Learning to Learn Second-Order Back-Propagation for CNNs Using LSTMs

Anirban Roy
SRI International
Menlo Park, USA
anirban.roy@sri.com

Sinisa Todorovic
Oregon State University
Corvallis, USA
sinisa@eecs.oregonstate.edu

***Abstract*—Convolutional neural networks (CNNs) typically suffer from slow convergence rates in training, which limits their wider application. This paper presents a new CNN learning approach, based on second-order methods, aimed at improving: a) Convergence rates of existing gradient-based methods, and b) Robustness to the choice of learning hyper-parameters (e.g., learning rate). We derive an efficient back-propagation algorithm for simultaneously computing both gradients and second derivatives of the CNN's learning objective. These are then input to a Long Short Term Memory (LSTM) to predict optimal updates of CNN parameters in each learning iteration. Both meta-learning of the LSTM and learning of the CNN are conducted jointly. Evaluation on image classification demonstrates that our second-order back-propagation has faster convergences rates than standard gradient-based learning for the same CNN, and that it converges to better optima leading to better performance under a budgeted time for learning. We also show that an LSTM learned to learn a small CNN network can be readily used for learning a larger network.**

I. Introduction

Convolutional neural networks (CNNs) have led to great advances in science and technology. CNNs are typically learned using stochastic gradient descent (SGD), or its more sophisticated variants, leveraging back-propagation [1]. However, these methods exhibit slow convergence rates for the highly nonconvex learning objectives of CNNs [2], [3], and their convergence is sensitive to the choice of hyper-parameters.

This paper presents a new learning approach to estimating parameters of a CNN, aimed at improving: (1) Convergence rates of existing gradient-based learning methods, and (2) Robustness to the choice of learning hyper-parameters. Our objectives (1) and (2) are constrained such that performance of the CNN should not be compromised relative to the gradient-based training of the same CNN.

Prior work considers: (i) Gradient values of the previous iteration as a momentum for updating CNN parameters in the next iteration [2], [3]; (ii) Curvature of the learning objective function as a second-order cue to automatically adjust the learning rate of SGD [4], [5], [6], [7]; and (iii) Efficient approximations of the Hessian for conducting quasi-Newton optimization [6], [7]. However, these approaches typically use hand-designed heuristics (e.g., heuristic approximation of the Hessian) tailored to a specific task that the CNN is trained for, e.g., image classification or object detection. Also, most second-order methods, which approximate the Hessian for efficiency, cannot handle large deep networks with millions of parameters, with a few exceptions [4], [5] that use only the main diagonal of the Hessian matrix.

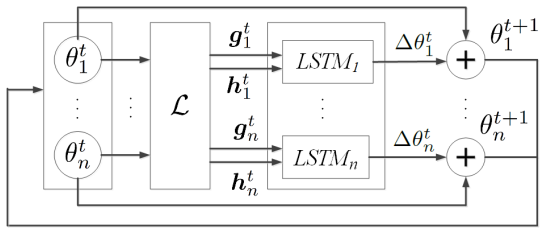


Fig. 1: Overview of our approach to learn a CNN with parameters $\boldsymbol{\theta} = \{\theta_n\}$ and objective $\mathcal{L}$. At tth iteration, we use LSTMs to predict optimal updates $\Delta\theta_n^t$ for each parameter θ_n, given the respective gradient g_n^t and second derivative h_n^t of $\mathcal{L}$ as input.

Motivation. In this paper, we resort to the framework of second-order methods. We are motivated by recent findings that deep learning objectives are typically characterized by numerous "plateau" regions with near-zero gradient values [7], [8], [9]. Also, as the number of hidden units in a neural network increases, any given critical point of a deep-learning objective function is more likely to be a saddle point than a local minimum [7], [8]. Consequently, first-order gradient-based learning is bound to have a slow convergence rate due to frequent "passes" through the "plateau" regions. For speeding-up the convergence, it seems critical to account for the Hessian, and in this way avoid the "plateau" regions. However, the main challenge is that computing the Hessian is prohibitively expensive both computationally and memory-wise.

Our Approach. To address this issue, we derive an efficient back-propagation, which simultaneously (thus efficiently) computes both gradients and second derivatives of the CNN's learning objective. At tth iteration, these gradients $\boldsymbol{g}^t$ and second derivatives $\boldsymbol{h}^t$ are then used to estimate optimal updates $\Delta\boldsymbol{\theta}^t$ of CNN parameters as

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \Delta\boldsymbol{\theta}^t(\boldsymbol{g}^t, \boldsymbol{h}^t). \tag{1}$$

Our key contribution is to employ a Long Short Term Memory (LSTM) [10] to *learn* to estimate $\Delta\boldsymbol{\theta}^t$, as shown in Fig. 1. An LSTM is a recurrent neural network with memory. Our LSTM takes gradients $\boldsymbol{g}^t$ and second derivatives $\boldsymbol{h}^t$ as input, and uses a memory of previous learning iterations to

predict $\Delta\boldsymbol{\theta}^t$ for the next iteration. As the LSTM is used in learning a CNN, then learning the LSTM represents meta-learning [11], [12], [13], [19]. Importantly, in our approach, both the meta-LSTM and the CNN are learned jointly.

LSTMs are suitable for our purposes for a number of reasons. First, they could capture long- and short-range dependences of sequential updates of CNN parameters. In this way, LSTMs facilitate avoiding "plateau" regions of the learning objective. Second, our meta-learning of the LSTM directly overcomes the issue of choosing optimal hyper-parameters (e.g., learning rate), as they are meta-learned.

Closely Related Previous Work on the second-order optimization in deep learning [14], [15], [5], [7] typically uses the additional curvature cues to automatically adjust the learning rate, and thus better navigate through the plateaus and saddle points. Due to the size of the network, approximate Hessian-free or Quasi-Newton approaches [5], [7] or a diagonal Hessian matrix [4], [5] are usually adopted in practice. As there is no efficient way of computing even the main diagonal of the inverse Hessian [14], we consider the approximate computation of the diagonal, following [4]. While majority of meta-learning approaches rely only on gradient based cues as inputs to their respective meta-leaners [11], [12], [13], [19], in contrast, we additionally consider second derivatives which results in our faster convergence.

Evaluation. We evaluate our approach on the task of image classification. First, we learn parameters of a standard CNN for image classification on the MNIST [16], CIFAR-10 [17] and ImageNet [18] datasets. In comparison with the standard hand-designed optimization methods, our approach achieves faster convergence rates and better performance with minor increase in computation. Second, following [19], we evaluate the generalization capability of our learning. Specifically, the meta-LSTM is learned to optimize parameters of a small network, and then the same LSTM is used for learning a larger network.

II. A Review of CNN Learning

This section gives an overview of learning CNN parameters. Let $\mathcal{L}(\boldsymbol{\theta})$ denote the CNN's learning objective (e.g., loss function) over the parameter space $\boldsymbol{\theta} \in \Theta$. The goal of CNN learning is to estimate optimal parameters $\boldsymbol{\theta}^*$ by minimizing the objective, $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\Theta} \mathcal{L}(\boldsymbol{\theta})$. This optimization problem can be iteratively solved using gradient descent as $\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \alpha^t \boldsymbol{g}^t$, where $\boldsymbol{g}^t = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}^t}$ is the gradient of $\mathcal{L}$, and α^t is the learning rate at iteration t.

Second-order methods consider a local approximation of $\mathcal{L}$ using the second-order Taylor expansion, $\mathcal{L}(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) \approx \mathcal{L}(\boldsymbol{\theta}) + \boldsymbol{g}^\top \Delta\boldsymbol{\theta} + \frac{1}{2}\Delta\boldsymbol{\theta}^\top \mathbf{H} \Delta\boldsymbol{\theta}$, where $\boldsymbol{g}$ and $\mathbf{H}$ are the gradient and Hessian of $\mathcal{L}$ at $\boldsymbol{\theta}$. Following the Newton's method, this gives the following parameter updates:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \alpha^t (\mathbf{H}^t)^{-1} \boldsymbol{g}^t. \tag{2}$$

As computing the inverse of the Hessian in (2) is prohibitively expensive for large CNNs, following [4], [7], we consider only the second derivatives of $\mathcal{L}$ that are on the main diagonal of the Hessian:

$$\frac{\partial^2 \mathcal{L}}{\partial \theta^l_{j'i'} \partial \theta^l_{ji}} = \begin{cases} \dfrac{\partial^2 \mathcal{L}}{\partial {\theta^l_{ji}}^2} & \text{if } \; i = i' \text{ and } j = j' \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where θ^l_{ji} is the weight between neuron j at layer l and neuron i at layer $l-1$. We denote the resulting approximate Hessian matrix $\hat{\mathbf{H}}$ as $\mathbf{H} \approx \hat{\mathbf{H}} = \text{diag}(\boldsymbol{h})$, where $\boldsymbol{h} : \boldsymbol{h}^l_{ji} = \frac{\partial^2 \mathcal{L}}{\partial {\theta^l_{ji}}^2}$ is the main diagonal of $\mathbf{H}$. We will use $\hat{\mathbf{H}}$ in one of our quasi-Newton baselines to compare with our meta-LSTM based method. This quasi-Newton baseline specifies the following updates:

$$\theta^l_{ji} = \theta^l_{ji} - \alpha \frac{\partial \mathcal{L} / \partial \theta^l_{ji}}{\partial^2 \mathcal{L} / \partial {\theta^l_{ji}}^2}. \tag{4}$$

From (1) and (4), it follows that our meta-LSTM serves to learn to optimally compute the ratio $-\alpha \frac{\partial \mathcal{L}/\partial \theta^l_{ji}}{\partial^2 \mathcal{L}/\partial {\theta^l_{ji}}^2}$ in each iteration.

Note that $\boldsymbol{h}$ can be iteratively estimated [6], [5]; however, this increases complexity and also makes it harder to incorporate into the popular back-propagation based learning framework.

Instead, we derive a second-order back-propagation algorithm, aimed at approximately computing the elements of $\boldsymbol{h}$ using only one forward and one backward pass through the CNN. Note that our complexity is of the same order as the standard gradient computation in back-propagation.

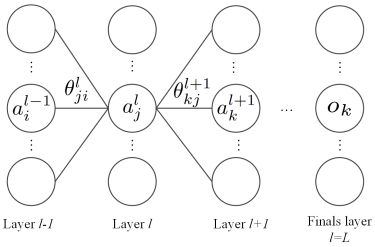


Fig. 2: A multi-layer CNN. i, j, k are indices or neurons on layers $l-1, l, l+1$ respectively. o_k represents neurons at the output layer.

III. Second-Order Back-propagation

In a CNN with L layers, illustrated in Fig. 2, neurons at consecutive layers are connected such that neuron j at layer l and neuron i at layer $l-1$ are connected with an edge with weight θ^l_{ji}. Let z^l_j denote the total input to neuron j at layer l, $z^l_j = \sum_i \theta^l_{ji} a^{l-1}_i$ (we drop the bias terms for simplicity), where $a^l_j = \sigma(z^l_j)$ is the corresponding non-liner activation, and $\sigma(\cdot)$ is a non-linear function (e.g., sigmoid, or ReLU). Using this notation, below, we define the following partial derivatives.

$$\frac{\partial a^l_j}{\partial z^l_j} = (\sigma^l_j)', \frac{\partial^2 a^l_j}{\partial {z^l_j}^2} = (\sigma^l_j)'', \frac{\partial z^l_j}{\partial \theta^l_{ji}} = a^{l-1}_i, \frac{\partial a^l_j}{\partial a^{l-1}_i} = (\sigma^l_j)' \theta^l_{ji}, \tag{5}$$

where for example $(\sigma^l_j)' = a^l_j(1 - a^l_j)$ and $(\sigma^l_j)'' = a^l_j(1 - a^l_j)(1 - 2a^l_j)$ for the sigmoid function. In the following, we use the shorthand σ' and σ'' to denote $(\sigma^l_j)'$ and $(\sigma^l_j)''$, respectively, where the neuron's layer and index are clear from the context.

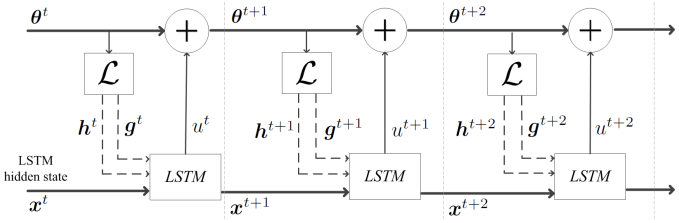


Fig. 3: Computational graph of our approach. In each step t, we compute the gradient $\boldsymbol{g}^t$, second derivatives $\boldsymbol{h}^t$ of the objective function $\mathcal{L}$ corresponding to the current parameters $\boldsymbol{\theta}^t$. These are input to the LSTM for computing the update u^t. The LSTM is parameterized by the parameters $\boldsymbol{\phi}$ and its hidden states $\boldsymbol{x}^t$. Both $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are learned jointly using back-propagation through time. All LSTMs have shared parameters, but separate hidden states.

A review of gradient based updates. The standard gradient based back-propagation at any particular step is defined as

$$\begin{aligned}\theta^l_{ji} &= \theta^l_{ji} - \alpha\frac{\partial\mathcal{L}}{\partial\theta^l_{ji}} = \theta^l_{ji} - \alpha\frac{\partial\mathcal{L}}{\partial a^l_j}\frac{\partial a^l_j}{\partial z^l_j}\frac{\partial z^l_j}{\partial\theta^l_{ji}},\\ &= \theta^l_{ji} - \alpha\delta^l_j\sigma' a^{l-1}_i,\end{aligned} \tag{6}$$

where $\delta^l_j = \frac{\partial\mathcal{L}}{\partial a^l_j}$ and $\sigma' = (\sigma^l_j)'$. δ^l_j is defined recursively as

$$\delta^l_j = \frac{\partial\mathcal{L}}{\partial a^l_j} = \sum_k \frac{\partial\mathcal{L}}{\partial a^{l+1}_k}\frac{\partial a^{l+1}_k}{\partial a^l_j} = \sum_k \delta^{l+1}_k\sigma'\theta^{l+1}_{kj}, \tag{7}$$

where $\sigma' = (\sigma^{l+1}_k)'$. k indicates the neurons in layer $l+1$ which are connected to jth neuron at layer l (Fig. 2).

The second-order term can be recursively computed as

$$\begin{aligned}\frac{\partial^2\mathcal{L}}{\partial{\theta^l_{ji}}^2} &= \left(\frac{\partial^2\mathcal{L}}{\partial{a^l_j}^2}\sigma'^2 + \frac{\partial\mathcal{L}}{\partial a^l_j}\sigma''\right)(a^{l-1}_i)^2\\ &= \left(\lambda^l_j\sigma'^2 + \delta^l_j\sigma''\right)(a^{l-1}_i)^2,\end{aligned} \tag{8}$$

where $\lambda^l_j = \frac{\partial^2\mathcal{L}}{\partial{a^l_j}^2}$, $\sigma' = (\sigma^l_j)'$, and $\sigma'' = (\sigma^l_j)''$. λ^l_j is defined recursively as

$$\begin{aligned}\lambda^l_j = \frac{\partial^2\mathcal{L}}{\partial{a^l_j}^2} &= \sum_k\left(\frac{\partial^2\mathcal{L}}{\partial{a^{l+1}_k}^2}\left(\sigma'\theta^{l+1}_{kj}\right)^2 + \frac{\partial\mathcal{L}}{\partial a^{l+1}_k}\sigma''\theta^{l+1}_{kj}\right)\\ &= \sum_k\left(\lambda^{l+1}_k\left(\sigma'\theta^{l+1}_{kj}\right)^2 + \delta^{l+1}_k\sigma''\theta^{l+1}_{kj}\right),\end{aligned} \tag{9}$$

where $\sigma' = (\sigma^{l+1}_k)'$ and $\sigma'' = (\sigma^{l+1}_k)''$. k indicates the neurons in layer $l+1$ which are connected to jth neuron at layer l (Fig. 2). Detailed derivation of (8) and (9) is provided in the appendix. From (8) and (9), it can be seen that back-propagation of gradient and second-order derivatives can be performed jointly in a single backward pass through the network. Due to this, the running of our proposed meta-learned is comparable to that of the state-of-the-art optimizers (e.g., RMSprop [2] and ADAM [3]).

IV. Learning the Update Function

The updates of CNN parameters in our quasi-Newton baseline, given by (4), are based on local approximation of the function $\mathcal{L}(\boldsymbol{\theta})$. These update steps are deterministically taken based on the current state of optimization, and are not affected by the previous steps. It is likely that better convergence rates could be achieved if the updates were estimated based on the history of previous updates as shown in RMSprop [2] and ADAM [3] update rules. To this end we replace the heuristic update rules of (4) with a learned update function. This new update function is implemented using the meta-LSTM. Both the meta-LSTM and the CNN parameters are learned with back-propagation through time (BPTT), which incorporates cues from previous learning iterations in subsequent updates of the LSTM and CNN parameters. At iteration t, we learn an update u^t as follows

$$\begin{aligned}\boldsymbol{\theta}^{t+1} &= \boldsymbol{\theta}^t + u^t,\\ \left[u^t, \boldsymbol{x}^{t+1}\right] &= LSTM(\boldsymbol{g}^t, \boldsymbol{h}^t, \boldsymbol{x}^t, \boldsymbol{\phi})\end{aligned} \tag{10}$$

where the update u^t is computed by the LSTM taking the gradient $\boldsymbol{g}^t$ and second-derivative $\boldsymbol{h}^t$ as inputs. The hidden state of the LSTM is denoted by $\boldsymbol{x}^t$ which is updated in each time step. LSTM parameters are denoted as $\boldsymbol{\phi}$. Note the both $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are learned jointly through BPTT.

Lets assume the final parameters of the CNN as $\boldsymbol{\theta}^*$ which depends of the underlying CNN objective $\mathcal{L}$ and the parameters of the update function, i.e., $\boldsymbol{\phi}$. Now we need to define an objective function for the LSTM to learn its parameters $\boldsymbol{\phi}$. Given a distribution of functions $\mathcal{L}$ we can express the LSTM objective as an expected loss

$$\mathcal{E}(\boldsymbol{\phi}) = \mathbb{E}_{\mathcal{L}}\left[\mathcal{L}(\boldsymbol{\theta}^*; \boldsymbol{\phi})\right]. \tag{11}$$

It can be noted that the objective function in (11) depends only on the final parameters $\boldsymbol{\theta}^*$. For training the LSTM to incorporate cues from previous time steps, it will be convenient to have an objective that depends on the entire trajectory of optimization, for some finite horizon T,

$$\mathcal{E}(\boldsymbol{\phi}) = \mathbb{E}_{\mathcal{L}}\left[\sum_{t=1}^{T} w^t\mathcal{L}(\boldsymbol{\theta}^t)\right], \tag{12}$$

Where $w^t \geq 0$ are arbitrary weights associated with each time-step. $\boldsymbol{g}^t$ and $\boldsymbol{h}^t$ are the gradient and Hessian, respectively, of the function $\mathcal{L}$ at $\boldsymbol{\theta}^t$. Note that this formulation is equivalent to (11) when $w^t = \mathbf{1}[t = T]$. Our learning framework is shown in Fig. 3.

To train the LSTM, we aim to optimize $\mathcal{E}(\boldsymbol{\phi})$ using gradient descent on $\boldsymbol{\phi}$. The gradient estimate $\partial\mathcal{E}(\boldsymbol{\phi})/\partial\boldsymbol{\phi}$ can be computed by sampling a random function $\mathcal{L}$ and applying back-propagation to the computational graph, shown in Fig. 3. We allow gradients to flow along the solid edges in the graph, but gradients along the dashed edges are dropped. Ignoring gradients along the dashed edges amounts to making the assumption that the gradients of the CNN, i.e., $\boldsymbol{\theta}$ do not depend on the LSTM parameters, i.e., $\boldsymbol{\phi}$. Which leads to $\partial\boldsymbol{g}^t/\partial\boldsymbol{\phi} = 0$ and $\partial\boldsymbol{h}^t/\partial\boldsymbol{\phi} = 0$. This assumption allows us to avoid computing derivatives of $\mathcal{L}$ with respect to $\boldsymbol{\phi}$.

Note that in (12) the gradient is non-zero only for terms where $w^t \neq 0$. For simplicity, we consider $w^t = 1$ for every t. This allows us to train the optimizer on partial trajectories.

A. LSTM Implementation of the Update Function

Optimizing millions of parameters in modern CNNs along with parameters of a fully connected LSTM is not scalable, as it requires a huge hidden state and an enormous number of parameters to model such state. Thus, we use an LSTM that operates coordinate-wise on the parameters of the objective function, similar to other common update rules like RMSprop [2] and ADAM [3]. This coordinate-wise network architecture allows us to use a very small network that only operates on a single coordinate of the optimizer. For scalability, LSTM share parameters across different parameters of the CNN.

As the LSTMs are learned coordinate-wise, once learned, it can be generalized to a CNN with arbitrary number of parameters. We evaluate the generalization capability of meta-learning in the results section. We implement the update rule for each coordinate using a two-layer LSTM network [10], using the forget gate architecture. The network takes as input the CNN gradient and second-order derivatives for a single coordinate as well as the previous hidden state and outputs the update for the corresponding updates u^t for CNN parameter.

The use of recurrence allows the LSTM to learn dynamic update rules which integrate information from the history of gradients, similar to momentum. This is known to have many desirable properties in convex optimization (e.g., [20]) and in fact many recent learning procedures such as ADAM [3] uses momentum in their updates.

V. Results

Experimental setup. We follow the experimental setup of [19] which considers learned gradient updates to train CNNs and can be considered as a reasonable baseline to justify the importance of second derivatives in meta-learning. We use two-layer LSTMs with 20 hidden units in each layer. The BPTT optimization is learned using the ADAM optimizer where the learning rate is chosen through a random search. We consider $T = 20$ as the length of time horizon. We consider CNNs consisting of two types of parameters: convolutional and fully connected parameters. For each type of parameters, we learn specific LSTMs. LSTMs share parameters across the units but the units maintain distinct hidden states.

Datasets. We consider the image classification task and evaluate the learning approach on three benchmark datasets: MNIST [16], CIFAR-10 [17], and ImageNet [18]. The MNIST digit dataset consists 28x28 images of the 10 handwritten digits. There are 60,000 training images and 10,000 test images. The CIFAR-10 dataset consists 60000 32x32 color images of 10 classes with 6000 examples per class. There are 50,000 training images and 10000 test images. Finally, we consider the Imagenet ILSVRC 2014 consisting 1.2 million training images of 1000 object classes and 100,000 test images.

CNN architecture. We consider a CNN architecture suitable for image classification as in [21]. For MNIST and CIFAR-10 datasets, we consider a CNN with 3 convolutional layers and 2 fully connected layers with 32 neurons. Convolutional layers consist of 5x5 filters and 64 feature maps for each layer with ReLU activation. For Imagenet, we consider 5 convolutional layers and 2 fully connected layers with 4096 neurons. The dimension of feature maps for five convolutional layers are 64, 256, 512, 512, 512, respectively.

Baselines. We consider the following baselines.

- **Sub-gradient descent (SGD)** : Here we use the vanilla SGD to update parameters.
- **RMSprop** [2]: In this approach, the learning rate is divided by an exponentially decaying average of squared gradients. The decay rate is set to 0.9 while computing the running average of the squared gradients.
- **ADAM** [3]: In addition to storing an exponentially decaying average of previous squared gradients like RMSprop, ADAM also keeps an exponentially decaying average of past gradients which is commonly known as momentum. The decay rate is set to 0.9 for momentum and 0.99 for squared gradients.
- **LSTM** [19]: This approach uses an LSTM to learn the update function based on only gradients. Comparison with this approach justifies the importance of considering the second derivatives in the update function
- **Second order updates (SOU)**: To justify the importance of learning of the second-order cues in parameter updates, we define this baseline where gradient steps are scaled using the Hessian matrix as in (2). We consider the approximate diagonal Hessian $\hat{\mathbf{H}}$.

We call our approach second-order LSTM (SLSTM). We compare SLSTM with the above-mentioned baselines on MNIST, CIFAR-10, and ImageNet datasets. In case of MNIST dataset, we sample random minibatches in each step of learning and update the SLSTM parameters. Then performance is evaluated on the test set keeping the optimizer fixed after each step. On MNIST, we evaluate our approach for a limited 100 steps compared with the baselines approaches in term of loss and error metrics. We present the loss vs. steps and error vs. steps plots in Fig. 4. In case of CIFAR-10 dataset, we train our SLSTM on the training set and apply the learned optimizer to fit an *unseen* test data. Corresponding loss vs. steps and error vs. steps plots are shown in Fig. 5. Similar to CIFAR-10, on ImageNet, we train the optimizer on the training set and use validation set to fit the optimizer. We present the error vs. steps plots in Fig. 6.

We see from the above comparisons that though other optimization methods are able to achieve similar loss values as SLSTM but our SLSTM converges faster than other baselines on all the datasets. Note that in case of the MNIST dataset (Fig. 4), our approach achieves better loss and error values when the optimization is run for a limited number of steps. Also, our SLSTM not only minimizes the desired loss function during training, it also achieves low error measure during testing (loss vs .steps plots in Fig.4, 5, and 6). This implies that SLSTM does not learn to minimize the loss function by overfitting the training data.

Superior convergence rate of SLSTM compared to the hand-designed gradient-based approaches, e.g., SGD, RMSprop, and ADAM justify the efficiency of learning based updates to train CNNs. SLSTM also outperforms LSTM and SOU, which

justifies the importance of considering second derivatives in the learned update function. Performance on ImageNet (Fig. 6) justifies that our SLSTM approach is scalable to bigger network and can be applicable to large-scale problems. As during training a CNN, randomly chosen samples can introduce uncertainty in training, we run SLSTM and other baseline approaches for five iterations and report averaged metrics.

Generalization of the LSTM learner. One important aspect of the learning based approach is that the update function learns a general policy to update individual parameters, based on the corresponding gradient and second derivative. Thus, once learned for any network, the learned update functions can be transferred to learn parameters of a new network as long as the gradient and second-derivatives are available. Recall that we learn distinct LSTMs for all the CNN parameters. Thus the learned LSTM modules can be easily exported to train a network with arbitrary number of parameters. To test the generalization capability, we perform an experiment where we learn the update function for a small network with 1 convolutional layer and 1 fully connected layer. Then the learned LSTM is used to train a bigger network with 3 convolutional layers and 2 fully connected layers. Moreover, we only consider 5 object classes during training while test set consists of 10 object classes. We perform the experiments on CIFAR-10 dataset and the results are shown in Fig. 7. Note that SLSTM outperforms the baselines while applied to learn a new network on a novel dataset, even though it is not specifically learned for that network.

VI. Conclusion

We have presented a new meta-learning approach to estimating CNN parameters leveraging recent advances in second-order optimization. Our key contribution is to employ a meta-LSTM that takes the gradients and second derivatives as input, and uses its memory of previous iterations to predict updates of CNN parameters for the next iteration. Learning of the meta-LSTM and CNN is done jointly. We have evaluated our approach on the task of image classification on three benchmark datasets: MNIST, CIFAR-10, and ImageNet. Our approach achieves faster convergence and better performance compared to the gradient based counterpart. Also the results demonstrates that out meta-LSTM learned for small networks generalizes well to learn a new larger network.

Appendix

Computing the Second-Order Derivatives. In this section, we explain the computation of the second derivatives as mentioned in (8) and (9). We assume a multilayer neural network architecture as described in Sec. III in Fig. 2 which can be easily extended to any network, such as CNNs..

Our goal is the compute the main diagonal of Hessian with respect to the loss term $\mathcal{L}$, i.e., $\frac{\partial^2 \mathcal{L}}{\partial {\theta^l_{ji}}^2}$ where we assume the network is feed-forward and neurons are arranged layer-wise. Furthermore, we assume, there are no skip connections in the network. Following [4], [22], we compute the diagonal Hessian as follows

$$\begin{aligned} \boldsymbol{h}^l_{ji} = \frac{\partial^2 \mathcal{L}}{\partial {\theta^l_{ji}}^2} &= \frac{\partial \frac{\partial \mathcal{L}}{\partial \theta^l_{ji}}}{\partial \theta^l_{ji}} = \frac{\partial \frac{\partial \mathcal{L}}{\partial a^l_j}\frac{\partial a^l_j}{\partial z^l_j}\frac{\partial z^l_j}{\partial \theta^l_{ji}}}{\partial \theta^l_{ji}}, \\ &= \left(\frac{\partial^2 \mathcal{L}}{\partial {a^l_j}^2}\sigma'^2 + \frac{\partial \mathcal{L}}{\partial a^l_j}\sigma''\right)(a^{l-1}_i)^2, \end{aligned} \tag{13}$$

where we use the shorthand $\sigma' = (\sigma^l_j)'$ and $\sigma'' = (\sigma^l_j)''$. Note that the above-mentioned expression is inherently recursive. When $l = L$, i.e., in the final layer, the derivatives are computed based on the loss function directly. Otherwise, derivatives are computed (and back-propagated) based on the layer above. Error derivatives depend on the loss function, for example, if we consider the cross-entropy loss, then at the final layer, $\frac{\partial \mathcal{L}}{\partial a^L_j} = -\sum_k \frac{o_k}{a^L_j}$ and $\frac{\partial^2 \mathcal{L}}{\partial {a^L_j}^2} = \sum_k \frac{o_k}{{a^L_j}^2}$, where $o_k \in \{0, 1\}$ is the ground-truth for k the class. For squared distance error, similarly, $\frac{\partial \mathcal{L}}{\partial a^L_j} = -(o_k - a^L_j)$ and $\frac{\partial^2 \mathcal{L}}{\partial {a^L_j}^2} = 1$.

For an intermediate layer $l \neq L$, $\frac{\partial^2 \mathcal{L}}{\partial {a^l_j}^2}$ and $\frac{\partial^2 \mathcal{L}}{\partial {a^l_j}^2}$ are computed from the layer above like standard back-propagation. We show the computation of the second derivative, i.e., $\frac{\partial^2 \mathcal{L}}{\partial {a^l_j}^2}$ and $\frac{\partial^2 \mathcal{L}}{\partial {a^l_j}^2}$ is computed as standard gradient back-propagation

$$\begin{aligned} \lambda^l_j = \frac{\partial^2 \mathcal{L}}{\partial {a^l_j}^2} &= \frac{\partial \frac{\partial \mathcal{L}}{\partial a^l_j}}{\partial a^l_j} = \frac{\partial \left(\sum_k \frac{\partial \mathcal{L}}{\partial a^{l+1}_k}\frac{\partial a^{l+1}_k}{\partial a^l_j}\right)}{\partial \theta^{l+1}_{kj}}, \\ &= \sum_k \left(\frac{\partial^2 \mathcal{L}}{\partial {a^{l+1}_k}^2}\left(\sigma' \theta^{l+1}_{kj}\right)^2 + \frac{\partial \mathcal{L}}{\partial a^{l+1}_k}\sigma'' \theta^{l+1}_{kj}\right) \end{aligned} \tag{14}$$

where $\sigma' = (\sigma^{l+1}_k)'$ and $\sigma'' = (\sigma^{l+1}_k)''$.

Acknowledgment

This work was supported in part by DARPA XAI Award N66001-17-2-4029

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," Tech. Rep., 1985.

[2] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, 2012.

[3] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2014.

[4] S. Becker, Y. Le Cun *et al.*, "Improving the convergence of back-propagation learning with second order methods," in *Proceedings of the 1988 connectionist models summer school*, 1988, pp. 29–37.

[5] J. Martens, I. Sutskever, and K. Swersky, "Estimating the hessian by back-propagating curvature," in *ICML*, 2012.

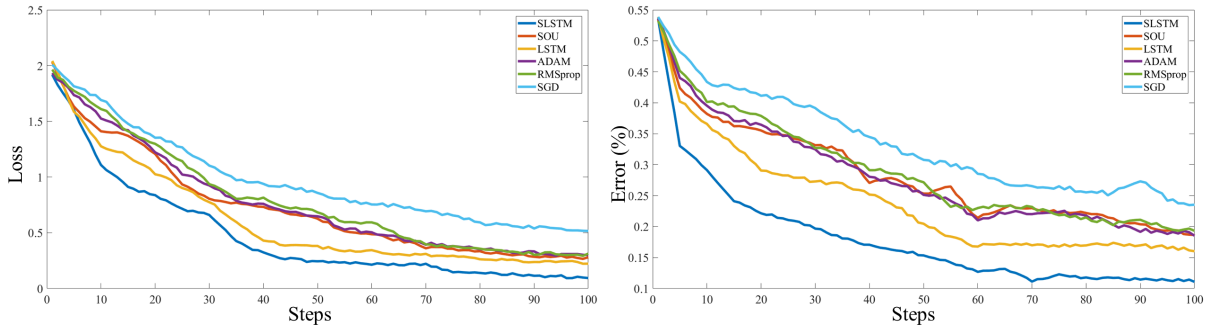
[6] O. Vinyals and D. Povey, "Krylov subspace descent for deep learning." in *AISTATS*, 2012.

[7] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," in *NIPS*, 2014.

[8] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The loss surfaces of multilayer networks." in *AISTATS*, 2015.

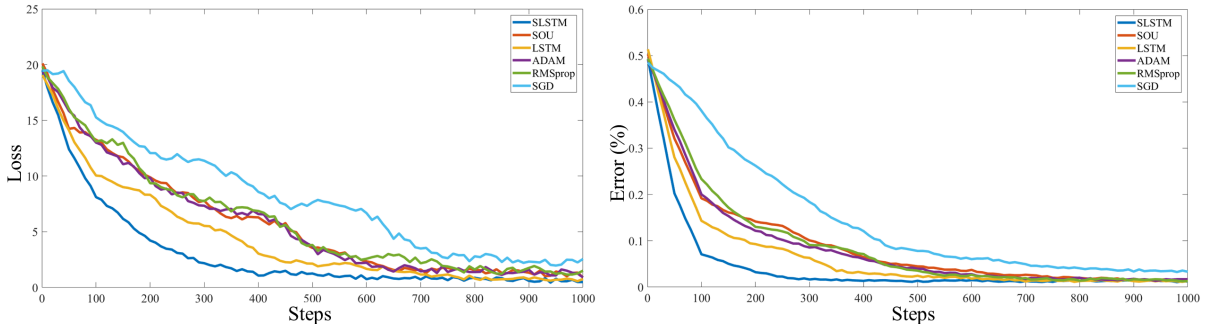
[9] K. Kawaguchi, "Deep learning without poor local minima," in *NIPS*, 2016.

[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.



(a) (b)

Fig. 4: Comparison of our SLSTM to the baselines in terms of a) Loss and (b) Error(%) vs. steps on the MNIST dataset.



(a) (b)

Fig. 5: Comparison of our SLSTM to the baselines in terms of a) Loss and (b) Error(%) vs. steps on the CIFAR-10 dataset.

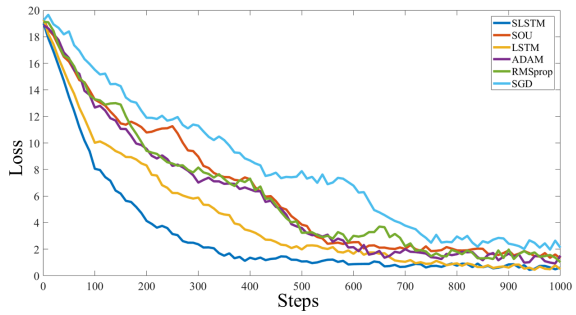


Fig. 6: Loss vs. steps plot corresponding to the baselines and our approach (SLSTM) on the ImageNet dataset.

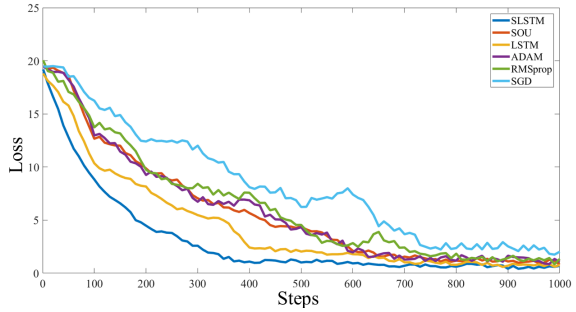


Fig. 7: Loss vs. steps plot on the CIFAR-10 dataset where we learn the LSTM with a smaller network and then use it to train a bigger network.

[11] J. Schmidhuber, "Learning to control fast-weight memories: An alternative to dynamic recurrent networks," *Neural Computation*, vol. 4, no. 1, pp. 131–139, 1992.

[12] ——, "A neural network that embeds its own meta-levels," in *ICNN*, 1993.

[13] S. Thrun and L. Pratt, *Learning to learn*. Springer Science & Business Media, 2012.

[14] J. Martens, "Deep learning via hessian-free optimization," in *ICML*, 2010.

[15] O. Chapelle and D. Erhan, "Improved preconditioner for hessian free optimization," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[17] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009.

[19] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas, "Learning to learn by gradient descent by gradient descent," in *NIPS*, 2016.

[20] Y. Nesterov, "A method of solving a convex programming problem with convergence rate o (1/k2)," in *Soviet Mathematics Doklady*, vol. 27, no. 2, 1983, pp. 372–376.

[21] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[22] W. L. Buntine and A. S. Weigend, "Computing second derivatives in feed-forward networks: A review," *IEEE transactions on Neural Networks*, vol. 5, no. 3, pp. 480–488, 1994.