

MOLLIFYING NETWORKS

Caglar Gulcehre¹, Marcin Moczulski^{2,*}, Francesco Visin^{3,*}, Yoshua Bengio¹

¹ University of Montreal, ² University of Oxford, ³ Politecnico di Milano

ABSTRACT

The optimization of deep neural networks can be more challenging than the traditional convex optimization problems due to highly non-convex nature of the loss function, e.g. it can involve pathological landscapes such as saddle-surfaces that can be difficult to escape from for algorithms based on simple gradient descent. In this paper, we attack the problem of optimization of highly non-convex neural networks objectives by starting with a smoothed – or *mollified* – objective function which becomes more complex as the training proceeds. Our proposition is inspired by the recent studies in continuation methods: similarly to curriculum methods, we begin by learning an easier (possibly convex) objective function and let it evolve during training until it eventually becomes the original, difficult to optimize objective function. The complexity of the mollified networks is controlled by a single hyperparameter that is annealed during training. We show improvements on various difficult optimization tasks and establish a relationship between recent works on continuation methods for neural networks and mollifiers.

1 INTRODUCTION

In the last few years, deep neural networks – i.e. convolutional networks (LeCun et al., 1989), LSTMs (Hochreiter & Schmidhuber, 1997a) or GRUs (Cho et al., 2014) – set the state of the art on a range of challenging tasks (Szegedy et al., 2014; Visin et al., 2015; Hinton et al., 2012; Sutskever et al., 2014; Bahdanau et al., 2014; Mnih et al., 2013; Silver et al., 2016). However when trained with variants of SGD (Bottou, 1998) deep networks can be difficult to optimize due to their highly non-linear and non-convex nature (Choromanska et al., 2014; Dauphin et al., 2014).

A number of approaches were proposed to alleviate the difficulty of optimization: addressing the problem of the internal covariate shift with Batch Normalization (Ioffe & Szegedy, 2015), learning with a curriculum (Bengio et al., 2009), recently an approach to train RNNs with diffusion process (Mobahi, 2016), and graduated optimization (Hazan et al., 2015). The impact of noise injection on the behavior of modern deep learning methods has been explored by Neelakantan et al. (2015a). Hazan et al. (2015) have shown that injecting a particular noise and scheduling it carefully, can guarantee the convergence in $O(1/\sigma^2\epsilon^2)$ steps for ϵ -optimal and σ -nice functions. Similar to our work graduated optimization optimizes a smoothed objective function without performing expensive convolutions. Injecting noise to the activation functions and scheduling it have been recently shown to improve the performance on a wide variety of tasks (Gulcehre et al., 2016).

We connect the ideas of curriculum learning and continuation methods with those arising from models with skip connections and using layers that compute near-identity transformations. Skip connections allow to train very deep residual and highway architectures (He et al., 2015; Srivastava et al., 2015) by skipping layers or block of layers. Similarly, it has been shown that stochastically changing the depth of a network during training (Huang et al., 2016b) does not prevent convergence and enables better generalization performance.

We discuss the idea of mollification for neural networks – a form of differentiable smoothing of the loss function connected to noisy activations – which in our case can be interpreted as a form of adaptive noise injection which is controlled by a single hyperparameter. Inspired by Huang et al. (2016b), we use a hyperparameter to stochastically control the depth of our network. This allows us to start the optimization from a *convex objective function* (as long as the optimized criterion is convex, e.g. linear or logistic regression) and to slowly introduce more complexity into the model by annealing the hyperparameter, thus making the network deeper and increasingly non-linear.

* This work was done while these students were interning at the MILA lab in University of Montreal.

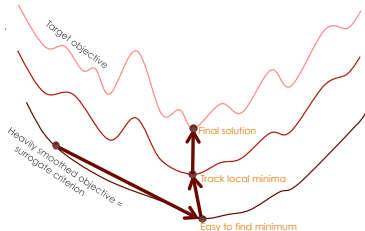


Figure 1: A sequence of optimization problems of increasing complexity, where the first ones are easy to solve but only the last one corresponds to the actual problem of interest. It is possible to tackle the problems in order, starting each time at the solution of the previous one and tracking the local minima along the way.

An important difference of our work compared to injecting noise to the gradients as it is explored in (Hazan et al., 2015; Neelakantan et al., 2015b) is that we inject the noise in the forward computation of the graph and we shape the cost function directly. As a result the cost function for the mollified network both at the test time and during the training are consistent and this makes the early-stopping much easier.

2 MOLLIFYING OBJECTIVE FUNCTIONS

2.1 CONTINUATION AND ANNEALING METHODS

Continuation methods and *simulated annealing* provide a general strategy to reduce the impact of local minima and deal with non-convex, continuous, but not necessarily everywhere differentiable objective functions by smoothing the original objective function and gradually reducing the amount of smoothing during training (Allgower & Georg, 1980) (see Fig. 1).

In machine learning, approaches based on curriculum learning (Bengio et al., 2009) are inspired by this principle and define a sequence of gradually more difficult training tasks (or training distributions) that eventually converge to the task of interest.

In the context of stochastic gradient descent, we use a stochastic estimation of the gradient for the smoothed objective function. This is convenient because it may not be analytically feasible to compute the smoothed function, but a Monte-Carlo estimate can often be obtained easily.

In this paper we construct a sequence of smoothed objective functions obtained with a form of mollification and we progressively optimize them. The training procedure iterates over the sequence of objective functions starting from the simpler ones – i.e. with a smoother loss surface – and moving towards more complex ones until the last, original, objective function is reached.¹

2.2 MOLLIFIERS AND WEAK GRADIENTS

We smooth the loss function \mathcal{L} , which is parametrized by $\theta \in \mathbb{R}^n$, by convolving it with another function $K(\cdot)$ with stride $\tau \in \mathbb{R}^n$:

$$\mathcal{L}_K(\theta) = (\mathcal{L} * K)(\theta) = \int_{-\infty}^{+\infty} \mathcal{L}(\theta - \tau)K(\tau)d\tau \quad (1)$$

Although there are many choices for the function $K(\cdot)$, we focus on those that satisfy the definition of a mollifier.

A mollifier is an infinitely differentiable function that behaves like an approximate identity in the group of convolutions of integrable functions. If $K(\cdot)$ is an infinitely differentiable function, that converges to the Dirac delta function when appropriately rescaled and for any integrable function \mathcal{L} , then it is a mollifier:

$$\mathcal{L}(\theta) = \lim_{\epsilon \rightarrow 0} \int \epsilon^{-n} K(\tau/\epsilon)\mathcal{L}(\theta - \tau)d\tau. \quad (2)$$

¹We plan to release the source code of the models and experiments under, http://github.com/caglar/molly_nets/.

If we choose $K(\cdot)$ to be a mollifier and obtain the smoothed loss function \mathcal{L}_K as in Eqn. 1, we can take its gradient with respect to θ using directly the result from Evans (1998):

$$\nabla_{\theta} \mathcal{L}_K(\theta) = \nabla_{\theta} (\mathcal{L} * K)(\theta) = (\mathcal{L} * \nabla K)(\theta). \quad (3)$$

To relate the resulting gradient $\nabla_{\theta} \mathcal{L}_K$ to the gradient of the original function \mathcal{L} , we introduce the notion of weak gradient, i.e. an extension to the idea of weak/distributional derivatives to functions with multidimensional arguments, such as loss functions of neural networks.

For an integrable function \mathcal{L} in space $\mathcal{L} \in L([a, b])$, $\mathbf{g} \in L([a, b]^n)$ is a n -dimensional weak gradient of \mathcal{L} if it satisfies:

$$\int_C \mathbf{g}(\tau) K(\tau) d\tau = - \int_C \mathcal{L}(\tau) \nabla K(\tau) d\tau, \quad (4)$$

where $K(\tau)$ is an infinitely differentiable function vanishing at infinity, $C \in [a, b]^n$ and $\tau \in \mathbb{R}^n$.

As long as the chosen $K(\cdot)$ fulfills the definition of a mollifier we can use Eqn. 3 and Eqn. 4² to rewrite the gradient as:

$$\nabla_{\theta} \mathcal{L}_K(\theta) = (\mathcal{L} * \nabla K)(\theta) \quad \text{by Eqn. 3} \quad (5)$$

$$= \int_C \mathcal{L}(\theta - \tau) \nabla K(\tau) d\tau \quad (6)$$

$$= - \int_C \mathbf{g}(\theta - \tau) K(\tau) d\tau \quad \text{by Eqn. 4} \quad (7)$$

For a differentiable almost everywhere function \mathcal{L} , the weak gradient $\mathbf{g}(\theta)$ is equal to $\nabla_{\theta} \mathcal{L}$ almost everywhere. With a slight abuse of notation we can therefore write:

$$\nabla_{\theta} \mathcal{L}_K(\theta) = - \int_C \nabla_{\theta} \mathcal{L}(\theta - \tau) K(\tau) d\tau \quad (8)$$

2.3 GAUSSIAN MOLLIFIERS

It is possible to use the standard Gaussian distribution $\mathcal{N}(0, \mathbf{I})$ as a mollifier $K(\cdot)$, as it satisfies the desired properties: it is infinitely differentiable, a sequence of properly rescaled Gaussian distributions converges to the Dirac delta function and it vanishes in infinity. With such a $K(\cdot)$ the gradient becomes:

$$\nabla_{\theta} \mathcal{L}_{K=\mathcal{N}}(\theta) = - \int_C \nabla_{\theta} \mathcal{L}(\theta - \tau) p(\tau) d\tau \quad (9)$$

$$= \mathbb{E}_{\tau} [\nabla_{\theta} \mathcal{L}(\theta - \tau)], \text{ with } \tau \sim \mathcal{N}(0, \mathbf{I}) \quad (10)$$

Exploiting the fact that a Gaussian distribution is a mollifier, we can focus on a sequence of mollifications indexed by scaling parameter ϵ introduced in Eqn. 2. A single element of this sequence takes the following form:

$$\nabla_{\theta} \mathcal{L}_{\mathcal{N}, \epsilon}(\theta) = - \int_C \nabla_{\theta} \mathcal{L}(\theta - \tau) \epsilon^{-1} p(\tau/\epsilon) d\tau \quad (11)$$

$$= \mathbb{E}_{\tau} [\nabla_{\theta} \mathcal{L}(\theta - \tau)], \text{ with } \tau \sim \mathcal{N}(0, \epsilon^2 \mathbf{I}) \quad (12)$$

Replacing ϵ with σ yields a sequence of mollifications indexed by σ :

$$\nabla_{\theta} \mathcal{L}_{\mathcal{N}, \sigma}(\theta) = \mathbb{E}_{\tau} [\nabla_{\theta} \mathcal{L}(\theta - \tau)], \text{ with } \tau \sim \mathcal{N}(0, \sigma^2 \mathbf{I}) \quad (13)$$

with the following property (by Eqn. 2):

$$\lim_{\sigma \rightarrow 0} \nabla_{\theta} \mathcal{L}_{\mathcal{N}, \sigma}(\theta) = \nabla_{\theta} \mathcal{L}(\theta) \quad (14)$$

An intuitive interpretation of the result is that σ determines the standard deviation of a mollifying Gaussian and is annealed in order to construct a sequence of gradually less "blurred" and closer

²We omit for brevity the algebraic details involved with a translation of the argument.

approximations to \mathcal{L} . This is consistent with the property that when σ is annealed to zero we are optimizing the original function \mathcal{L} .

So far we obtained the mollified version $\mathcal{L}_K(\boldsymbol{\theta})$ of the cost function $\mathcal{L}(\boldsymbol{\theta})$ by convolving it with a mollifier $K(\boldsymbol{\theta})$. The kernel $K(\boldsymbol{\theta})$ corresponds to the average effect of injecting noise ξ sampled from standard Normal distribution. The amount of noise controls the amount of smoothing. Gradually reducing the noise during training is related to a form of *simulated annealing* (Kirkpatrick et al., 1983). Similarly to the analysis in Mobahi (2016), we can write a Monte-Carlo estimate of $\mathcal{L}_K(\boldsymbol{\theta}) = (\mathcal{L} * K)(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^N \mathcal{L}(\boldsymbol{\theta} - \xi^{(i)})$. We provide the derivation and the gradient of this equation in Appendix A.

The Monte-Carlo estimators of the mollifiers can be easily implemented with neural networks, where the layers typically have the form:

$$\mathbf{h}^l = f(\mathbf{W}^l \mathbf{h}^{l-1}) \quad (15)$$

with \mathbf{h}^{l-1} a vector of activations from the previous layer in the hierarchy, \mathbf{W}^l a matrix representing a linear transformation and f an element-wise non-linearity of choice.

A mollification of such a layer can be formulated as:

$$\mathbf{h}^l = f((\mathbf{W}^l - \xi^l) \mathbf{h}^{l-1}), \text{ where } \xi^l \sim \mathcal{N}(\mu, \sigma^2) \quad (16)$$

From Eqn. 16, it is easy to see that both weight noise methods proposed by Hinton & van Camp (1993) and Graves (2011) can be seen as a variation of Monte-Carlo estimate of mollifiers.

2.4 GENERALIZED AND NOISY MOLLIFIERS

We introduce a generalization of the concept of mollifiers that encompasses the approach we explored here and that is targeted during optimization via a continuation method using stochastic gradient descent.

Definition 2.1. (Generalized Mollifier). A *generalized mollifier* is an operator, where $T_\sigma(f)$ defines a mapping between two functions, such that $T_\sigma : f \rightarrow f^*$.

$$\lim_{\sigma \rightarrow 0} T_\sigma f = f, \quad (17)$$

$$f^0 = \lim_{\sigma \rightarrow \infty} T_\sigma f \quad \text{is an identity function} \quad (18)$$

$$\frac{\partial(T_\sigma f)(x)}{\partial x} \quad \text{exists } \forall x, \sigma > 0 \quad (19)$$

In addition, we consider noisy mollifiers which can be defined as an expected value of a stochastic function $\phi(x, \xi)$ under some noise source ξ with variance σ :

$$(T_\sigma f)(x) = E_\xi[\phi(x, \xi_\sigma)] \quad (20)$$

Definition 2.2. (Noisy Mollifier). We call a stochastic function $\phi(x, \xi_\sigma)$ with input x and noise ξ a *noisy mollifier* if its expected value corresponds to the application of a generalized mollifier T_σ , as per Eqn. 20.

The composition of two noisy mollifiers sharing the same σ is also a noisy mollifier, since the three properties in the definition (Eqns. 17,18,19) are still satisfied. When $\sigma = 0$ no noise is injected and therefore the original function will be optimized. If $\sigma \rightarrow \infty$ instead, the function will become an identity function. Thus, for instance, if we mollify each layer of a feed-forward network except the output layer, when $\sigma \rightarrow \infty$ all the mollified layers will become identity function and the objective function of the network with respect to its inputs will be convex.

Consequently, corrupting separately the activation function of each level of a deep neural network (but with a shared noise level σ) and annealing σ yields a noisy mollifier for the objective function. This is related to the work of Mobahi (2016), who recently introduced a way of analytically smoothing of the non-linearities to help the training of recurrent networks. The differences of that approach from our algorithm is two-fold: we use a noisy mollifier (rather than an analytic smoothing of the network’s non-linearities) and we introduce (in the next section) a particular form of the noisy mollifier that empirically proved to work well.

Shaping the cost function to define a sequence of costs that are progressing from easier to more difficult ones can be related to the reward shaping (Ng et al., 1999; Ng, 2003) algorithms. In our algorithm, we shape the cost and the model architecture itself, rather than rewards or the targets in order to make the optimization easier. In that sense, reward shaping can be considered to be more closer to curriculum learning.

3 METHOD

We propose an algorithm to mollify the cost of a neural network which also addresses an important drawback of the previously proposed noisy training procedures: as the noise gets larger, it can dominate the learning process and lead the algorithm to perform a random walk on the energy landscape of the objective function. Conversely in our algorithm, as the noise gets larger gradient descent minimizes a simpler (e.g. convex) but still meaningful objective function.

We define the desired behavior of the network in the limit cases where the noise is very large or very small, and modify the model architecture accordingly. Specifically, during training we minimize a sequence of increasingly complex noisy objectives $L = (\mathcal{L}^1(\theta; \xi_{\sigma_1}), \mathcal{L}^2(\theta; \xi_{\sigma_2}), \dots, \mathcal{L}^k(\theta; \xi_{\sigma_k}))$ that we obtain by annealing the scale (variance) of the noise σ_i . Let us note that our algorithm satisfies the fundamental properties of the *generalized and noisy mollifiers* that we introduced earlier.

We use a noisy mollifier based on our definition in Section 2.4. Instead of convolving the objective function with a kernel:

1. We start the training by optimizing a convex objective function that is obtained by configuring all the layers between the input and the last cost layer to compute an identity function, i.e., by skipping both the affine transformations and the blocks followed by nonlinearities.
2. During training, the magnitude of noise which is proportional to p is annealed, allowing to gradually evolve from identity transformations to linear transformations between the layers.
3. Simultaneously, as we decrease the p , the noisy mollification procedure allows the element-wise activation functions to gradually change from linear to be nonlinear.

4 SIMPLIFYING THE OBJECTIVE FUNCTION FOR FEEDFORWARD NETWORKS

For *every unit* of each layer, we either copy the activation (output) of the corresponding unit of the previous layer (the identity path in Figure 2) or output a noisy activation $\tilde{\mathbf{h}}^l$ of a non-linear transformation of it $\psi(\mathbf{h}^{l-1}, \xi; \mathbf{W}^l)$, where ξ is noise, \mathbf{W}^l is a weight matrix applied on \mathbf{h}^{l-1} and π is a vector of binary decisions for each unit (the convolutional path in Figure 2):

$$\tilde{\mathbf{h}}^l = \psi(\mathbf{h}^{l-1}, \xi; \mathbf{W}^l) \quad (21)$$

$$\phi(\mathbf{h}^{l-1}, \xi, \pi^l; \mathbf{W}^l) = \pi^l \odot \mathbf{h}^{l-1} + (1 - \pi^l) \odot \tilde{\mathbf{h}}^l \quad (22)$$

$$\mathbf{h}^l = \phi(\mathbf{h}^{l-1}, \xi, \pi^l; \mathbf{W}^l). \quad (23)$$

To decide which path to take, for each unit in the network, a binary stochastic decision is taken by drawing from a Bernoulli distribution with probability dependent on the decaying value of p^l :

$$\pi^l \sim \text{Bernoulli}(p^l) \quad (24)$$

If the number of hidden units of layer $l-1$ and layer $l+1$ is not the same, we can either zero-pad layer $l-1$ before feeding it into the next layer or apply a linear projection to obtain the right dimensionality.

For $p^l = 1$, the layer computes the identity function leading to a **convex** objective. If $p^l = 0$ the layer computes the original non-linear transformation unfolding the full capacity of the model. We call the connections that are introduced this way as *unitwise stochastic skip connections*(USSC).

In DropIn layers (Smith et al., 2016) a binary random vector is sampled from a Bernoulli distribution to decide whether to introduce skip a connection from the layer below $l-1$ for each layer l and they use this as a regularization. As opposed to the USSC, DropIn layers do not necessarily do not necessarily achieve a convex objective function as the DropIn ratio (p^l) increases.

The pseudo-code for the mollified activations is reported in Algorithm 1.

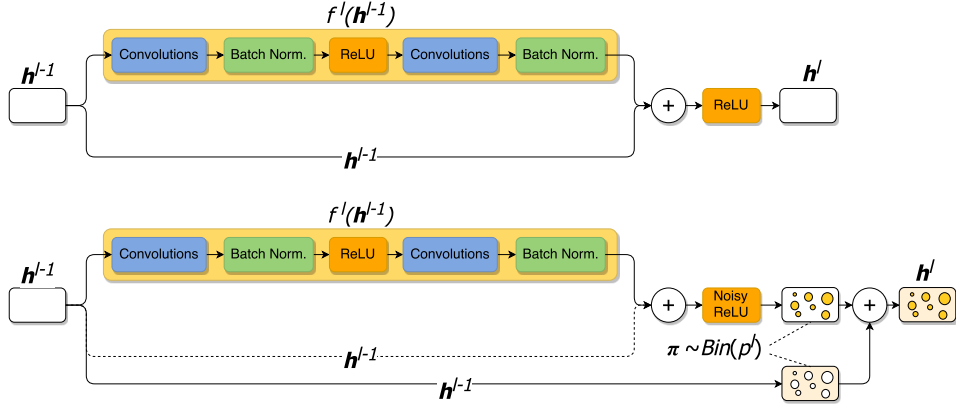


Figure 2: **Top:** Stochastic depth. **Bottom:** mollifying network. The dashed line represents the optional residual connection. In the top path, the input is processed with a convolutional block followed by a noisy activation function, while in the bottom path the original activation of the layer $l - 1$ is propagated untouched. For each unit, one of the two paths is picked according to a binary stochastic decision π .

Algorithm 1 Activation of a unit i at layer l .

- 1: $x_i \leftarrow \mathbf{w}_i^\top \mathbf{h}^{l-1} + b_i$ ▷ an affine transformation of \mathbf{h}^{l-1}
 - 2: $\Delta_i \leftarrow \mathbf{u}(x_i) - \mathbf{f}(x_i)$ ▷ Δ_i is a measure of a saturation of a unit
 - 3: $\sigma(x_i) \leftarrow (\text{sigmoid}(a_i \Delta_i) - 0.5)^2$ ▷ std of the injected noise depends on Δ_i
 - 4: $\xi_i \sim \mathcal{N}(0, 1)$ ▷ sampling the noise from a basic Normal distribution
 - 5: $s_i \leftarrow p^l c \sigma(x_i) |\xi_i|$ ▷ Half-Normal noise controlled by $\sigma(x_i)$, const. c and prob-ty p^l
 - 6: $\psi(x_i, \xi_i) \leftarrow \text{sgn}(\mathbf{u}^*(x_i)) \min(|\mathbf{u}^*(x_i)|, |\mathbf{f}^*(x_i) + \text{sgn}(\mathbf{u}^*(x_i))|s_i|) + \mathbf{u}(0)$ ▷ noisy activation
 - 7: $\pi_i^l \sim \text{Bernoulli}(p^l)$ ▷ p^l controls the variance of the noise AND the prob of skipping a unit
 - 8: $\tilde{h}_i^l = \psi(x_i, \xi_i)$ ▷ \tilde{h}_i^l is a noisy activation candidate
 - 9: $\phi(\mathbf{h}^{l-1}, \xi_i, \pi_i^l; \mathbf{w}_i) = \pi_i^l h_i^{l-1} + (1 - \pi_i^l) \tilde{h}_i^l$ ▷ make a HARD decision between h_i^{l-1} and \tilde{h}_i^l
-

5 LINEARIZING THE NETWORK

In Section 2, we show that convolving the objective function with a particular kernel can be approximated by adding noise to the activation function. This method may suffer from excessive random exploration when the noise is very large.

We address this issue by bounding the element-wise activation function $\mathbf{f}(\cdot)$ with its linear approximation when the variance of the noise is very large, after centering it at the origin. The resulting function $\mathbf{f}^*(\cdot)$ is bounded and centered around the origin.

Note that centering the sigmoid or hard-sigmoid will make them symmetric with respect to the origin. With a proper choice of the standard deviation $\sigma(\mathbf{h})$, the noisy activation function becomes a linear function of the input when p is large, as illustrated by Figure 10.

Let $\mathbf{u}^*(x) = \mathbf{u}(x) - \mathbf{u}(0)$, where $\mathbf{u}(0)$ is the offset of the function from the origin, and x_i the i -th dimension of an affine transformation of the output of the previous layer \mathbf{h}^{l-1} : $x_i = \mathbf{w}_i^\top \mathbf{h}^{l-1} + b_i$. Then:

$$\psi(x_i, \xi_i; \mathbf{w}_i) = \text{sgn}(\mathbf{u}^*(x_i)) \min(|\mathbf{u}^*(x_i)|, |\mathbf{f}^*(x_i) + \text{sgn}(\mathbf{u}^*(x_i))|s_i|) + \mathbf{u}(0) \quad (25)$$

The noise is sampled from a Normal distribution with mean 0 and whose standard deviation depends on c :

$$s_i \sim \mathcal{N}(0, p c \sigma(x_i))$$

5.1 LINEARIZING RELU ACTIVATION FUNCTION

We have a simpler form of the equations to linearize ReLU (Nair & Hinton, 2010) activation function when $p^l \rightarrow \infty$. Instead of the complicated Eqn. 23. We can use a simpler equation as in Eqn. 26 to

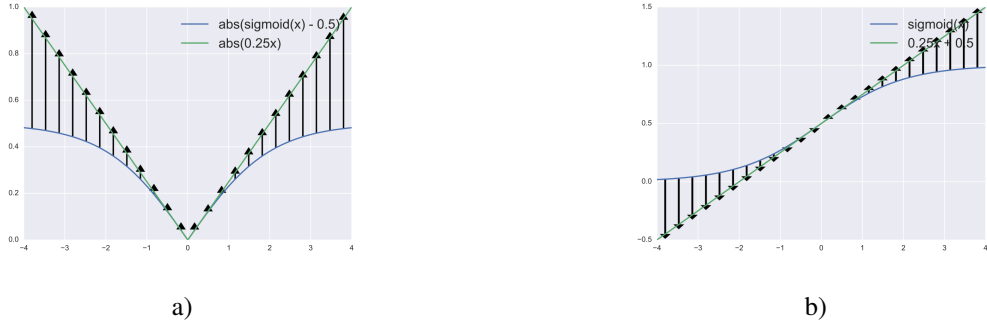


Figure 3: The figures show how to evolve the model to make it closer to a linear network. Arrows denote the direction of the noise pushing the activation function towards the linear function. **a)** The quasi-convex envelope established by a $|\text{sigmoid}(\cdot)|$ around $|0.25x|$. **b)** A depiction of how the noise pushes the sigmoid to become a linear function.

achieve the linearization of the activation function when we have a very large noise in the activation function:

$$s_i = \text{minimum}(|x_i|, p\sigma(x_i)|\xi|) \quad (26)$$

$$\psi(x_i, \xi_i, \mathbf{w}_i) = f(x_i) - s_i \quad (27)$$

6 MOLLIFYING LSTMS AND GRUS

In a similar vein it is possible to smooth the objective functions of LSTM and GRU networks by starting the optimization procedure with a simpler objective function such as optimizing a word2vec, BoW-LM or CRF objective function at the beginning of training and gradually increasing the difficulty of the optimization by increasing the capacity of the network.

For GRUs we set the update gate to $\frac{1}{t}$ – where t is the time-step index – and reset the gate to 1 if the noise is very large, using Algorithm 1. Similarly for LSTMs, we can set the output gate to 1 and input gate to $\frac{1}{t}$ and forget gate to $1 - \frac{1}{t}$ when the noise is very large. The output gate is 1 or close to 1 when the noise is very large. This way the LSTM will behave like a BOW model. In order to achieve this behavior, the activations $\psi(x_t, \xi_i)$ of the gates can be formulated as:

$$\psi(x_t^l, \xi) = f(x_t^l + p^l \sigma(x)|\xi|)$$

By using a particular formulation of $\sigma(x)$ that constraints it to be in expectation over ξ when $p^l = 1$, we can obtain a function for $\gamma \in \mathbb{R}$ within the range of $f(\cdot)$ that is discrete in expectation, but still per sample differentiable:

$$\sigma(x_t^l) = \frac{f^{-1}(\gamma) - x_t^l}{E_\xi[|\xi|]} \quad (28)$$

We provide the derivation of Eqn. 28 in Appendix B. The gradient of the Eqn. 28 will be a Monte-Carlo approximation to the gradient of $f(x_t^l)$.

7 ANNEALING SCHEDULES FOR p

We used a different schedule for each layer of the network, such that the noise in the lower layers will anneal faster. This is similar to the linearly decaying probability of layers in Huang et al. (2016b).

Exponential Decay In our experiments, we focused on using an annealing schedule similar to the inverse sigmoid rule in Bengio et al. (2015) with p_t^l ,

$$p_t^l = 1 - e^{-\frac{kv_t^l}{\tau L}} \quad (29)$$

with hyper-parameter $k \geq 0$ at t^{th} update for the l^{th} layer, where L is the number of layers of the model. We stop annealing when the expected depth $p_t = \sum_{i=1}^L p_t^i$ reaches some threshold δ . In our experiment we set v_t to be a moving average of the loss³ of the network, but for some of our experiments that resulted unstable behaviors in the training and thus we have to fix v_t to 1. An advantage of using running average of loss for the v_t is because the behavior of the loss/optimization can directly influence the annealing behavior of the network. Because we will have:

$$\lim_{v_t \rightarrow \infty} p_t^l = 1 \text{ and, } \lim_{v_t \rightarrow 0} p_t^l = 0. \quad (30)$$

This has a desirable property: when the training-loss is high, the noise injected into the system will be large as well. As a result, the model is encouraged to do more exploration, while if the model converges the noise injected into the system by the mollification procedure will be zero.

Square-root Decay The square root decaying of the learning rate is very popular. In our experiments we have tried to decay the p_t^l according to $\min(p_{min}, 1 - \sqrt{\frac{t}{N_{epochs}}})$ where t is the index of the iteration/epoch over the data and N_{epochs} corresponds to the total number of iterations that the model will do on the dataset. p_{min} is hyperparameter to define the lower bound on the p_t^l .

Linear Decay We have also experimented with linearly decaying the p_t^l via $\min(p_{min}, 1 - \frac{t}{N_{epochs}})$. We have compared the plot of different annealing methods described in this paper as in Figure 4.

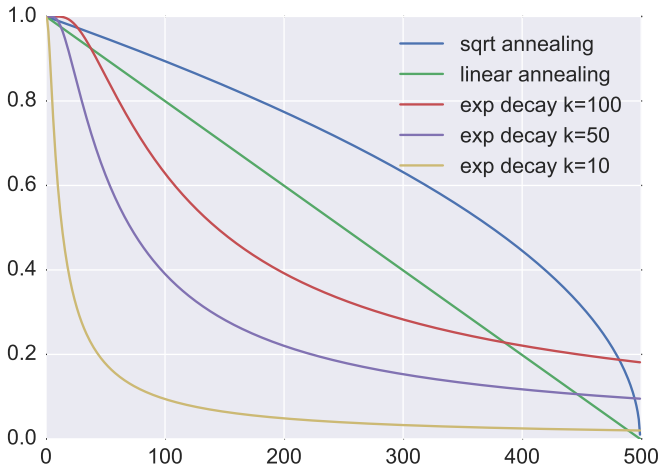


Figure 4: We compare different annealing schedules with respect to the time in y-axis(iterations).

Furthermore, in our experiments we observe that training with noisy mollifiers can potentially be helpful for the generalization. This can be due to the noise induced to the backpropagation through the noisy mollification, that makes SGD more likely to converge to a flatter-minima (Hochreiter & Schmidhuber, 1997b) because the noise will help it escape from sharper local minima.

8 EXPERIMENTS

In this section we mainly focus on training of difficult to optimize models, in particular deep MLPs with sigmoid or tanh activation functions. The details of the experimental procedure is provided in Appendix C.

8.1 EXPLORATORY STUDY ON MNIST

We train a thin deep neural network on MNIST (LeCun & Cortes, 1998) dataset with 72 hidden layers and 100 hidden units. We train our model with Adam(Kingma & Ba, 2014) optimizer and fix the

³Depending on whether the model overfits or not, this can be a moving average of training or validation loss.

learning rate of all the models to $3e-4$. We have used the same learning-rate for all the models in order to factor out the possibility that a model converges faster, due to the fact of using a larger learning rate.

Firstly, in Figure 5, we investigate the effect of using different annealing schedules. Exponential decay converges faster compared to linear decay and square root decay of p . We find it to be very unstable to train our model with linear and square root decay in particular for large c values. Thus we have to use smaller c value (20 instead of 100) to be able to train the model with causing it to diverge.

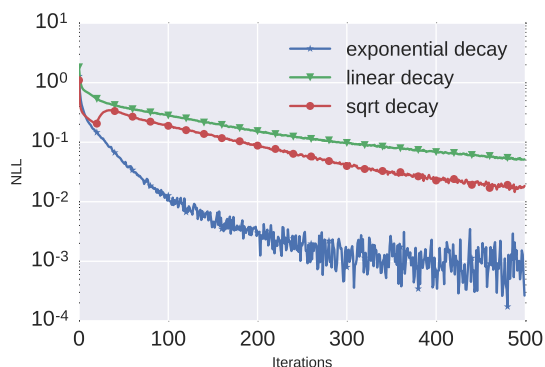


Figure 5: We compare the training performance of different types of annealing methods used with mollification procedure to anneal the parameter of the mollification p . Decaying the p exponentially achieves both better training and validation performance.

In Figure 6, we show the effect of using the noisy training procedure that we have introduced by sampling a mask from Bernoulli and Gaussian distributions versus using the deterministic approximation of this noisy procedure which we also use for the test time but during the training as well.

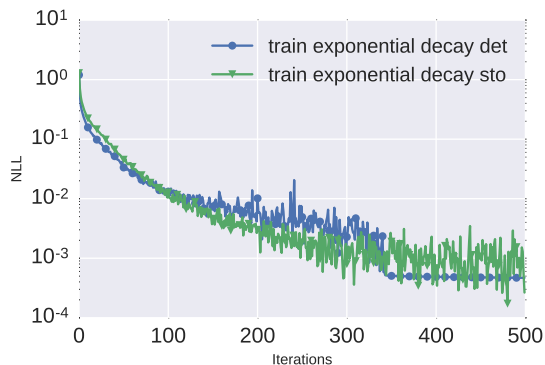


Figure 6: We show the learning curves of the model where we don't inject the noise during the training and instead use the deterministic approximation for the mollification during the training as well. The difference in terms of speed of learning is very small.

In Figure 7, we compare the results obtained for the model using mollification obtained with or without batch normalization and feed-forward residual networks. The mollified model performs very closely to the MLP trained with residual connections and the batchnorm. However, using residual connections and batch-norm does not seem to improve the results.

We have tried to run experiment with the Monte-Carlo approximation of the mollification which is derived in Appendix A, however when we start with large noise and anneal the noise during the training, the model was very unstable and training was diverging. If we start with small noise and anneal the magnitude of the noise during the training, we could not observe any effect of it on the training.

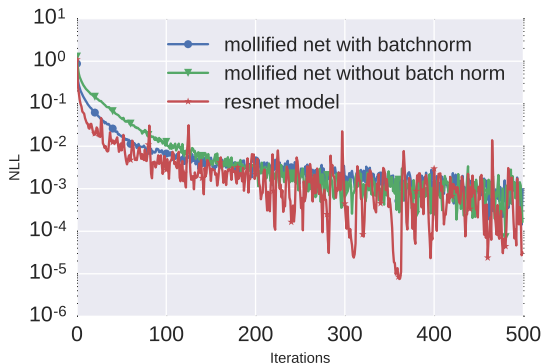


Figure 7: We investigate the effect of using batch norm and residual connection for mollification and compare against to the network with residual connections and batch-norm. The effect of batch norm on this task for mollification seems to be very small and training convergence performance of the all the approaches are very close to each other.

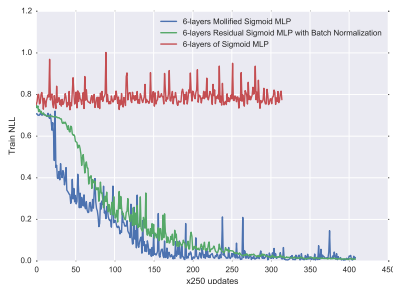


Figure 8: The learning curves of a 6-layers MLP with sigmoid activation function on 40 bit parity task.

	Test Accuracy
Stochastic Depth	93.25
Mollified Convnet	92.45
ResNet	91.78

Table 1: CIFAR10 deep convolutional neural network.

8.2 DEEP MLP EXPERIMENTS

Deep Parity Experiments Training neural networks on a high-dimensional parity problem can be challenging (Graves, 2016; Kalchbrenner et al., 2015). We experiment on forty dimensional (bits) parity problem with 6-layer MLP using sigmoid activation function. All the models are initialized with Glorot initialization Glorot et al. (2011) and trained with SGD with momentum. We compare an MLP with residual connections using batch normalization and a mollified network with sigmoid activation function. As can be seen in Figure 8, the mollified network converges faster.

Deep Pentomino Pentomino is a toy-image dataset where each image has 3 Pentomino blocks. The task is to predict whether if there is a different shape in the image or not (Gülçehre & Bengio, 2013). The best reported result on this task with MLPs is 68.15% accuracy (Gulçehre et al., 2014). The same model as ours trained without noisy activation function and vanilla residual connections scored 69.5% accuracy, while our mollified version scored 75.15% accuracy after 100 epochs of training on the 80k dataset.

CIFAR10 We experimented with deep convolutional neural networks of 110-layers with residual blocks and residual connections comparing our model against ResNet and Stochastic depth. We adapted the hyperparameters of the Stochastic depth network from Huang et al. (2016a) and we used the same hyperparameters for our algorithm. We report the training and validation curves of the three models in Figure 10 and the best test accuracy obtained early stopping on validation accuracy over 500 epochs in Table 1. Our model achieves better generalization than ResNet. Stochastic depth achieves better generalization, but it might be possible to combine both and obtain better results.

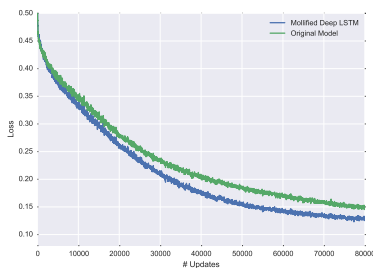


Figure 9: The training curve of a bidirectional-RNN that predicts the embedding corresponding to a sequence of characters.

	Test PPL
LSTM	119.4
Mollified LSTM	115.7

Table 2: 3-layered LSTM network on word-level language modeling for PTB.

9 LSTM EXPERIMENTS

Predicting the Character Embeddings from Characters Learning the mapping from sequences of characters to the word-embeddings is a difficult problem. Thus one needs to use a highly non-linear function. We trained a word2vec model on Wikipedia with embeddings of size 500 (Mikolov et al., 2014) with a vocabulary of size 374557.

LSTM Language Modeling We evaluate our model on LSTM language modeling. Our baseline model is a 3-layer stacked LSTM without any regularization. We observed that mollified model converges faster and achieves better results. We provide the results for PTB language modeling in Table 2.

10 CONCLUSION

We propose a novel method for training neural networks inspired by an idea of continuation, smoothing techniques and recent advances in non-convex optimization algorithms. The method makes learning easier by starting from a simpler model, solving a well-behaved problem, and gradually transitioning to a more complicated setting. We show improvements on very deep models, difficult to optimize tasks and compare with powerful techniques such as batch-normalization and residual connections. We also show that the mollification procedure improves the generalization performance of the model on two tasks.

Our future work includes testing this method on large-scale language tasks that require long training time, e.g., machine translation and language modeling. Moreover, (Kaiser & Sutskever, 2015) observed that the training of Neural-GPU model can be improved significantly by using gradient noise which can be related to the smoothing of the loss surface, it would be interesting to try mollification on this model to see if the training of Neural GPU can be made easier by using mollification procedure.

ACKNOWLEDGEMENTS

We thank Nicholas Ballas and Misha Denil for the valuable discussions and their feedback. We would like to also thank the developers of Theano⁴, for developing such a powerful tool for scientific computing Theano Development Team (2016). We acknowledge the support of the following organizations for research funding and computing support: NSERC, Samsung, Calcul Québec, Compute Canada, the Canada Research Chairs and CIFAR.

REFERENCES

- E. L. Allgower and K. Georg. *Numerical Continuation Methods. An Introduction*. Springer-Verlag, 1980.
- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

⁴<http://deeplearning.net/software/theano/>

- Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pp. 1171–1179, 2015.
- Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48. ACM, 2009.
- Léon Bottou. Online algorithms and stochastic approximations. In David Saad (ed.), *Online Learning in Neural Networks*. Cambridge University Press, Cambridge, UK, 1998.
- Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surface of multilayer networks, 2014.
- Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS’2014*, 2014.
- Lawrence C Evans. Partial differential equations. *Graduate Studies in Mathematics*, 19:251–258, 1998.
- Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *AISTATS*, pp. 315–323, 2011.
- Alex Graves. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems*, pp. 2348–2356, 2011.
- Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.
- Çağlar Gülçehre and Yoshua Bengio. Knowledge matters: Importance of prior information for optimization. *arXiv preprint arXiv:1301.4083*, 2013.
- Caglar Gulcehre, Kyunghyun Cho, Razvan Pascanu, and Yoshua Bengio. Learned-norm pooling for deep feedforward and recurrent neural networks. In *Machine Learning and Knowledge Discovery in Databases*, pp. 530–546. Springer, 2014.
- Caglar Gulcehre, Marcin Moczulski, Misha Denil, and Yoshua Bengio. Noisy activation functions. 2016.
- Elad Hazan, Kfir Y Levy, and Shai Shalev-Shwartz. On graduated optimization for stochastic non-convex problems. *arXiv preprint arXiv:1503.03712*, 2015.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.
- Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.
- Geoffrey E Hinton and Drew van Camp. Keeping neural networks simple. In *ICANN’93*, pp. 11–18. Springer, 1993.
- S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997a.
- Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997b.
- Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Weinberger. Deep networks with stochastic depth. *arXiv preprint arXiv:1603.09382*, 2016a.
- Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. Deep networks with stochastic depth. *CoRR*, abs/1603.09382, 2016b. URL <http://arxiv.org/abs/1603.09382>.

- Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL <http://arxiv.org/abs/1502.03167>.
- Lukasz Kaiser and Ilya Sutskever. Neural gpu learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.
- Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.
- Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- S. Kirkpatrick, C. D. Gelatt Jr., , and M. P. Vecchi. Optimization by simulated annealing. 220: 671–680, 1983.
- Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, December 1989. ISSN 0899-7667. doi: 10.1162/neco.1989.1.4.541. URL <http://dx.doi.org/10.1162/neco.1989.1.4.541>.
- Yann LeCun and Corinna Cortes. The mnist database of handwritten digits, 1998.
- Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. word2vec, 2014.
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, and Daan Wierstra. Playing atari with deep reinforcement learning. Technical report, arXiv:1312.5602, 2013.
- Hossein Mobahi. Training recurrent neural networks by diffusion. *arXiv preprint arXiv:1601.04114*, 2016.
- Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814, 2010.
- Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *CoRR*, abs/1511.06807, 2015a. URL <http://arxiv.org/abs/1511.06807>.
- Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015b.
- Andrew Y Ng. *Shaping and policy search in reinforcement learning*. PhD thesis, University of California, Berkeley, 2003.
- Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pp. 278–287, 1999.
- David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- Leslie N Smith, Emily M Hand, and Timothy Doster. Gradual dropin of layers to train very deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4763–4771, 2016.
- Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in Neural Information Processing Systems*, pp. 2368–2376, 2015.
- Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.
- Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. Technical report, Google, 2014.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL <http://arxiv.org/abs/1605.02688>.

Francesco Visin, Kyle Kastner, Aaron Courville, Yoshua Bengio, Matteo Matteucci, and Kyunghyun Cho. Reseg: A recurrent neural network for object segmentation. *arXiv preprint arXiv:1511.07053*, 2015.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

Appendix

A MONTE-CARLO ESTIMATE OF MOLLIFICATION

$$\begin{aligned} \mathcal{L}_K(\boldsymbol{\theta}) &= (\mathcal{L} * K)(\boldsymbol{\theta}) = \int_C \mathcal{L}(\boldsymbol{\theta} - \xi)K(\xi)d\xi \text{ which can be estimated by a Monte Carlo:} \\ &\approx \frac{1}{N} \sum_{i=1}^N \mathcal{L}(\boldsymbol{\theta} - \xi^{(i)}), \text{ where } \xi^{(i)} \text{ is a realization of the noise random variable } \xi \\ &\text{yielding } \frac{\partial \mathcal{L}_K(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \\ &\approx \frac{1}{N} \sum_{i=1}^N \frac{\partial \mathcal{L}(\boldsymbol{\theta} - \xi^{(i)})}{\partial \boldsymbol{\theta}}. \end{aligned} \tag{31}$$

Therefore introducing additive noise to the input of $\mathcal{L}(\boldsymbol{\theta})$ is equivalent to mollification.

B DERIVATION OF THE NOISY ACTIVATIONS FOR THE GATING

Assume that $z_t^l = x_t^l + p_t^l \sigma(x) |\xi_t^l|$ and $E_\xi[\psi(x_t^l, \xi)] = t$. Thus for all z_t^l ,

$$E_\xi[\psi(x_t^l, \xi_t^l)] = E_\xi[f(z_t^l)], \tag{32}$$

$$t = E_\xi[f(z_t^l)], \text{ assuming } f(\cdot) \text{ behaves similar to a linear function:} \tag{33}$$

$$E_\xi[f(z_t^l)] \approx f(E_\xi[z_t^l]) \text{ since we use hard-sigmoid for } f(\cdot) \text{ this will hold.} \tag{34}$$

$$f^{-1}(t) \approx E_\xi[z_t^l] \tag{35}$$

$$\tag{36}$$

As in Eqn. 32, we can write the expectation of this equation as:

$$f^{-1}(t) \approx x_t^l + p_t^l \sigma(x) E_\xi[\xi_t^l]$$

Corollary, the value that $\sigma(x_t^l)$ should take in expectation for $p_t^l = 1$ would be:

$$\sigma(x_t^l) \approx \frac{f^{-1}(t) - x_t^l}{E_\xi[\xi_t^l]}$$

In our experiments for $f(\cdot)$ we used the hard-sigmoid activation function. We used the following piecewise activation function in order to use it as $f^{-1}(x) = 4(x - 0.5)$. During inference we use the expected value of random variables π and ξ .

C EXPERIMENTAL DETAILS

C.1 MNIST

The weights of the models are initialized with Glorot & Bengio initialization Glorot et al. (2011). We use the learning rate of $4e - 4$ along with RMSProp. We initialize a_i parameters of mollified activation function by sampling it from a uniform distribution, $U[-2, 2]$. We used 100 hidden units at each layer with a minibatches of size 500.

C.2 PENTOMINO

We train a 6-layer MLP with sigmoid activation function using SGD and momentum. We used 200 units per layer with sigmoid activation functions. We use a learning rate of $1e-3$.

C.3 CIFAR10

We use the same model with the same hyperparameters for both ResNet, mollified network and the stochastic depth. We borrowed the hyperparameters of the model from Huang et al. (2016a). Our mollified convnet model has residual connections coming from its layer below.

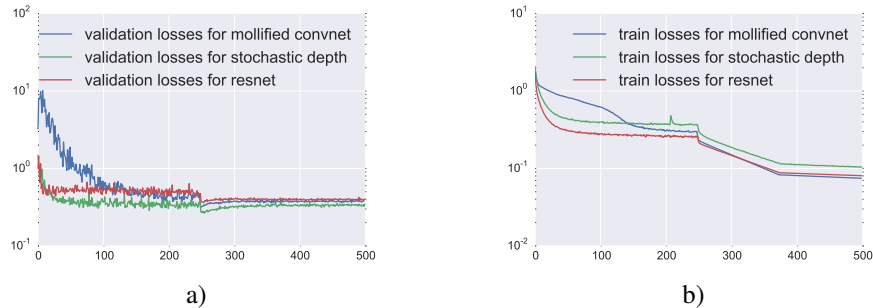


Figure 10: Training and validation losses over 500 epochs of a mollified convolutional network composed by 110-layers. We compare against ResNet and Stochastic depth.

C.4 PARITY

The n -dimensional parity task is the task to figure out whether the sum of n -bits in a binary vector is even or odd. We use SGD with Nesterov momentum and initialize the weight matrices by using Glorot&Bengio initialization (Glorot et al. (2011)). For all models, we use the learning rate of $1e-3$ and momentum of 0.92. a_i is the parameters of mollified activation function are initialized by sampling from uniform distribution, $U[-2, 2]$.

C.5 LSTM LANGUAGE MODELING

We trained 2-layered LSTM language models on PTB word-level. We used the models with the same hyperparameters as in Zaremba & Sutskever (2014). We used the same hyperparameters for both the mollified LSTM language model and the LSTM. We use hard-sigmoid activation function for both the LSTM and mollified LSTM language model. We use hard-sigmoid activation function for the gates of the LSTM.

C.6 PREDICTING THE CHARACTER EMBEDDINGS FROM CHARACTERS

We use $10k$ of these words as a validation and another $10k$ word embeddings as test set. We train a bidirectional-LSTM on top of each sequence of characters for each word and on top of the representation of bidirectional LSTM, we use a 5-layered tanh-MLP to predict the word-embedding.

We train our models using RMSProp and momentum with learning rate of $6e-4$ and momentum 0.92. The size of the minibatches, we used is 64. As seen in Figure 9, mollified LSTM network converges faster.