum_{v_{\text{in}}[i] \xrightarrow{e} \dots v_{\text{out}}[j]} \prod_{e_k \neq e} |w_{e_k}|^p \right)^{2/p} \quad (9)$$

We call the optimization using the update rule (8) path-normalized gradient descent. When used in stochastic settings, we refer to it as Path-SGD.

Now that we know Path-SGD is an approximate steepest descent with respect to the path-regularizer, we can ask whether or not this makes Path-SGD a *rescaling invariant* optimization method. The next theorem proves that Path-SGD is indeed rescaling invariant.

Theorem 5.1 *Path-SGD is rescaling invariant.*

Proof It is sufficient to prove that using the update rule (8), for any $c > 0$ and any $v \in E$, if $\tilde{w}^{(t)} = \rho_{c,v}(w^{(t)})$, then $\tilde{w}^{(t+1)} = \rho_{c,v}(w^{(t+1)})$. For any edge e in the network, if e is neither incoming nor outgoing edge of the node v , then $\tilde{w}(e) = w(e)$, and since the gradient is also the same for edge e we have $\tilde{w}_e^{(t+1)} = w_e^{(t+1)}$. However, if e is an incoming edge to v , we have that $\tilde{w}^{(t)}(e) = cw^{(t)}(e)$. Moreover, since the outgoing edges of v are divided by c , we get $\gamma_p(\tilde{w}^{(t)}, e) = \frac{\gamma_p(w^{(t)}, e)}{c^2}$ and $\frac{\partial L}{\partial w_e}(\tilde{w}^{(t)}) = \frac{\partial L}{c \partial w_e}(w^{(t)})$. Therefore,

$$\begin{aligned} \tilde{w}_e^{(t+1)} &= cw_e^{(t)} - \frac{c^2 \eta}{\gamma_p(w^{(t)}, e)} \frac{\partial L}{c \partial w_e}(w^{(t)}) \\ &= c \left(w^{(t)} - \frac{\eta}{\gamma_p(w^{(t)}, e)} \frac{\partial L}{\partial w_e}(w^{(t)}) \right) = cw_e^{(t+1)}. \end{aligned}$$

A similar argument proves the invariance of Path-SGD update rule for outgoing edges of v . Therefore, Path-SGD is rescaling invariant. $\blacksquare$

Efficient Implementation: The Path-SGD update rule (8), in the way it is written, needs to consider all the paths, which is exponential in the depth of the network. However, it can be calculated in a time that is no more than a forward-backward step on a single data point. That is, in a mini-batch setting with batch size B , if the backpropagation on the mini-batch can be done in time BT , the running time of the Path-SGD on the mini-batch will be roughly $(B + 1)T$ – a very moderate runtime increase with typical mini-batch sizes of hundreds or thousands of points. Algorithm 1 shows an efficient implementation of the Path-SGD update rule.

We next compare Path-SGD to other optimization methods in both balanced and unbalanced settings.

6. Experiments on Path-SGD

We compare ℓ_2 -Path-SGD to two commonly used optimization methods in deep learning, SGD and AdaGrad. We conduct our experiments on four common benchmark datasets: the standard MNIST dataset of handwritten digits (LeCun et al., 1998); CIFAR-10 and CIFAR-100 datasets

Algorithm 1 Path-SGD update rule

```

1:  $\forall_{v \in V_{\text{in}}^0} \gamma_{\text{in}}(v) = 1$  ▷ Initialization
2:  $\forall_{v \in V_{\text{out}}^0} \gamma_{\text{out}}(v) = 1$ 
3: for  $i = 1$  to  $d$  do
4:    $\forall_{v \in V_{\text{in}}^i} \gamma_{\text{in}}(v) = \sum_{(u \rightarrow v) \in E} \gamma_{\text{in}}(u) |w_{(u,v)}|^p$ 
5:    $\forall_{v \in V_{\text{out}}^i} \gamma_{\text{out}}(v) = \sum_{(v \rightarrow u) \in E} |w_{(v,u)}|^p \gamma_{\text{out}}(u)$ 
6: end for
7:  $\forall_{(u \rightarrow v) \in E} \gamma(w^{(t)}, (u, v)) = \gamma_{\text{in}}(u)^{2/p} \gamma_{\text{out}}(v)^{2/p}$ 
8:  $\forall_{e \in E} w_e^{(t+1)} = w_e^{(t)} - \frac{\eta}{\gamma(w^{(t)}, e)} \frac{\partial L}{\partial w_e}(w^{(t)})$  ▷ Update Rule

```

Table 1: General information on datasets used in the experiments on feedforward networks.

Data Set	Dimensionality	Classes	Training Set	Test Set
CIFAR-10	3072 (32×32 color)	10	50000	10000
CIFAR-100	3072 (32×32 color)	100	50000	10000
MNIST	784 (28×28 grayscale)	10	60000	10000
SVHN	3072 (32×32 color)	10	73257	26032

of tiny images of natural scenes (Krizhevsky and Hinton, 2009); and Street View House Numbers (SVHN) dataset containing color images of house numbers collected by Google Street View (Netzer et al., 2011). Details of the datasets are shown in Table 1.

In all of our experiments, we trained feed-forward networks with two hidden layers, each containing 4000 hidden units. We used mini-batches of size 100 and the step-size of $10^{-\alpha}$, where α is an integer between 0 and 10. To choose α , for each dataset, we considered the validation errors over the validation set (10000 randomly chosen points that are kept out during the initial training) and picked the one that reaches the minimum error faster. We then trained the network over the entire training set. All the networks were trained both with and without dropout. When training with dropout, at each update step, we retained each unit with probability 0.5.

The optimization results are shown in Figure 3. For each of the four datasets, the plots for objective function (cross-entropy), the training error and the test error are shown from left to right where in each plot the values are reported on different epochs during the optimization. The dropout is used for the experiments on CIFAR-100 and SVHN. Please see Neyshabur et al. (2015a) for a more complete set of experimental results.

We can see in Figure 3 that not only does Path-SGD often get to the same value of objective function, training and test error faster, but also the plots for test errors demonstrate that implicit regularization due to steepest descent with respect to path-regularizer leads to a solution that generalizes better. This provides further evidence on the role of implicit regularization in deep learning.

The results suggest that Path-SGD outperforms SGD and AdaGrad in two different ways. First, it can achieve the same accuracy much faster and second, the implicit regularization by Path-SGD leads to a local minima that can generalize better even when the training error is zero. This can be better analyzed by looking at the plots for more number of epochs which we have provided in Neyshabur et al. (2015a). We should also point that Path-SGD can be easily combined with

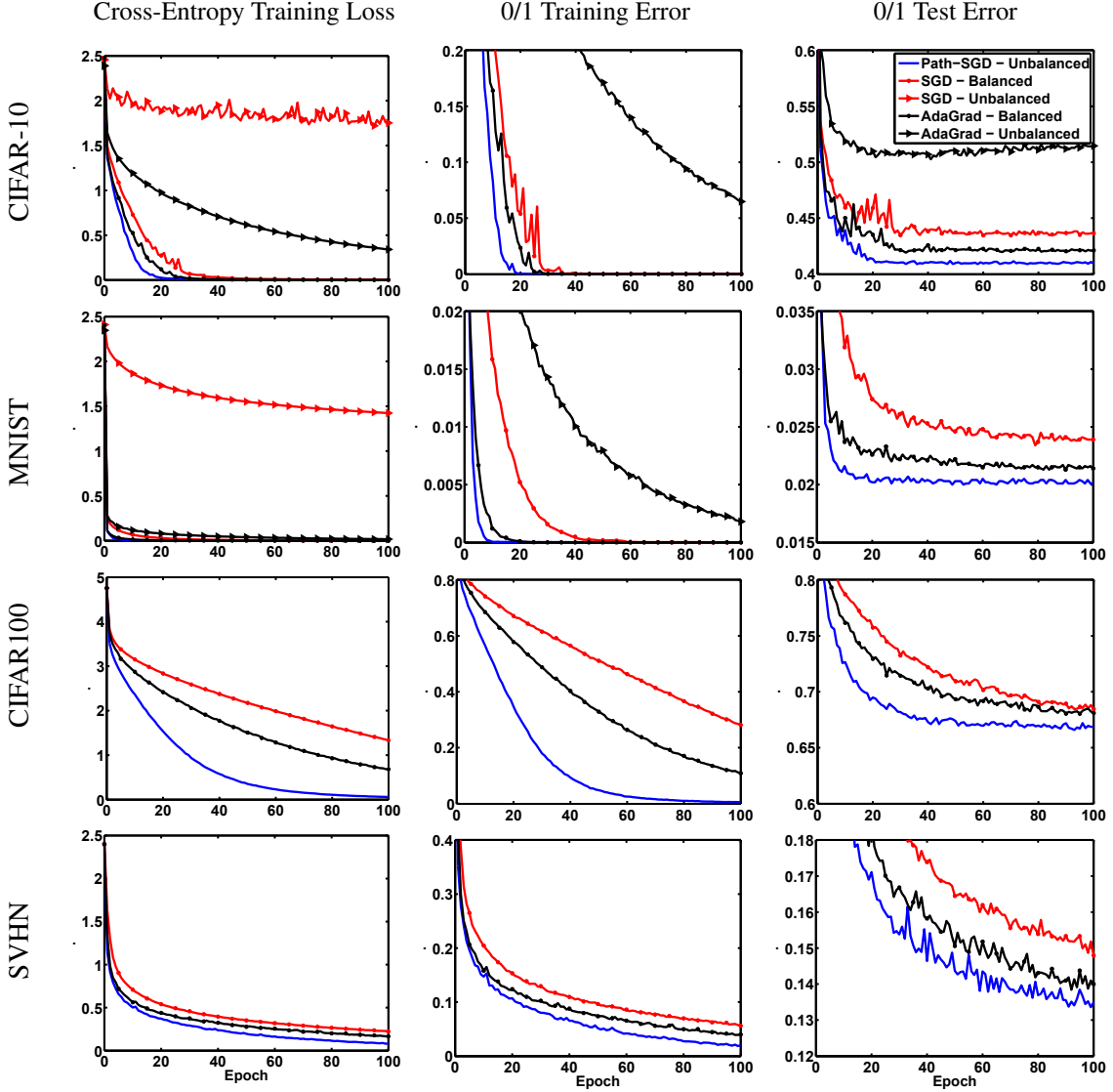


Figure 3: Learning curves using different optimization methods for 4 datasets without dropout. Left panel displays the cross-entropy objective function; middle and right panels show the corresponding values of the training and test errors, where the values are reported on different epochs during the course of optimization. We tried both balanced and unbalanced initializations. In balanced initialization, incoming weights to each unit v are initialized to i.i.d samples from a Gaussian distribution with standard deviation $1/\sqrt{\text{fan-in}(v)}$. In the unbalanced setting, we first initialized the weights to be the same as the balanced weights. We then picked 2000 hidden units randomly with replacement. For each unit, we multiplied its incoming edge and divided its outgoing edge by $10c$, where c was chosen randomly from log-normal distribution. Although we proved that Path-SGD updates are the same for balanced and unbalanced initializations, to verify that despite numerical issues they are indeed identical, we trained Path-SGD with both balanced and unbalanced initializations. Since the curves were exactly the same we only show a single curve. Best viewed in color.

AdaGrad or Adam to take advantage of the adaptive stepsize or used together with a momentum term. This could potentially perform even better compare to Path-SGD.

7. Discussion

We demonstrated the implicit regularization in deep learning through experiments and discussed the importance of geometry of optimization in finding a “low complexity” solution. Based on that, we revisited the choice of the Euclidean geometry on the weights of RELU networks, suggested an alternative optimization method approximately corresponding to a different geometry, and showed that using such an alternative geometry can be beneficial. In this work we show proof-of-concept success, and we expect Path-SGD to be beneficial also in large-scale training for very deep convolutional networks. Combining Path-SGD with AdaGrad, with momentum or with other optimization heuristics might further enhance results.

Although we do believe Path-SGD is a very good optimization method, and is an easy plug-in for SGD, we hope this work will also inspire others to consider other geometries, other regularizers and perhaps better, update rules. A particular property of Path-SGD is its rescaling invariance, which we argue is appropriate for RELU networks. But Path-SGD is certainly not the only rescaling invariant update possible, and other invariant geometries might be even better.

Finally, we choose to use steepest descent because of its simplicity of implementation. A better choice might be mirror descent with respect to an appropriate potential function, but such a construction seems particularly challenging considering the non-convexity of neural networks.

Acknowledgments

Research was partially funded by NSF award IIS-1302662 and Intel ICRI-CI.

References

- Martin Anthony and Peter L Bartlett. *Neural network learning: Theoretical foundations*. Cambridge University Press, 1999.
- Amit Daniely, Nati Linial, and Shai Shalev-Shwartz. From average case complexity to improper learning complexity. *STOC*, 2014.
- Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, and Yoshua Bengio. Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML*, pages 1319–1327, 2013.
- Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009.
- Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, 2011.
- Behnam Neyshabur, Ruslan R Salakhutdinov, and Nati Srebro. Path-sgd: Path-normalized optimization in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2413–2421, 2015a.

- Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. Norm-based capacity control in neural networks. In *The 28th Conference on Learning Theory*, pages 1376–1401, 2015b.
- Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. In search of the real inductive bias: On the role of implicit regularization in deep learning. *International Conference on Learning Representations (ICLR) workshop track*, 2015c.
- Adam R Klivans and Alexander A Sherstov. Cryptographic hardness for learning intersections of halfspaces. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 553–562. IEEE, 2006.
- Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006.
- Nathan Srebro and Adi Shraibman. Rank, trace-norm and max-norm. In *Learning Theory*, pages 545–560. Springer, 2005.
- Nathan Srebro, Karthik Sridharan, and Ambuj Tewari. On the universality of online mirror descent. In *Advances in neural information processing systems*, pages 2645–2653, 2011.
- Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.