Decoupling Backpropagation using Constrained Optimization Methods

Akhilesh Gotmare[*]
EPFL
akhilesh.gotmare@epfl.ch

Valentin Thomas[*]
MILA, Université de Montréal
vltn.thomas@gmail.com

Johanni Brea
EPFL
johanni.brea@epfl.ch

Martin Jaggi
EPFL
martin.jaggi@epfl.ch

Abstract

We propose BlockProp, a neural network training algorithm. Unlike backpropagation [1], it does not rely on direct top-to-bottom propagation of an error signal. Rather, by interpreting backpropagation as a constrained optimization problem we split the neural network model into sets of layers (blocks) that must satisfy a consistency constraint, *i.e.* the output of one set of layers must be equal to the input of the next. These decoupled blocks are then updated with the gradient of the optimization constraint violation. The main advantage of this formulation is that we decouple the propagation of the error signal on different subparts (blocks) of the network making it particularly relevant for multi-devices applications.

1 Introduction

The backpropagation algorithm [1] has been the core ingredient to most successes in deep learning, from image recognition to reinforcement learning [2, 3]. Nevertheless, studying alternatives to backpropagation is currently a very active recent area of research, see e.g. [4, 5] and many others. While motivation for some approaches comes from biological plausibility of the learning, others address on computational aspects, such as large-scale distributed training in the data-parallel or model-parallel scenario.

The focus of our work here is the latter scenario, when a neural network is split into several pieces. Here, we consider joint training with many devices, each device only holding (the weights of) a subgroup of the layers. This setup becomes increasingly important with the ever increasing growth of model sizes on one hand, and with Moore's law on single-chip density increase coming to an end. In this setting, our main contribution is an algorithm called BlockProp for efficient model-parallel training, which shares similarities with [6] and [7]. By viewing backpropagation as a constrained optimization problem we divide the neural network into different "blocks" which can consist in either a single layer as in [6, 8, 7] but more importantly a block can also be a subset of layers. Between blocks, we introduce a consistency constraint epitomizing the fact that the output of block k: h_k must be equal to the input of next block, block $k + 1$. This way, we can train the network by decoupling optimization between the blocks, by alternating the steps of updating the auxiliary variables, or "gluing" variables h (h-step) and the weights of each block (W-step) as shown in Figure 2 and Section 2. Furthermore, compared to [6] and [8] which are batch algorithms our algorithm is on-line –

[*]Equal contribution

Submitted to 35th International Conference on Machine Learning's Workshop on Credit Assignment in Deep Learning and Deep Reinforcement Learning (ICML 2018 ECA), Stockholm, Sweden.

i.e. forms updates based on one training example a time – and uses backpropagation in each of the blocks sequentially to satisfy the induced constraints.

2 Backpropagation as an optimization procedure

We begin by establishing some notation before describing our proposed training strategy. Given a dataset of N training instance pairs, let $\{x^{(n)}, y^{(n)}\}$ where $x^{(n)} \in \mathbb{R}^d$ is the feature vector of n-th data instance and $y^{(n)}$ is the corresponding scalar (regression) or one-hot target vector of size p (p-ary classification). The predictions or output values for input $x^{(n)}$ obtained by a forward pass of a neural network with L hidden layers can be written as

$$\hat{y}^{(n)} = f(x^{(n)}; \{W_1 \ldots W_{L+1}\}) = f_{L+1}(W_{L+1}^\top f_L(W_L^\top f_{L-1}(\ldots f_1(W_1^\top x^{(n)})))) \tag{1}$$

where $W_1, W_2 \ldots W_{L+1}$ are the linear transformations (weight matrices corresponding to respective hidden layers) of the neural network and $f_1, f_2 \ldots f_L$ are non-linear activation functions (typically we use the same non-linearities for all the layers e.g. ReLU or softmax) while f_{L+1} represents the last layer's non-linearity that performs the desired task (often identity for regression and softmax for classification).

Training the neural network amounts to minimizing the empirical risk

$$\underset{\{W_1, W_2, \ldots W_{L+1}\}}{\text{minimize}} \quad \sum_{n=1}^{N} \ell(f(x^{(n)}; \{W_1, \ldots W_{L+1}\}), y^{(n)}) = \ell(f(x; W), y) \tag{2}$$

where ℓ is a loss function (eg. cross-entropy for classification, ℓ_2 or mean-squared error for regression).

In our formulation we split the ordered set consisting of layer weights $W = \{W_1, W_2, \ldots W_{L+1}\}$ into K consecutive blocks $\{B_1, \ldots B_K\}$ where each block is a subset of W, $B_k = \{W_{l_k}, W_{l_k+1}, \ldots W_{l_{k+1}-1}\}$. While we could allow this division to be arbitrary, as we discuss in Section 4, it is preferable to have a small number of blocks and roughly equal number of layers in each block. We specify the forward pass function of a block B_k in Definition 2.1.

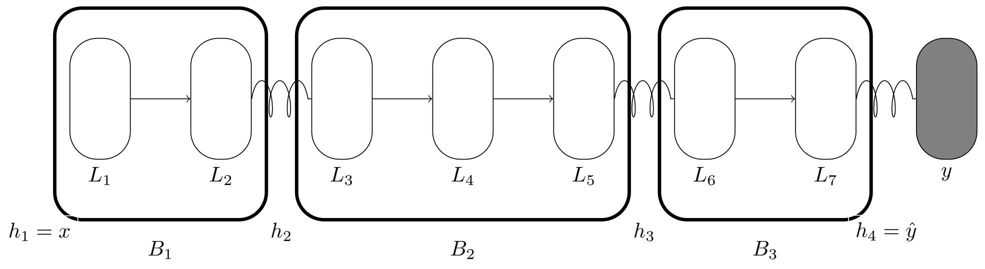

Figure 1: Neural network with 7 layers ($L_1, \ldots, L_7$) separated into 3 arbitrary blocks. The spring links represent the relaxed constraints.

Definition 2.1. If block B_k consists of the weights $\{W_j, \ldots, W_{j+l}\}$ the function associated with block B_k is

$$b_k(h) = f_{j+l}(W_{j+l}^\top f_{j+l-1}(\ldots f_j(W_j^\top h)))$$

We can then re-write the optimization problem in (2) as

$$\begin{aligned} &\underset{\{W_l\}}{\text{minimize}} && \sum_n \ell(b_K(h_K^{(n)}), y^{(n)}) \\ &\text{subject to} && h_{k+1}^{(n)} = b_k(h_k^{(n)}) \text{ for } 2 \leq k \leq K-1 \\ &\text{and} && h_2^{(n)} = b_1(x^{(n)}) \end{aligned} \tag{3}$$

Given a training instance n, we choose to minimize an augmented (stochastic) loss function for the above problem, that is

$$\mathcal{L}(h^{(n)}; W) = \sum_{k=1}^{K-1} \widetilde{\ell}_k(h_{k+1}^{(n)} - b_k(h_k^{(n)})) + \ell(y^{(n)}, b_K(h_K^{(n)})) \tag{4}$$

Where the functions $\widetilde{\ell}_k$ are positive and null if and only if both arguments are equal. Note that in most case we will simply choose the L_2 penalty $\widetilde{\ell}_k(a,b) = \lambda_k||a-b||^2$ for $\lambda_k > 0$.

where $h^{(n)}$ is the set $\{h_2^{(n)}, h_3^{(n)} \ldots h_K^{(n)}\}$. Note that we arbitrarily chose a L_2 quadratic penalty, but other choices are valid as well. Now that we have the augmented loss function, we solve the constrained optimization problem in (3) by minimizing the Lagrangian in (4) with respect to the set of h variables first, and then with respect to the weights W and by iterating over these two sets of variables in an alternating fashion.

Let us first consider minimization with respect to $h_k^{(n)}$ for a given k. We begin by defining the quantity $\mathcal{L}_k^{(n)}$, the Lagrangian for the k-th block as follows

$$\mathcal{L}_k^{(n)}(h_k^{(n)}, B_k, B_{k-1}) = \begin{cases} \widetilde{\ell}_1(h_2^{(n)}, b_1(x^{(n)})) + \widetilde{\ell}_2(h_3^{(n)}, b_2(h_2^{(n)})), & \text{if } k = 2 \\ \widetilde{\ell}_k(h_{k+1}^{(n)}, b_k(h_k^{(n)})) + \widetilde{\ell}_{k-1}(h_k^{(n)}, b_{k-1}(h_{k-1}^{(n)})), & \text{if } 2 < k \leq K-1 \\ \ell(y^{(n)}, b_K(h_K^{(n)})) + \widetilde{\ell}_{K-1}(h_K^{(n)}, b_{K-1}(h_{K-1}^{(n)})), & \text{if } k = K \end{cases} \tag{5}$$

For a given k in the above definition we assume $\{h_2^{(n)}, \ldots h_{k-1}^{(n)}\}$ and $\{h_{k+1}^{(n)} \ldots h_K^{(n)}\}$ as fixed. The quantity $\mathcal{L}_k^{(n)}$ for each block is defined by collecting terms in the augmented loss $\mathcal{L}^{(n)}(h^{(n)}, W)$ in (4) that involve $h_k^{(n)}$ for a given k, thus we have

$$\underset{\{h_k^{(n)}\}}{\arg\min}\ \mathcal{L}^{(n)} = \underset{\{h_k^{(n)}\}}{\arg\min}\ \mathcal{L}_k^{(n)}$$

To obtain (or approximate) the optimal $h_k^{(n)}$ one could take SGD steps (as shown in Section 3) or use second-order methods. We denote the optimal $h_k^{(n)}$ obtained by $h_k^{(n)\star}$ and by $h^{(n)\star}$ the set $\{h_2^{(n)\star}, h_3^{(n)\star} \ldots h_K^{(n)\star}\}$.

We now discuss the minimization over the W variable, by first defining

$$\mathcal{J}^{(n)}(W) = \min_h \mathcal{L}^{(n)}(h, W)$$

The h step discussed before approximates the $\underset{h}{\arg\min}\ \mathcal{L}^{(n)}$ as $h^{(n)\star}$. Therefore

$$\mathcal{J}^{(n)}(W) \approx \mathcal{L}^{(n)}(h^{(n)\star}, W)$$

(the sanity of this approximation depends on how well the optimization is performed in the h-step). The next step would then be to minimize $\mathcal{L}^{(n)}(h^{(n)\star}, W)$ over the W variable. Using (4) we can write

$$\mathcal{J}^{(n)}(W) = \mathcal{L}^{(n)}(h^{(n)\star}, W) = \widetilde{\ell}_1(h_2^{(n)\star}, b_1(x^{(n)})) + \sum_{k=2}^{K-1} \widetilde{\ell}_k(h_{k+1}^{(n)\star}, b_k(h_k^{(n)\star})) + \ell(y^{(n)}, b_K(h_K^{(n)\star})) \tag{6}$$

We now define $\mathcal{J}_k^{(n)}$ for the k-th block as

$$\mathcal{J}_k^{(n)}(B_k) = \begin{cases} \widetilde{\ell}_1(h_2^{(n)\star}, b_1(x^{(n)})), & \text{if } k = 1 \\ \widetilde{\ell}_k(h_{k+1}^{(n)\star}, b_k(h_k^{(n)\star})), & \text{if } 2 \leq k \leq K-1 \\ \ell(y^{(n)}, b_K(h_K^{(n)\star})), & \text{if } k = K \end{cases} \tag{7}$$

Note $\mathcal{J}^{(n)}(W) = \sum_k \mathcal{J}_k^{(n)}(B_k)$. Using (7), it is clear that for a given k,

$$\underset{B_k}{\arg\min}\ \mathcal{J}^{(n)}(W) = \underset{B_k}{\arg\min}\ \mathcal{J}_k^{(n)}(B_k)$$

It is worth emphasizing here that these minimization subproblems of finding the optimal weights B_k for different k can be solved in parallel since they are independent in the sense that solving one does not require solving the other. These substeps of minimizing with respect to the weights in B_k are essentially backpropagation procedures over the blocks defined by our splitting. Lemma B.1 presents a straightforward way to compute these gradients. In the next section, we present the algorithm involving the minimization steps discussed here.

3 Algorithm

The weights of the neural network are randomly initialized, similar to standard training. For each training instance, the 'h-step' optimizes the partial objective $\mathcal{L}_k^{(n)}(h_k^{(n)}, B_k, B_{k-1})$ defined in (5), withh respect to the gluing variable $h_k^{(n)}$, for each k, in a sequential manner, starting from $k = K$ down to $k = 2$. We use SGD to perform this subproblem minimization, by computing $\nabla_{h_k^{(n)}} \mathcal{L}_k^{(n)}$ for each k and taking negative steps in this direction to obtain an approximately optimal $h_k^{(n)\star}$. Next, for each k, we optimize $\mathcal{J}_k^{(n)}$ over the set of weights B_k by performing the smaller backpropagation on the k-th block with $h_k^{(n)\star}$ as its input and $h_{k+1}^{(n)\star}$ as its output, where $h_1^{(n)\star} = x^{(n)}$ and $h_{K+1}^{(n)\star} = y^{(n)}$. The pseudo-code for the resulting training strategy is shown in Algorithm 1. We need to choose the hyperparameter N_h, which denotes the number of steps taken in the 'h-step' of the minimization. In the pseudo-code below, we consider a single training instance at a time (similar to online learning setup), however we could also run the training using mini-batches as a trivial extension of the presented framework.

In Algorithm 1, the blocks B_k, obtained after splitting the set of weights in the neural network, are assumed to be residing on the memory of different devices (block B_k on device D_{B_k}'s memory). This setup brings the possibility of parallelizing the W-step (operation 20 in the pseudocode). This step of block updates is equivalent to :

$$W \leftarrow W - \gamma \nabla_W \mathcal{J}^{(n)}(W)$$

as $W = \bigcup_{k=1}^{K} B_k$ and $\mathcal{J}^{(n)}(W) = \sum_{k=1}^{K} \mathcal{J}_k^{(n)}(B_k)$. Note there is no overlap between the variables inside different blocks and therefore it is possible to parallelize this step as discussed in the previous section.

Hence the last step of block updates can be computed in a parallel fashion; either on different devices (GPUs) or different threads of a single device. This would however require that the initialization for $h_k^{(n)}$ and the optimal gluing variable $h_k^{(n)\star}$ get communicated within devices (operations 6 and 15 in the pseudo-code). Note that the algorithm can be run equivalently on a single device D_0 by treating $D_{B_k} = D_0 \,\forall\, k$ and ignoring the data copy operations 6 and 15 in the pseudo-code.

We need not store a copy of the entire network on each device (GPU). Each device only takes care of one block (D_{B_k} stores B_k), thus we have a lesser RAM footprint, we could potentially use much deeper/larger networks. In comparison with Data Parallelism, the algorithm discussed above incur lesser communication overhead, since unlike the gather and broadcast of weights (shared parameters) in data parallel training, here we need only the gluing variables, that will typically be smaller in size than the weights, to be communicated between devices. However, Data Parallelism can be thought of as an orthogonal way to speed up training and it might not be fair to compare the two.

As we allude to in Appendix A, the above strategy will be necessarily slower than standard (sequential) training using backpropagation, since the h-step would involve backpropagation over the subsequent blocks. In order to be faster than standard backpropagation using model parallelism, we need to cache the activations from the gluing variable as described in Appendix C and perform the h-steps infrequently.

In order to investigate the effectiveness of the methods discussed, we study the empirical performance of Algorithm 1 in the next section. These experiments focus on the validating the training capabilities of the above strategy for simple networks trained on small scale datasets (MNIST and CIFAR10). Further large scale experiments that make use of model parallelism, in order to speed up the training process using our framework, are future pieces of this ongoing work.

4 Experiments

4.1 MLP on MNIST

We use the algorithm discussed in the previous section to decouple and train deep neural networks on small scale datasets. A fully connected neural network with a single hidden layer ($L = 1$) and set

Algorithm 1 BlockProp

```
1:  Initialize weights $\{W_1, \dots W_{L+1}\}$ of the neural network
2:  for $epoch = 1, 2 \dots, N_{epoch}$ do
3:      for $n = 1, 2 \dots, N$ do
4:          // h step
5:          Initialize $h_k^{(n)}$ $\forall k$ with the activations of the forward pass
6:          // Device $D_{B_k}$ will send $h_{k+1}^{(n)} = b_k(h_k^{(n)})$ to the next device $D_{B_{k+1}}$
7:          for $k = K, K-1, \dots 2$ do
8:              // Compute the $h_k^{(n)\star}$ to get $\mathcal{J}_k^{(n)}$
9:              // Find $h_k^{(n)\star} \approx \arg\min_{h_k^{(n)}} \mathcal{L}_k^{(n)}(h_k^{(n)}, B_k, B_{k-1})$ via gradient descent ($N_h$ steps):
10:             $\widetilde{h}^{(1)} \leftarrow h_k^{(n)}$
11:             for $t = 1, 2 \dots, N_h$ do
12:                 $\widetilde{h}^{(t+1)} \leftarrow \widetilde{h}^{(t)} - \gamma_{h_k} \nabla_{h_k^{(n)}} \mathcal{L}_k^{(n)}(h_k^{(n)}, B_k, B_{k-1}) \,|_{h_k^{(n)} = \widetilde{h}^{(t)}}$
13:             end for
14:             $h_k^{(n)\star} \leftarrow \widetilde{h}^{(N_h+1)}$
15:             // Device $D_{B_k}$ will send $h_k^{(n)\star}$ to the previous device $D_{B_{k-1}}$
16:         end for
17:         // W-step
18:         // Compute $\nabla \mathcal{J}^{(n)}$ and take an SGD step with respect to the weights
19:         // Perform stochastic gradient step on block $B_k$
20:         $B_k \leftarrow B_k - \gamma_{B_k} \nabla_{B_k} \mathcal{J}_k^{(n)}, \forall k$
21:     end for
22: end for
```

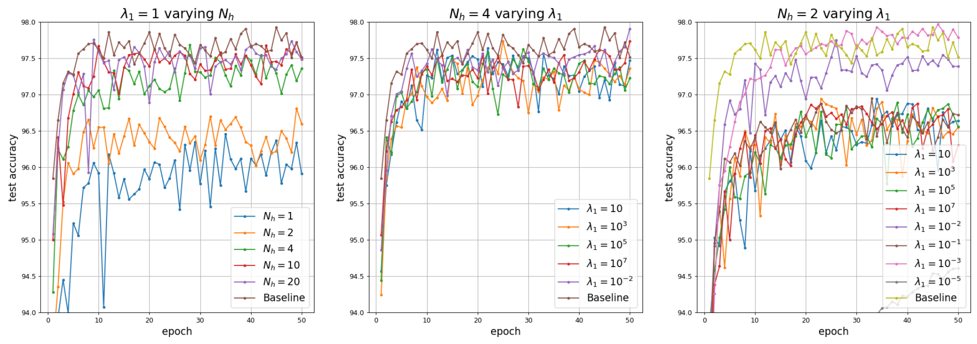

Figure 2: Fully connected network with single hidden layer trained on MNIST

of weights $W = \{W_1, W_2\}$ is split into blocks $B_1 = \{W_1\}$ and $B_2 = \{W_2\}$ as per the formulation discussed in Section 2. We could write the Lagrangian for this setup as

$$\begin{aligned} \mathcal{L}_2^{(n)}(h_2^{(n)}, B_2, B_1) &= \widetilde{\ell}_1(h_2^{(n)}, b_1(x^{(n)})) + \ell(y^{(n)}, b_2(h_2^{(n)})) \\ &= \underbrace{\lambda_1 \, \|h_2^{(n)} - b_1(x^{(n)})\|^2}_{\text{gluing loss}} + \underbrace{\ell(y^{(n)}, b_2(h_2^{(n)}))}_{\text{last layer loss}} \end{aligned}$$

where λ_1 is the weight of the gluing loss penalty. We use MNIST digit classification dataset of $60,000$ images [9] for training.

We begin by fixing the hyper-parameter λ_1 to 1 and observe that increasing N_h (the number of 'h-steps' taken) seems to improve the training as shown in Figure 2 (left). We also vary the penalty weight λ_1 for $N_h = 4$ (Figure 2 (center)) and $N_h = 2$ (Figure 2), where an evident pattern is hard to find. The baseline shown in Figure 2 corresponds to standard training using SGD. The difference in test accuracies attained between standard training and the algorithm from Section 3 can be attributed to the inconsistency that we allow in our network training by the means of introducing gluing variables and breaking the inter-block dependencies. Finding ways of mitigating this limitation is a challenging future direction of our work.

4.2 ResNet on CIFAR10

The framework and analysis presented in previous sections for fully connected networks can be easily extended to convolutional networks, by simply treating the convolution operation as a sparse linear transformation. We therefore experiment with Residual Networks, specifically the ResNet18 architecture presented in [10, 11] and train them on the CIFAR10 dataset [12]. The network is split into two blocks B_1 and B_2 such that each block carries roughly equal number of layers.

Figure 3 consists plots for the gluing loss $||b_1(x) - h||_2^2$ without the λ_1 scaling (top left), cross entropy loss $\ell(b_2(h), y)$ computed at the last layer (top right), test accuracy (bottom left) and test loss (bottom right) over the course of training iterations for different choices of λ_1, γ_{B_1} - the step-size for B_1 update and γ_{B_2} - step-size for B_2 update. For all these variants, the number of h-steps N_h is fixed to 1 and the step-size for the h-step γ_h is fixed to 0.01. The four subplots in Figure 3 share the same legend as in subplot 3 (bottom left). As can be seen from the plots, higher values of λ_1 seem to help the performance to a certain extent. For very small values, the training seems to be unstable (the light blue curve below). For an unidentified reason, we were unable to successfully train these models when using dropout, this might partially explain the performance gap for our strategy in comparison to standard training.

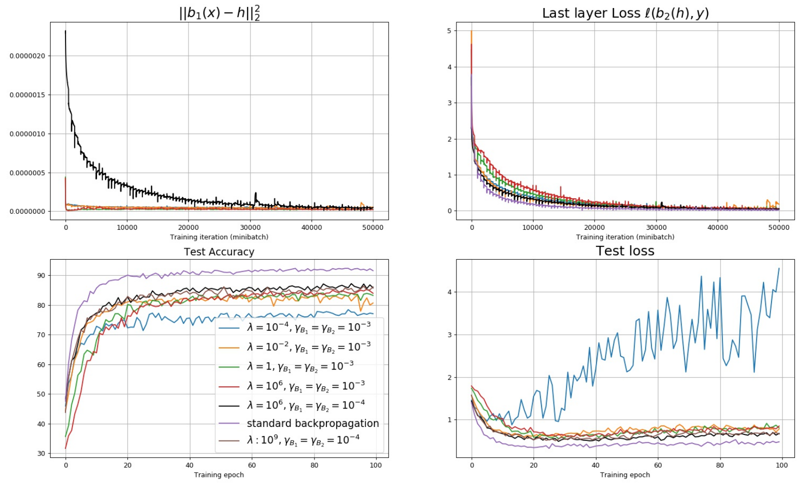

Figure 3: Distributed training vs Standard backpropagation - ResNet18

5 Related work

5.1 Link with other works in optimization

From an optimization perspective, looking for more scalable alternatives to backprogagation is also an active area of research. [6] proposed to rewrite the backpropagation as a constrained optimization problem which is solved sequentially. Another work [8] extends this idea using an Alternating Direction Method of Multipliers (ADMM) [13] and were able to train neural networks without using gradients from backpropagation at all. However both these methods work in a batch setting, where the updates have to be computed on the whole training set, which makes it impractical for modern deep learning with very large datasets. In practice, we also found the lagrangian multiplier in [8] to be unnecessary, and the algorithm to be more stable without the same. Using SGD to minimize the constraints enables us to be on-line and as such much more efficient. In a similar line of work, [7] recently proposed a proximal variant of backpropagation, allowing for bigger learning rates and thus faster convergence.

5.2 Relation to Equilibrium Propagation

In [4], authors present the Equilibrium Propagation algorithm for energy based models and introduce it as a framework for training the continuous Hopfield model. The objective to be minimized is a sum of the internal and external energy of the model.

We present some theoretical analysis to justify the validity of the proposed decoupling scheme.

Theorem 5.1 (Equivalence with Equilibrium propagation)**.** *BlockProp is equivalent to Equilibrium propagation, if each block consists of only one layer and the (unweigthed) internal energy function is given by*

$$E_n = \sum_{k=1}^{K-1} \|h_{k+1}^{(n)} - b_k(h_k^{(n)})\|_2^2$$

and the external cost

$$\mathcal{C}_n = \ell(y^{(n)}, h_K^{(n)})$$

with $h_1^{(n)} = x^{(n)}$ (instead of Eq. (1) in [4]).

***More concretely**, for the 2 block case ($K = 3$), if we denote the usual loss by $\mathcal{J}$, where*

$$\mathcal{J}(x^{(n)}, y^{(n)}) := \ell(y^{(n)}, b_2(b_1(x^{(n)}))) = C_n(y^{(n)}, \arg\min E_n)$$

and the augmented loss function by $\mathcal{L}$, where

$$\mathcal{L}(x^{(n)}, y^{(n)}, h^{(n)}, \beta) := \ell_1(x^{(n)}, h^{(n)}) + \beta\, \ell_2(y^{(n)}, h^{(n)})$$

with $\ell_1(x^{(n)}, h^{(n)}) = \|h^{(n)} - b_1(x^{(n)})\|_2^2$ and $\ell_2(y^{(n)}, h^{(n)}) = \|y^{(n)} - b_2(h^{(n)})\|_2^2$,
then the following equality holds true

$$-\frac{d\mathcal{J}}{d\theta} = -\lim_{\beta \to 0} \frac{1}{\beta}\left[\frac{d}{d\theta}\ell_1(x^{(n)}, h^{(n)}(\beta)) + \beta \frac{d}{d\theta}\ell_2(y^{(n)}, h^{(n)}(\beta))\right]$$

Proof. In the augmented loss function $\mathcal{L}$, for a given β, we denote the optimal h variable and the corresponding loss value using

$$h^{(n)}(\beta) := \text{argmin}_{h^{(n)}} \mathcal{L}(x^{(n)}, y^{(n)}, h^{(n)}, \beta) \quad \text{and} \quad \mathcal{L}^\star(x^{(n)}, y^{(n)}, \beta) := \mathcal{L}(x^{(n)}, y^{(n)}, h^{(n)}(\beta), \beta)\,.$$

respectively. Using these definitions, we can write

$$h^{(n)}(0) = b_1(x^{(n)}) \quad \text{and} \quad \mathcal{C}(x^{(n)}, y^{(n)}) = \frac{\partial \mathcal{L}^\star}{\partial \beta}|_{\beta=0}\,.$$

Thus, the gradient of the cost function with respect to the parameters is given by

$$\begin{aligned}
\frac{d\mathcal{J}}{d\theta} &= \frac{d}{d\theta}\frac{\partial \mathcal{L}^\star}{\partial \beta}|_{\beta=0} = \frac{\partial}{\partial \beta}\frac{d\mathcal{L}^\star}{d\theta}|_{\beta=0} \\
&= \lim_{\beta \to 0} \frac{1}{\beta}\left[\frac{d\mathcal{L}^\star}{d\theta}(x^{(n)}, y^{(n)}, \beta) - \frac{d\mathcal{L}^\star}{d\theta}(x^{(n)}, y^{(n)}, 0)\right] \\
&= \lim_{\beta \to 0} \frac{1}{\beta}\left[\frac{d\mathcal{L}^\star}{d\theta}(x^{(n)}, y^{(n)}, \beta) - 0\right]
\end{aligned}$$

and with $\ell_1(x^{(n)}, h^{(n)}) = \|h^{(n)} - b_1(x^{(n)})\|_2^2$ and $\ell_2(y^{(n)}, h^{(n)}) = \|y^{(n)} - b_2(h^{(n)})\|_2^2$ we get

$$-\frac{d\mathcal{C}}{d\theta} = -\lim_{\beta \to 0} \frac{1}{\beta}\left[\frac{d}{d\theta}\ell_1(x^{(n)}, h^{(n)}(\beta)) + \beta \frac{d}{d\theta}\ell_2(y^{(n)}, h^{(n)}(\beta))\right]$$

We prove this equivalence using the simple case of the 2 block split, extending this result to the general case is straightforward. □

6 Discussion

We proposed BlockProp with distributed optimization on conventional computer hardware in mind. It may, however, also be interesting from a neuroscience perspective. We can think of the blocks as subnetworks or brain regions that run an approximation of backpropagation, e.g. with random feedback weights [14]. Learning in these subnetworks would be coordinated with the gluing variables that follow the dynamics dictated by some form of equilibrium propagation. It could be a topic of future work to show that the gradient descent dynamics proposed in Algorithm 1 can be linked to biophysical processes or propose alternative dynamics for the gluing variables.

We empirically show that the proposed decoupling strategy can achieve comparable performance to standard backpropagation for a single hidden layer neural network and deep residual networks. However there is some difference in the performance attained, ideally we would like to make insignificant compromise on the accuracy side while being able to guarantee faster training using model parallelism. Incorporating modern empirical methods like layer normalization and dropout into our framework could be further steps in addressing this limitation. Other prominent next steps of our work include large scale experiments that make use of the decoupling to split the network using multiple nodes (potentially more than 2 blocks) along with caching strategies that enable bypassing the h-step.

The inference procedure followed during the test time is another aspect that could be reexamined. In the framework discussed, we pass the unseen data through B_2 and B_1, and generate the estimate $\hat{y} = b_2(b_1(x))$. The gluing variables learned during the training phase are ignored and this could mean that we lose of information about the inter-block inconsistency in the network due to the proposed decoupling. However, it is not straightforward to incorporate this information into the inference procedure.

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[4] B. Scellier and Y. Bengio, "Equilibrium propagation: Bridging the gap between energy-based models and backpropagation," *Frontiers in computational neuroscience*, vol. 11, 2017.

[5] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning," *Nature communications*, vol. 7, p. 13276, 2016.

[6] M. Carreira-Perpinan and W. Wang, "Distributed optimization of deeply nested systems," in *Artificial Intelligence and Statistics*, 2014, pp. 10–19.

[7] T. Frerix, T. Möllenhoff, M. Moeller, and D. Cremers, "Proximal backpropagation," *arXiv preprint arXiv:1706.04638*, 2017.

[8] G. Taylor, R. Burmeister, Z. Xu, B. Singh, A. Patel, and T. Goldstein, "Training neural networks without gradients: A scalable admm approach," in *International Conference on Machine Learning*, 2016, pp. 2722–2731.

[9] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[11] ——, "Identity mappings in deep residual networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 630–645.

[12] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[13] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein *et al.*, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine learning*, vol. 3, no. 1, pp. 1–122, 2011.

[14] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning," *Nature Communications*, vol. 7, p. 13276, Nov 2016. [Online]. Available: http://dx.doi.org/10.1038/ncomms13276

A Performance Analysis of algorithm without caching

To evaluate the performance of the presented algorithm, let us first consider the cost of W-step which involves K smaller backpropagation procedures.

$$\mathcal{C}(W_{step}) = \sum_{k=1}^{K} \mathcal{C}(\nabla_{B_k} \mathcal{J}_k^{(n)}) = \mathcal{C}(\text{backprop})$$

where $\mathcal{C}(\nabla_{B_k} \mathcal{J}_k^{(n)})$ is the cost of the k-th backpropagation. It is straightforward to prove that the cost for the smaller backpropagation procedures sum up to the cost correspoding to that of backpropagation across the entire network.

In the h-step, we compute the forward pass for initialization and make N_h gradient calls for the gluing variable updates.

$$\mathcal{C}(h_{step}) = \mathcal{C}(\text{initialization}) + \sum_{k=1}^{K} N_h \cdot \mathcal{C}(\nabla_{h_k^{(n)}} \mathcal{L}_k^{(n)}) = \mathcal{C}(\text{forward prop}) + N_h \cdot \mathcal{C}(h\text{-backprop})$$

Here $\mathcal{C}(h\text{-backprop})$ is the cost of computing $\nabla_{h_k^{(n)}}(\mathcal{L}_k^{(n)}(h_k^{(n)}, B_k, B_{k-1})) \mid_{h_k^{(n)}=\widetilde{h}_t^{(n)}}$ and taking a single gradient step on the gluing variable. Note that this cost is less than the cost of full backpropagation i.e. $\mathcal{C}(\text{h-backprop}) < \mathcal{C}(\text{backprop})$

Thus, the total cost will be

$$\mathcal{C}(\text{total}) = \mathcal{C}(W_{step}) + \mathcal{C}(h_{step}) = \mathcal{C}(\text{backprop}) + \mathcal{C}(\text{forward prop}) + N_h \cdot \mathcal{C}(\text{h-backprop})$$

$$\mathcal{C}(\text{backprop}) + \mathcal{C}(\text{forward prop}) < \mathcal{C}(\text{total}) < \mathcal{C}(\text{forward prop}) + (N_h + 1) \cdot \mathcal{C}(\text{backprop})$$

B

Lemma B.1 (Function of 2 variables). *Let us consider L a function of two variables, h and w. For, J defined as*

$$J(w) = \min_h \big[L(h, w)\big] = L(h^\star(w), w)$$

we have

$$\nabla J(w) = \frac{\partial L}{\partial w}(h^\star(w), w)$$

Proof.

$$\begin{aligned} \nabla J(w) &= \frac{d}{dw} L(h^\star(w), w)) \\ &= \frac{\partial L}{\partial h^\star}(h^\star(w), w)\frac{\partial h^\star}{\partial w}(w) + \frac{\partial L}{\partial w}(h^\star(w), w) \end{aligned}$$

However, by definition of $h^\star(w) = \arg\min_h L(h, w)$, we have $\frac{\partial L}{\partial h^\star}(h^\star, w) = 0$.

□

C Caching the activations

For a given $n \in N$, the Algorithm 1 re-initializes the gluing variables with the forward pass at every epoch using the current state of weights e.g. $h_2^{(n)} = b_1(x^{(n)})$ where b_1 corresponds to the current state of the set of weights B_1. One alternative to this re-initialization is to store for every $n \in N$ the last state of gluing variables ($h_k^{(n)\star}$) in each epoch and use $h_k^{(n)\star}$ obtained this time as the initialization in the next epoch for the corresponding instance. Note that the weights in B_k in the e-th epoch can be different from B_k in the $(e + 1)$-th epoch, hence we are solving different problems in the h-step of the two epochs, meaning that one finds $\arg\min_h \mathcal{L}_k^{i_t}(h, B_k^{(e)}, B_{k-1}^{(e)})$ and the other finds $\arg\min_h \mathcal{L}_k^{i_t}(h, B_k^{(e+1)}, B_{k-1}^{(e+1)})$.

Algorithm 2

```
for $t = 1..\infty$ do
    Sample $i_t \sim \mathcal{U}(1, \ldots, n)$
    // h-step
    Initialize the $h_k^{i_t}$ with the last known activations for this example. If there is no cache for $i_t$,
    initialize with the forward propagation
    for k=K..2 do
        // Compute the $h_k^{i_t\star}$ to get $\mathcal{J}_k^{i_t}$ (this step is executed in parallel by the K computing entities)
        Find and store $h_k^{i_t\star}$ via numerical optimization: $h_k^{i_t\star} = \arg\min_h \mathcal{L}_k^{i_t}(h, B_k, B_{k-1})$
    end for

    // W-step
    // Compute $\nabla \mathcal{J}^{i_t}$ and take a gradient step
    Perform stochastic gradient step on $B_k$: $B_k \leftarrow B_k - \gamma_{B_k} \nabla_{B_k} \mathcal{J}_k^{i_t}(B_k), \forall k$
end for
```

In practice, the h-step involving numerical optimization in Algorithm 2 can be computed infrequently (once every E epochs). If the blocks reside on different devices' memory, the advantage with caching the activations of the gluing variable h is that W-steps (block updates) can now be computed in a parallel fashion. However, this could lead to the difficulty of the gluing variables in memory being very stale and possibly hindering optimization.