Predict Globally, Correct Locally: Parallel-in-Time Optimal Control of Neural Networks

Panos Parpas [1] **Corey Muir** [2]

Abstract

The links between optimal control of dynamical systems and neural networks have proved beneficial both from a theoretical and from a practical point of view. Several researchers have exploited these links to investigate the stability of different neural network architectures and develop memory efficient training algorithms. We also adopt the dynamical systems view of neural networks, but our aim is different from earlier works. We exploit the links between dynamical systems, optimal control, and neural networks to develop a novel distributed optimization algorithm. The proposed algorithm addresses the most significant obstacle for distributed algorithms for neural network optimization: the network weights cannot be updated until the forward propagation of the data, and backward propagation of the gradients are complete. Using the dynamical systems point of view, we interpret the layers of a (residual) neural network as the discretized dynamics of a dynamical system and exploit the relationship between the co-states (adjoints) of the optimal control problem and backpropagation. We then develop a parallel-in-time method that updates the parameters of the network without waiting for the forward or back propagation algorithms to complete in full. We establish the convergence of the proposed algorithm. Preliminary numerical results suggest that the algorithm is competitive and more efficient than the state-of-the-art.

1. Introduction

As transistors get smaller the amount of power per unit volume no longer remains constant as Dennard and co-authors predicted in 1974 (Dennard et al., 1974). After over thirty years, Dennard's scaling law ended in 2005, and CPU manufacturers are no longer able to increase clock frequencies significantly (Koomey et al., 2011). The physical limitations of silicone-based microprocessors gave rise to computing architectures with many cores. The reduction in cost and wide availability of multi-core processors completely revolutionized the field of neural networks, especially for computer vision tasks. Therefore, the recent success of neural networks in many learning tasks can, in large part, be attributed to advances in computer architectures. Alternative computing technologies (e.g., quantum computers) are not likely to be available soon. Thus any future advances in more efficient training of neural networks will come from algorithms that can exploit distributed computer architectures.

Currently, the greatest obstacle in the development of distributed optimization algorithms for neural network training is the entirely serial nature of forward and backward propagation. The parameters of the neural network can only be updated after the forward propagation algorithm propagates the data from the first to the last layer and the backpropagation algorithm propagates the gradient information back to the first layer through all the layers of the network. This entirely serial nature of the training process severely hinders the efficiency of distributed optimization algorithms for deep neural networks. As a result, all the widely used frameworks for training neural networks only offer data parallelism, and the problem of layer-wise parallelism remains open.

We propose a novel distributed optimization algorithm that breaks the serial nature of forward/backward propagation and allows for layer-wise parallelism. The proposed algorithm exploits the interpretation of neural networks, and Residual Neural Networks in particular, as dynamical systems. Several authors have recently adopted the dynamical systems point of view (see, e.g., (Haber & Ruthotto, 2018; E, 2017; Chen et al., 2018; Li et al., 2017) and Section 1.1 for a discussion of related work). Reformulating the problem of Residual Neural Network (RNN) optimization as a continuous-time optimal comtrol problem allows us to model the layers of a neural network as the discretization time-points of a continuous-time dynamical system. Thus RNN training can be interpreted as a classical optimal control problem. The results of this paper hold for Residual

[1]Department of Computing, Imperial College London, London, United Kingdom [2]Department of Computing, Imperial College London, London, United Kingdom. Correspondence to: Panos Parpas <p.parpas@imperial.ac.uk>.

Neural Networks. It is possible to extend the dynamical systems view to different architectures but in this paper we focus on Residual Neural Networks (RNNs). Using the interpretation of RNN training as an optimal control problem, we decompose the neural network across time (layers) and optimize the different sub-systems in parallel. The central insight of this paper is that if the state and co-state (adjoint) of the dynamical system are approximately known in N intermediate points, then we can parallelize the time dimension of the system and perform N forward/backward propagations in parallel. This description justifies the name *"parallel-in-time"* method because we parallelize across the time dimension of the optimal control model. A significant challenge is to produce approximately correct state and co-state information for the optimal control problem. We address this problem by using a coarse discretization of the problem with a phase we call *global prediction* phase. The global prediction phase is followed by a *local correction* phase that attempts to improve the predicted optimal state and co-states of the control model solution.

1.1. Related Work & Contributions

Layer-wise optimization of NNs. Several authors have identified the limitations that backpropagation imposes on distributed algorithms. As a result, many approaches have been proposed including ADMM, block-coordinate descent (Zeng et al., 2018) delayed gradients(Huo et al., 2018), synthetic gradients(Jaderberg et al., 2017), proximal (Lau et al., 2018), penalty based methods(Carreira-Perpinan & Wang, 2014; Huo et al., 2018), and alternating minimization methods (Choromanska et al., 2018). Our method (especially the local correction phase) is related to the synthetic gradient approach in (Jaderberg et al., 2017). The major differences between synthetic gradients of (Jaderberg et al., 2017) and our approach are that we exploit the dynamical-systems view and a multilevel discretization scheme to approximate the co-state (adjoint) variables quickly. We also establish the convergence of our method with weaker conditions than in (Jaderberg et al., 2017). In Section 5 we also show that our method outperforms even an improved version of the synthetic gradient method.

The dynamical systems point of view. Several authors have formulated neural network training as an optimal control problem in continuous time. For example, the authors in (Haber & Ruthotto, 2018) have adopted this point of view to develop more stable architectures for neural networks. In (Li et al., 2017) the authors used this approach to propose a maximum-principle based method as an alternative to backpropagation. The work in (Li et al., 2017) also allows layer-wise optimization, but unfortunately, it was slower than stochastic gradient descent (SGD). In (Chen et al., 2018) the authors proposed to use classical ODE solvers to compute the forward and backward equations associated with NN training. We note that none of the existing approaches used the dynamical systems viewpoint to allow more parallelization in the optimization process. A recent exception is the method in (Günther et al., 2018). In (Günther et al., 2018) the authors used the so-called parareal method to parallelize across the time (layer) dimension of neural networks. However, the parareal method is known to be problematic for non-linear systems (Gander & Hairer, 2008) and the efficiency reported by the authors is lower than our method and other methods such as synthetic and delayed gradients.

Contributions. We exploit coarse and fine discretizations of continuous systems to compute predictions for optimal states/co-states of optimal control problems. The state/co-state predictions allows us to develop a layer-wise (parallel-in-time) optimization algorithm for RNNs (Section 3). We exploit the structure of the neural network training problem to show that the value function of the problem can be decomposed (Lemma 2.1). We also discuss the relationship between stochastic gradient descent, backpropagation and the co-state (adjoint) variables in optimal control (Lemma 2.2). We establish appropriate conditions under which the proposed algorithm will converge (Theorems 4.1 & 4.2). We report encouraging numerical results in Section 5.

2. Optimal Control & Residual Neural Networks

The focus of this paper is Residual Neural Networks (RNNs) because they can achieve state-of-the-art performance in many learning tasks and can be reformulated as continuous-time optimal control problems. This reformulation offers several benefits such as stable classifiers, memory efficient algorithms and further insights into how and why they work so well (see, e.g., (E, 2017; Li et al., 2017; Haber & Ruthotto, 2018; Chen et al., 2018)). In this section, we review the dynamical systems viewpoint of RNN training. We review the links between the co-state (dual) variables of optimal control problems and the most widely used method to train neural networks, namely stochastic gradient descent and backpropagation (see Lemma 2.2). In Lemma 2.1 we make a simple observation that enables the decomposition of the optimal control problem across different initial conditions. This observation greatly simplifies the proof for a randomized version of our algorithm.

For the origins, and advantages of the formulation below we refer to (E, 2017; Haber & Ruthotto, 2018), where training with residual neural networks is reformulated as the

following optimal control problem.

$$\begin{aligned} V_0([X]) = \min_U \sum_{i=1}^m \Phi_i(X_T^i) + \int_0^T R(U_t)dt \\ \dot{X}_t^i = f_t(X_t^i, U_t) \\ X_0^i = x_i \; i = 1, \ldots, m. \end{aligned} \tag{1}$$

Where $\Phi_i : \mathbb{R}^n \to \mathbb{R}$ is a data fidelity term, and $R : \mathbb{R}^d \to \mathbb{R}$ is a regularizer. We note that the label for the i^{th} data-point has been subsumed in the definition of Φ_i. The regularizer function could also be a function of the state (X) and time t. Regularization is typically used to prevent over-fitting and therefore applied to the parameters (U) of the network. Hence we assume that the regularizer is only a function of the control parameter U. The notation $[X] = [x_1, \ldots, x_m]^\top \in \mathbb{R}^{md}$ is used to denote the initial conditions of the dynamics in (1). We use $[X]_i \in \mathbb{R}^d$ to denote the i^{th} element in $[X]$. The function $f : \mathbb{R}_+ \times \mathbb{R}^d \times \mathbb{R}^u \to \mathbb{R}^d$ describes the activation function applied at time t. For example, for the parameters $U = [W, b]$, the activation function may be defined as $\tanh(XW + b)$, where $\tanh$ is applied element wise. For the usual activation, regularization and data fidelity terms used in practice, the optimal control problem above is well defined (see also (Bardi & Capuzzo-Dolcetta, 1997)).

The function $V : \mathbb{R}^{md} \to \mathbb{R}$ defined in (1) is called the *value function* of the problem. The value function, and its derivatives, will play a crucial role in this paper, and below we make a simple observation regarding the dimensionality of the value function. Using the principle of optimality (Bertsekas, 2017) we can rewrite the original model in (1) as follows,

$$\begin{aligned} V_0([X]) = \min_u \int_0^s R(U_t)dt + \sum_{i=1}^m V_s([X_s]). \\ \dot{X}_t^i = f_t(X_t^i, U_t) \\ X_0^i = x_i \; i = 1, \ldots, m. \end{aligned} \tag{2}$$

Where $V_s([X])$ is defined as follows,

$$\begin{aligned} V_s([X]) = \min_u \sum_{i=1}^m \Phi(X_T^i) + \int_s^T R(U_t)dt \\ \dot{X}_t^i = f_t(X_t^i, U_t) \\ X_s^i = [X]_i \; i = 1, \ldots, m. \end{aligned} \tag{3}$$

We take advantage of the structure of the problem to show that the value function can be written as a sum of identical functions from $\mathbb{R}^d$ to $\mathbb{R}$.

Lemma 2.1. *Let $V_s([X])$ denote the optimal value function of* (3)*. Then for any $0 \le s \le T$, there exists a function $V_s : \mathbb{R}^d \to \mathbb{R}$ such that: $V_s([X]) = \sum_{i=1}^m V_s(x^i)$.*

The result above simplifies the problem dramatically. It allows us to approximate the md-dimensional value function with a sum of d-dimensional functions.

2.1. Discretization

To approximate the problem in (1) we use a discretization scheme to obtain the following finite dimensional optimization problem.

$$\begin{aligned} \min_U \mathcal{J}(U) = \sum_{i=1}^m \Phi_i(X_{T_\delta}^i) + \sum_{j=0}^{T_\delta - 1} R_j^\delta(U(t_j)) \\ X^i(t_{j+1}) = f_j^\delta(X^i(t_j), U(t_j)), 0 \le j \le T_\delta - 1 \\ X_0^i = x_i, \; i = 1, \ldots, m. \end{aligned} \tag{4}$$

Where we used a discretization parameter δ, such that with $\delta = T/T_\delta$ we obtain the discrete-time optimal control problem in (4) with T_δ time-steps. We use the notation f^δ to denote an explicit discretization scheme of the dynamics in (1). For example, if an explicit Euler scheme is used then $f_j^\delta(X, U) = X + \delta f_j(X, U)$. When we use the simple explicit Euler scheme, with $\delta = 1$ the formulation above reduces to the standard RNN architecture. The deficiencies in terms of numerical stability of the explicit Euler scheme are well known in the numerical analysis literature (Butcher, 2016). In (Haber & Ruthotto, 2018) the authors argue that the use of the explicit Euler scheme for RNNs explains numerical issues related to the training of NNs such as exploding/vanishing gradients and classifier instability. To resolve these numerical issues the authors in (Haber & Ruthotto, 2018) propose to use stable discretization schemes. We follow the same line of reasoning, and show that a further advantage of appropriately defined discretization schemes is that they can be used to design convergent distributed algorithms to solve (4). Before we describe our approach we review serial-in-time methods and explain why it is difficult to design efficient distributed algorithms for (4).

Multilevel Discretization. Since (1) is a continuous time model we can approximate it using different levels of discretization. To keep the notation simple and compact we assume that we want to solve (1) using a step-size of δ. This gives rise to the model in (4), with T_δ time-steps/layers. With our terminology the number of time-steps refers to the number of layers in the NN. We use the terms time-steps and layers interchangeably. We call the model with the a step-size of δ the *fine model*. Later on we will take advantage of a coarse discretization of (1), and we refer to the resulting model as the *coarse model*. The time-step parameter of the coarse model is denoted by Δ. We use T_Δ to denote the number of time-steps in the coarse model. The coarse model has less time-steps than the fine model, and it is therefore faster to optimize. As a practical example, suppose we are interested in the solution of (1) using a fine model with 1024

steps (layers). Suppose we also use a coarse model with 64 steps/layers (for example). A forward/backward pass of the coarse model will roughly be 16 times faster. The use of multiple levels of discretization is a well known technique in the solution of optimal control problems, and has its origins in the multigrid literature(Briggs et al., 2000). From the multigrid literature we will use the idea of interpolation operators in order to transform a trajectory from the coarse discretization grid to the fine grid (see (Briggs et al., 2000; Haber & Ruthotto, 2018))

2.2. Serial-in-Time Optimization

The most popular optimization algorithm for (4) is batch stochastic gradient descent(Curtis & Scheinberg, 2017; Bottou et al., 2018). Stochastic first order methods compute the gradient of the objective function in (4) with respect to the parameters U using backpropagation. In optimal control algorithms, the same gradient information is computed by solving backwards the *co-state* equations associated with the dynamics in (4). The connections between co-states, adjoints, Lagrange multipliers and backpropagation are well known (see e.g. (Baydin et al., 2018; LeCun et al., 1988)). We adopt the language used in scientific computing, and call the forward propagation the *forward solve*, and the backward propagation the *backward solve*. The forward solve is specified in Algorithm 1. This algorithm plays the same role as forward propagation of conventional NN algorithms. The difference is that we do not rely on explicit Euler discretization but *we use a discretization scheme with a forward propagator that is stable and consistent with the dynamics in* (1). We use stable in the sense used in numerical analysis (see (Butcher, 2016)) and consistent in the sense used in optimization (see (Polak, 1997)). The backwards solve is specified in Algorithm 2. The purpose of the backward solve is to generate the information needed to compute the gradient of the objective function of (4) with respect to the controls U. We use a stable and consistent scheme for the backward solve too. After the forward and backward equations are solved, the information generated is used in some *algorithmic mapping* denoted by $\mathcal{A}$. This mapping generates a (hopefully) improved set of controls. A full iteration of a serial (in time) stochastic first order algorithm consists of a forward solve, a backward solve, followed by an update for U. We state the standard serial-in-time algorithm in Algorithm 3. In order to make our terminology more concrete we show that for a specific choice of $\mathcal{A}$, the procedure above reduces to the standard stochastic gradient method with backpropagation.

Lemma 2.2. *Suppose that the algorithmic mapping in* (5) *is defined as follows,*

$$\mathcal{A}(U,X,P) = U - \eta\left(\langle \nabla_u f_t^\delta(X,U), P\rangle + \nabla_u R(u)\right)$$

then Algorithm 3 generates the same iterations as batch stochastic gradient descent with a learning rate of η.

3. A Parallel-in-Time Method

It is challenging to parallelize optimization algorithms for the model in (4) because the backward solve cannot start before the forward solve finishes. Moreover, it is not possible to parallelize the forward solve because $X_{t+\delta}$ cannot be computed before X_t is computed. Similarly $P(t+\delta)$ must be computed before $P(t)$ in the backward solve. Thus most algorithms only allow for data parallelism (e.g. across batches or in the calculation of f), but not across time/layers (see Section 1.1 for some exceptions).

Algorithm 1 `Forward`$(\delta, X_s, t_0, t_1, \{U(t)\}_{t=t_0}^{t=t_1})$

$t \leftarrow t_0,\ X(t) = X_s$
while$(t \le t_1)$ **do**
$X(t+\delta) = f_t^\delta(X(t), U(t)),\ t \leftarrow t+\delta$
return $X(t), t_0 \le t \le t_1$

Algorithm 2 `Backward`$(\delta, P_e, t_0, t_1, \{U(t), X(t)\}_{t=t_0}^{t=t_1})$

$t \leftarrow t_1,\ P(t_1) = P_e$
while$(t \ge t_0)$ **do**
$P(t-\delta) = -\langle \nabla_x f_t^\delta(X(t), U(t), P(t))\rangle, t \leftarrow t-\delta$
return $P(t), t_0 \le t \le t_1$

Algorithm 3 `Serial-in-time`$(\delta, 0, T_\delta, \{U^0(t)\}_{t=t_0}^{t=T_\delta-1})$

Let $X^k(0)$ be a random sample from $[X]$.
$X^k(t)$ =`Forward`$(\delta, X^k(0), 0, T_\delta, U^k(t)), 0 \le t \le T_\delta$
$P^k(T_\delta) = \nabla_x \Phi(X^k(T_\delta))$
$P^k(t)$=`Backward`$(\delta, P^k(T), 0, T, U^k(t)), 0 \le t \le T_\delta$
`Update control` for $0 \le t \le T_\delta - 1$

$$U^{k+1}(t) = \mathcal{A}(U^k(t), X^k(t), P^k(t+\delta)). \quad (5)$$

Suppose (somehow) we had an approximately optimal trajectory at some intermediate point $X^\star(s)$ and the corresponding co-state variable $P^\star(s)$. Suppose we also had two processors, Processor A and B, then we could potentially halve the cost of a full iteration of Algorithm 3. We achieve this impressive reduction in time by using Processor A to do a backward solve from $t = s$ to $t = 0$, followed by a forward solve from $t = 0$ to $t = s$. In parallel, processor B is able to do a forward solve from time $t = s$ to time T, followed by a backward solve from time $t = T$ to time $t = s$. Thus, with two processors we can (potentially) halve the cost of a full iteration of any stochastic first-order method for (4). In reality, the reduction in time due to the extra processing power will not be halved (due to communication). The difficult issue is how to compute the approximately optimal intermediate points $X^\star(s)$ and $P^\star(s)$? To address this question we proceed using two phases: a prediction phase,

followed by a correction phase. To be more precise, we first construct a coarse discretization of (1) which we solve approximately using Algorithm 3. We call this phase the *global prediction* phase because it generates global information regarding the optimal trajectory. The global prediction phase generates useful information but it is not exact. To correct the prediction we have a second phase we call the *local correction* phase. The local correction is the time-parallel phase where we solve discretized versions of (2) to find better initial conditions for (3) in parallel. Below we explain these two phases in turn.

3.1. Global Prediction Phase

In the global prediction phase we construct an approximate solution of (1) using a coarse discretization scheme. Our main assumption regarding the discretization scheme is that it provides a consistent approximation in the sense of (Polak, 1997). We use a large step-size $\Delta > \delta$, to approximate the model in (1) with a finite dimensional optimization problem. We use a standard algorithm (e.g. Algorithm 3) to perform some iterations on the coarse model. We call this phase the *global prediction* phase because we use the coarse model to quickly generate global information about the solution of the model in (1). In this context, the global information is contained in the forward trajectory (good initial conditions to initialize the local correction phase), and the backward trajectory (sensitivity information regarding the initial conditions at time s). We also obtain a good initial point for the parameters U. The coarse model is only used to generate predictions regarding the optimal state and co-states of the control problem. The predictions are corrected in parallel using the local correction phase described below.

3.2. Local Correction Phase

Using the information generated from the global prediction phase we split the original model into two sub-systems. The first subsystem is responsible for identifying the optimal initial conditions for the second subsystem. The second subsystem receives the initial conditions from the first subsystem and is responsible for solving the classification problem. The second subsystem also passes information back to the first subsystem in the form of sensitivity information (from the co-state variables).

The local correction phase is shown in Algorithm 4 for the case when the original model is decomposed into two subsubsystems. The left column in Algorithm 4 describes the steps to optimize the first subsystem, and the right column describes the steps for the second subsystem. Note that the two sub-systems are solved in parallel. This is why we call the algorithm *parallel-in-time*. Our method computes the optimal parameters for time $t > s$ without waiting for information from the past $t \leq s$.

We first describe the work that Processor A performs (left column) on the first sub-system i.e. the optimization of (4) from time $t = 0$ to $t = s$ using a time-step of δ. To start a backward solve at iteration k from time s we need $X^k(s)$, $P^k(s)$ and $U^k(s)$. In the first iteration of the local correction phase ($k = 0$), we compute $X^0(s)$ and $U^0(s)$ by simple interpolation from the coarse model. In subsequent iterations $X^k(s)$ and $U^k(s)$ are also available to the local correction phase of the first sub-system. This is because the first subsystem has all the information required from time 0 to time s. Unfortunately, the co-state variable $P^k(s)$ is not available at time s because to compute it we need to perform a backward solve from time T to time s. Using coarse information only is not sufficient to build a good approximation for the co-state variables. Instead, we approximate $P^k(s)$ from state/co-state observations from the prediction phase, along with state/co-state observations collected from the second sub-system. In the first iteration of the local correction phase we only have the observations from the prediction phase to approximate $P^0(s)$ (information from the second sub-system is not yet available). We use $\mathcal{I}_k$ to denote the state/co-state pairs observed by the first sub-system at iteration k of the local correction phase. The prediction phase, after H iterations of the coarse model, produces the following information $\mathcal{I}_0 = \{(X^i(s), P^i(s)), i = 0, \ldots, H-1\}$. We use the state/co-state pairs observed so far to (approximately) solve the following regression problem,

$$\min_{A,B} L[\mathcal{I}_0] = \sum_{(X^i(s),P^i(s))\in\mathcal{I}_0} \|AX_i(s)+B-P_i(X_i(s))\|^2. \tag{6}$$

Using the solution of the linear regression problem above we can approximate $P(s) = P(X(s))$ at any state $X(s)$ as $\hat{P}(s) \approx A^\star X(s)+B^\star$ (where $A^\star, B^\star$ are approximate solutions to the regression problem above). After the backward solve finishes, we update the controls, do a forward-solve and pass the state $X_k(s)$ to the second subsystem. We next discuss how to solve the problem from time s to time T, and how to update the information set $\mathcal{I}_0$.

In the first iteration of the local correction phase, Processor B (right column) in Algorithm 4, receives the approximate state $X^0(s)$, and controls $\{U^0(t)\}_{t=s}^T$ from the global prediction phase. Starting from $X^0(s)$ it performs a forward solve, a backward solve and updates its controls. After the backward solve finishes, Processor B passes sensitivity information in the form of the co-state variables P_s^0 to Processor A. Processor A then sets $\mathcal{I}_1 = \mathcal{I}_0 \cup (X_0(s), P_0(s))$. The same steps are then repeated by both processors. We use the notation $L[\mathcal{I}_k]$ when the regression problem in (6) is solved with the information set $\mathcal{I}_k$. With a slight abuse of notation we write $\widehat{P}^k(s) = L[\mathcal{I}_k]$ to denote the approximate co-state information P^k obtained using the solution of the regression problem in (6).

Remark 1 (More than two sub-systems)**.** *We described the*

algorithm using only two sub-systems. To use more than two sub-systems we observe that the second sub-system is a standard optimal control problem from time s to time T. We can therefore use the same procedure we described in this section to divide the second subsystem into two. We can then continue to divide the system to as many sub-systems as required.

Remark 2 (Mini-batches and asynchronous computation)**.** *To simplify the notation we did not describe the algorithm using mini-batches. In order to use mini-batches we just change Algorithms 1 and 2 to use mini-batches. We then change the notation so that $X^{k,i}(t)$ denotes the state at iteration k, batch i at time t (with similar notation for the control and co-state variables). Finally, Algorithm 4 has a synchronization step after each sub-system completes a single iteration. This is how the algorithm was analyzed and implemented and we leave the asynchronous version for future work.*

Algorithm 4 Parallel-in-Time Optimization

Global Prediction: Use Algorithm 3 to make m iterations of (4) with step-size Δ. Initialize with $\{U_t^0, P_t^0, X_t^0]\}_{t=0}^T$ from global prediction phase.

Processor A	Processor B
Backward solve:	**Forward solve:**
$\widehat{P}_s^k = \mathcal{L}[\mathcal{I}_k]$	$\texttt{Forward}(\delta, X_s^k, s, T_\delta, U_t^k)$
$\texttt{Backward}(\delta, \widehat{P}_s^k, 0, s, U^k)$	**Backward solve:**
Update:	$P_{T_\delta}^k = \nabla_x(X_{T_\delta}^k)$
$U_t^{k+1} = \mathcal{A}(X_t^k, \widehat{P}_t^k)$	$\texttt{Backward}(P_{T_\delta}^k, T_\delta, s, U^k)$
Forward solve:	**Update:**
$\texttt{Forward}(\delta, X_0^k, 0, s, U_t^k)$	$U_t^{k+1} = \mathcal{A}(X_t^k, P_t^k)$
Synchronization: Send X_s^k to Processor B.	**Synchronization:** Send P_s^k to Processor A.

4. Convergence Analysis

In this section we summarize the theoretical convergence results for Algorithm 4. All proofs and technical lemmas appear in the supplementary material. Algorithm 4 is quite general because we do not specify the algorithmic mapping in the update step, or the discretization scheme used to derive (4). In order to keep the convergence analysis as close to the numerical implementation as possible we use the algorithmic mapping specified in Lemma 2.2. It is possible to establish similar convergence results for other schemes too. For example, because our method can decompose a large network into smaller sub-networks, it might be possible to use second-order methods. We leave such refinements of the scheme to future work. Moreover, as discussed in Remark 2 the fact that the algorithm has a synchronization step simplifies the analysis. This simplifying assumption allows us to analyze the algorithm as if it was run on a single processor.

The starting point of our analysis is the result in Lemma 2.1. It allows us to compute the optimal value function as the sum of m $d-$dimensional functions as opposed to a single $md-$dimensional function. This is an important insight, because we know from the maximum principle (Fattorini, 1999) that $P_i^\star(t) = -\nabla V_t(X_i^\star(t))$ at the optimal solution (where i denotes the ith initial condition in (1)). Since V is a function from $\mathbb{R}^d$ to $\mathbb{R}$, it follows that the adjoint is function from $\mathbb{R}^d$ to $\mathbb{R}^d$. In fact, in the proof of Lemma 2.2 we show that,

$$P_t^{k,\delta} = -\nabla_{x_t}\Phi(X_T^k).$$

The equation above explains why the co-state variables provide sensitivity information for the first sub-system. The main additional assumption we need to prove the convergence of Algorithm 4 (beyond the assumptions needed for any stochastic first-order methods) is that there exists an $\epsilon_p > 0$ such that,

$$\|\widehat{P}^\delta(t) - P^\delta(t)\| \leq \epsilon_p \eta \|\widehat{P}^\delta(t)\|. \tag{7}$$

We note that, at least in principle, such an ϵ_p is guaranteed to exist. The more interesting question is, of course, whether this constant is small or not. The theoretical results below assume an arbitrary ϵ_p. In practice, we found that this constant is small. In Section 5 we conjecture that this constant is small because we use data for several levels of discretization to construct $\widehat{P}$. As was mentioned in the introduction, the regression step in the local correction phase has similarities with the the so called synthetic gradient method proposed in (Jaderberg et al., 2017). The conceptual differences between synthetic gradients and the proposed method were detailed in 1.1). However, the convergence analysis and assumptions are different too. For example, in (Jaderberg et al., 2017) the authors assume that $\|S_g - \nabla J\| \leq \epsilon$ (where S_g is the, so called, synthetic gradient). Our assumptions are weaker. In addition, the convergence analysis in (Jaderberg et al., 2017) is for gradient descent and not for stochastic gradient descent. Our first convergence result is for the case where the full gradient is used.

Theorem 4.1 (Reduction in objective function)**.** *Suppose that, $\eta_k \leq \frac{2}{2\epsilon_p + L}$. Then, $\mathcal{J}(U^{k+1}) \leq \mathcal{J}(U^k)$ and in particular,*

$$\mathcal{J}(U^{k+1}) - \mathcal{J}(U^k) \leq -\sum_{j=0}^{T_\delta - 1} \Bigg(\theta_1 \|\partial_j f_k^\delta\|^2 \|\widehat{P}_{j+1}^k\|^2 + \theta_2 \|\partial_j f_k^\delta\| \|\widehat{P}_{j+1}^k\| \|\partial_j R_k^\delta\| + \theta_3 \|\partial_j R_k^\delta\|^2 \Bigg),$$

where the scalars $\theta_1 \leq \theta_2 \leq \theta_3$ are positive and depend only on L (the Lipschitz constant of $\mathcal{J}$) and ϵ_p.

We note that if $\epsilon_p = 0$, then Theorem 4.1 gives exactly the same result as the gradient descent method for (4). Our next

result deals with the case when the algorithm is run using mini-batches.

Theorem 4.2. *Suppose that the step-size in Algorithm 4 satisfies the following conditions,*

$$\sum_{k=1}^{\infty} \eta_k = \infty, \ \sum_{k=1}^{\infty} \eta_k^2 < \infty.$$

Then, $\lim_{M\to\infty} \frac{1}{H_M} \mathbb{E}\left(\sum_{k=1}^{M} \eta_k \|\nabla J(U^k)\|^2\right) = 0$*, where* $H_M = \sum_{k=1}^{M} \eta_k$.

We note that for the theorem above to hold some additional restrictions on ϵ_p are required (see the on-line supplement for additional discussion).

5. Numerical Experiments

In this section, we report preliminary numerical results for Algorithm 4. This paper aims to establish a framework for time-parallel training of neural networks, and not to report on extensive numerical experiments. Still, it is essential to show that the algorithm is promising. We implemented the algorithm on a standard computer with a quad-core processor and 8GB of RAM. We decomposed the system into only two sub-systems. We believe that the performance of our algorithm will improve with more than two sub-systems, and when ported to systems with a large number of cores. Below we report results from three datasets. The first two datasets (co-centric ellipse, and swiss roll) are relatively low dimensional. To save space, we refer to (Haber & Ruthotto, 2018) for a description of these two datasets. The third dataset is the well known MNIST dataset. For large-scale problems, the cost of approximately solving a coarse model becomes a less significant part of the overall cost. Therefore, we believe that the efficiency of our method will improve for large-scale problems. Indeed we see that for the MNIST dataset our algorithm achieves good speed-ups even for relatively shallow neural networks. But such extensive numerical experiments are beyond the scope of the current paper. The code is available from the authors' GitHub page (and in the on-line supplement during the review phase).

In Section 1.1 we mentioned several approaches for layer (time) wise parallelization of neural networks. Because the synthetic gradient method of (Jaderberg et al., 2017) is closely related to Algorithm 4 we only compare against a stable version of the synthetic gradient approach. We also compare different variants of our method against the data-parallel implementation of SGD in Pytorch 0.4.1. It will be interesting to compare all the different approaches described in Section 1.1, but this is not the aim of this paper. In (Huo et al., 2018) the authors compared several layer-wise parallelization frameworks and concluded that their method, when tested on ResNet architectures on the CIFAR datasets, outperformed others and achieved speedups of 15% to 50%, without significant loss of accuracy. Below we report similar results, but with higher parallel efficiency. In (Huo et al., 2018) the authors compared their method against the synthetic gradient method in (Jaderberg et al., 2017)and found that their method significantly outperforms synthetic gradients. Their implementation of synthetic gradients was based on network architectures that were shown to be unstable in (Haber & Ruthotto, 2018). So in our view, more careful numerical experiments are needed in order to decide the merits of the different approaches. However, these are beyond the scope of this paper.

Discretization schemes. We considered two discretization schemes to derive the model in (4). The first one is based on an explicit Euler scheme and the second on symplectic integration using the Verlet method. We chose the explicit Euler scheme because this scheme gives rise to the standard ResNet architecture. We chose the Verlet scheme because it was shown to perform well in previous works (Haber & Ruthotto, 2018). We also tested our method with and without the global prediction phase. When the global prediction phase is used we refer to our method as the multilevel parallel-in-time algorithm. When we do not use the global prediction phase then we call our method the single-level parallel-in-time method. The regression step in our single-level, parallel-in-time method is similar to the decoupled neural interfaces method with synthetic gradients of (Jaderberg et al., 2017). However, our single-level method is implemented with a stable discretization scheme. It is shown below that the discretization scheme makes a significant impact in the parallel efficiency and accuracy of the method. The coarse model we used are exactly half the size of the fine model (e.g. if the fine model has 64 layers (steps) then the coarse model is constructed with 32 layers (steps)).

Accuracy of serial and parallel-time method. The first observation from our results is that the the accuracy of our method (especially with the Verlet discretization scheme) is similar to the accuracy obtained with the data-parallel SGD method. It is clear from Figures 5, 6 and 7 (in the supplement) that the parallel-in-time method produces results with similar accuracy as Stochastic Gradient Descent (SGD).

Speedup and parallel efficiency results. Figure 1 summarizes the speed-up obtained from our method against the data-parallel implementation of SGD in Pytorch. For our parallel-in-time method we include the time needed to solve the coarse (when used) and fine models in the speed-up calculations. We report results for the explicit discretization scheme without using the global prediction phase i.e. using a single-level discretization, and in Figure 1 we refer to this method as the single-level Euler method. We also report results using the multilevel scheme (i.e. using the global prediction phase) and the Verlet discretization scheme. In

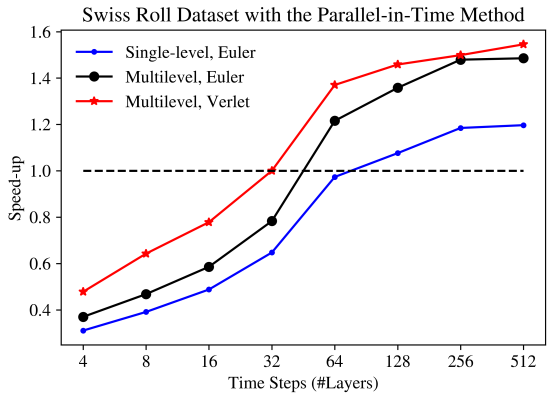

Figure 1. Speed-up for the Swiss-Roll dataset over SGD.

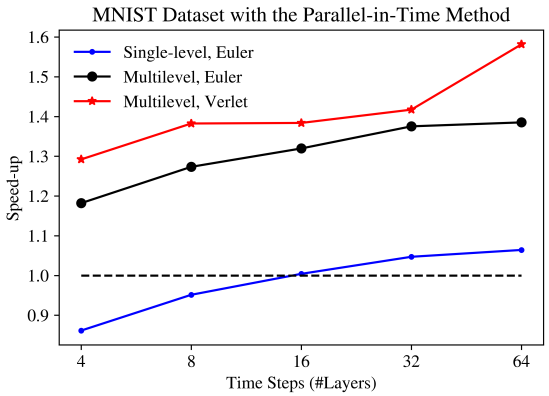

Figure 2. Speed-up for the MNIST dataset over SGD.

Figure 1 we refer to this method as the multi-level Verlet method. We observed similar speed-ups for both the ellipse and Swiss-Roll and so to save space we only report the results from the Swiss-Roll dataset in Figure 1 (see Figure 4 in the on-line supplement for the ellipse dataset). From Figure 1 we see that for relatively small data-sets (e.g. the ellipse and Swiss-Roll data-sets) there is a cut-off point (around 32 to 64 layers) after which our method is faster than the data parallel implementation. Since we only use two processors, we observe that, for deep networks, our method achieves an efficiency of about 75%. The efficiency of our algorithm is substantially better than the speed-up efficiency of 50% reported in (Huo et al., 2018). Our results compare even more favorably with speed-up efficiencies of 3-4% reported in (Günther et al., 2018) for an alternative parallel-in-time method.

In Figure 2 we summarize the results for the MNIST dataset. For the MNIST dataset we see that our method is faster than the SGD even for relatively shallow networks (4 layers). From these results we see that our method achieves much better speed-ups on MNIST than other speed-ups reported in the literature. Moreover, the efficiency of over 75% for the MNIST dataset suggest that the communication overheads of our method are small. These results validate our claim that our method will have an advantage over existing methods for larger models. The reason is that, for large models, the time spent solving the coarse model is a small proportion of the total solution time.

Increased stability due to the global prediction phase. To test the impact of the global prediction phase using the coarse model we report the mean-square errors from the regression step in the backward solve of Algorithm 4. When the global prediction step is not used our method is similar to the synthetic gradient method from (Jaderberg et al., 2017). We see from Figure 3 that when the global prediction phase is not used the mean-square error of the regression step varies significantly in the first 20 iterations before converging to a non-zero value. When the global prediction phase is used, we can see from Figure 3 (left y-axis) that the MSE is an order of magnitude lower and eventually converges to zero. These results explain why our method is so efficient. These results also provide empirical validation for the assumption in (7) required to prove the convergence of our method. The results in Figure 3 are for the Swiss-Roll dataset with 512 layers using the Verlet discretization scheme. Similar behavior was observed in the other models.

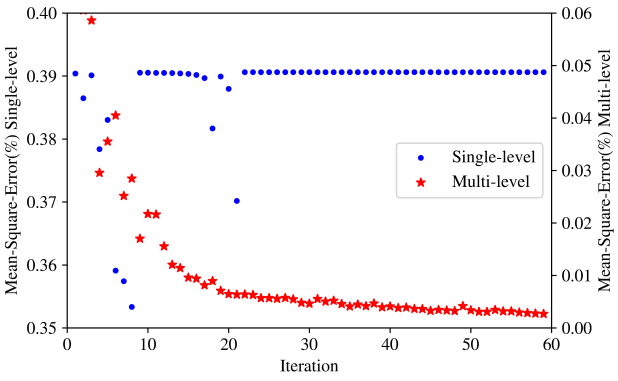

Figure 3. Mean Square Errors of regression step in Algorithm 4

6. Conclusions

We proposed a novel parallel-in-time distributed optimization method for neural networks. The method exploits the dynamical systems view of residual neural networks to parallelize the optimization process across time (layers). We discussed how to take advantage of multilevel discretization schemes in order to predict the optimal states and co-states of the control model. We established the convergence of our method. Our initial numerical results suggest that the proposed method has the same accuracy as Stochastic Gradient Descent, reduces computation time significantly and achieves higher parallel efficiency than existing methods. The method can be improved in several ways including an asynchronous implementation, using more than two dis-

cretization levels and decomposing the original network to several components. More detailed numerical experiments are needed to understand the behavior of the method for large datasets, but the initial efficiency results are extremely encouraging.

References

Bardi, M. and Capuzzo-Dolcetta, I. *Optimal control and viscosity solutions of Hamilton-Jacobi-Bellman equations*. Systems & Control: Foundations & Applications. Birkhäuser Boston, Inc., Boston, MA, 1997.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

Bertsekas, D. P. *Dynamic programming and optimal control. Vol. I*. Athena Scientific, Belmont, MA, fourth edition, 2017. ISBN 978-1-886529-43-4; 1-886529-43-4; 1-886529-08-6.

Bottou, L., Curtis, F. E., and Nocedal, J. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

Briggs, W. L., McCormick, S. F., et al. *A multigrid tutorial*, volume 72. Siam, 2000.

Butcher, J. C. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

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

Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. Neural ordinary differential equations. *arXiv preprint arXiv:1806.07366*, 2018.

Choromanska, A., Kumaravel, S., Luss, R., Rish, I., Kingsbury, B., Tejwani, R., and Bouneffouf, D. Beyond backprop: Alternating minimization with co-activation memory. *arXiv preprint arXiv:1806.09077*, 2018.

Curtis, F. E. and Scheinberg, K. Optimization methods for supervised machine learning: From linear models to deep learning. In *Leading Developments from INFORMS Communities*, pp. 89–114. INFORMS, 2017.

Dennard, R., Gaensslen, F., Rideout, V., Bassous, E., and LeBlanc, A. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

E, W. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5(1):1–11, Mar 2017. ISSN 2194-671X. doi: 10.1007/s40304-017-0103-z.

Fattorini, H. O. *Infinite-dimensional optimization and control theory*, volume 62 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 1999. ISBN 0-521-45125-6. doi: 10.1017/CBO9780511574795.

Gander, M. J. and Hairer, E. Nonlinear convergence analysis for the parareal algorithm. In *Domain decomposition methods in science and engineering XVII*, pp. 45–56. Springer, 2008.

Günther, S., Ruthotto, L., Schroder, J., Cyr, E., and Gauger, N. Layer-parallel training of deep residual neural networks. *arXiv preprint arXiv:1812.04352*, 2018.

Haber, E. and Ruthotto, L. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, 2018.

Huo, Z., Gu, B., Yang, Q., and Huang, H. Decoupled parallel backpropagation with convergence guarantee. *arXiv preprint arXiv:1804.10574*, 2018.

Jaderberg, M., Czarnecki, W. M., Osindero, S., Vinyals, O., Graves, A., Silver, D., and Kavukcuoglu, K. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1627–1635. JMLR. org, 2017.

Koomey, J., Berard, S., Sanchez, M., and Wong, H. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33 (3):46–54, 2011.

Lau, T. T.-K., Zeng, J., Wu, B., and Yao, Y. A proximal block coordinate descent algorithm for deep neural network training. *arXiv preprint arXiv:1803.09082*, 2018.

LeCun, Y., Touresky, D., Hinton, G., and Sejnowski, T. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pp. 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.

Li, Q., Chen, L., Tai, C., and Weinan, E. Maximum principle based algorithms for deep learning. *The Journal of Machine Learning Research*, 18(1):5998–6026, 2017.

Polak, E. *Optimization*, volume 124 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 1997. ISBN 0-387-94971-2. doi: 10.1007/978-1-4612-0663-7. Algorithms and consistent approximations.

Zeng, J., Lau, T. T.-K., Lin, S., and Yao, Y. Block coordinate descent for deep learning: Unified convergence guarantees. *arXiv preprint arXiv:1803.00225*, 2018.

7. Supplementary Material for: Predict Globally, Correct Locally: Parallel-in-Time Optimal Control of Neural Networks

7.1. Notation

In this section we briefly define our notation. We use $X_\delta^{\xi,k}(t)$ represent the state at time t, iteration k, batch ξ, with discretization parameter δ. When it is clear from context we drop ξ and δ and write X_t^k instead. When it is relevant to identify or sum over the different time-steps we use the notation

$$X_j^k = X^k(t_j).$$

We use X^k to denote the vector form of X_j^k. We use the same conventions for the exact co-state variables (P_t^k), the approximate co-states $(\widehat{P}_t^k)$ and the control parameters/weights (U_t^k). Below all norms are ℓ_2 norms.

Algorithm 4 has a synchronization step. This assumption implies that if the algorithmic mapping in Lemma 2.2 is used then the control parameters U are updated as follows,

$$U^{k+1} = U^k - \eta_k G(U^k) \tag{8}$$

where,

$$G(U^k) = -\langle \nabla_x f(X^k, U^k), \widehat{P}^k \rangle + \nabla_u R(U^k).$$

We note that if gradient descent is used to solve (4) then Algorithm 3 reduces to,

$$U^{k+1} = U^k - \nabla_u \mathcal{J}(U^k)$$

where $\mathcal{J}(U)$ is the objective function of (4), and the gradient $\nabla_u \mathcal{J}$ is calculated using backpropagation. Below we also use the following short-hand notation,

$$\partial_j f_k^\delta = \nabla_{u_j} f^\delta(X_j^k, U_j^k)$$
$$\partial_j R_k^\delta = \nabla_{u_j} R^\delta(U_j^k)$$
$$\partial_j \mathcal{J}_k = \nabla_{u_j} \mathcal{J}(U^k)$$

We note that with the simplified notation above we have $G_j(U^k) = -\langle \partial_j f_k^\delta, \widehat{P}_{j+1}^k \rangle + \partial_j R_k^\delta$.

When the algorithm is run with mini-batches, we use $\mathbb{E}[A]$ to denote the total expectation of the random variable A, and $\mathbb{E}[A \mid \mathcal{I}_k]$ is conditional on the information available up to and including iteration k.

7.2. Assumptions

In this section we outline our main assumptions.

1. The objective function ($\mathcal{J}$) in (4) has Lipschitz continuous gradient. We denote the Lipschitz constant with L.
2. The problems in (1) and (4) have a finite solution.
3. The discretization scheme used to obtain the (4) from (1) is stable (in the sense of (Butcher, 2016)) and consistent in the sense of (Polak, 1997).
4. The error in the adjoint calculation satisfies the inequality,

$$\|P_j^k - \widehat{P}_j^k\| \leq \epsilon_p \eta_k \|\widehat{P}_j^k\| \tag{9}$$

7.3. Proofs for Section 2

Proof. (Of Lemma 2.1).
This result can easily be established by induction. At the terminal time $s = T$, we must have,

$$V_s([X]) = \sum_{i=1}^m \Phi(x^i).$$

Therefore, by taking $V_T(x) \triangleq \frac{1}{m}\Phi(x)$ we establish that the Lemma is true for $s = T$. Suppose that the Lemma is true for some $0 < t + \delta t < T$, we show that it is true for t also. Let $u^\star(t)$ denote an optimal solution, then by assumption we have,

$$\begin{aligned} V_t([X]) &= \int_t^{t+\delta t} R(u_s^\star)ds + V_{t+\delta t}([X]) \\ &= \int_t^{t+\delta t} R(u_s^\star)ds + \sum_{i=1}^m V_{t+\delta}(x^i). \end{aligned}$$

Let $V_t(x) \triangleq \frac{1}{m}(\int_t^{t+\delta t} R(u_s^\star)ds + \sum_{i=1}^m V_{t+\delta}(x))$, and the result follows. □

Proof. (Of Lemma 2.2).
At time $t = T = T_\delta$, it follows from the boundary condition enforced by Algorithm 2 that $P_T^k = -\nabla_{X_T}\Phi(X_T^k)$. If we take one step of the backward solve then for $t = T - \delta$ we obtain,

$$\begin{aligned} P_t^k &= -\langle \nabla_{x_t} f^\delta(X_t^k, U_t^k), P_T^k\rangle \\ &= \langle \nabla_{x_t} f^\delta(X_t^k, U_t^k), \nabla_{X_T}\Phi(X_T^k)\rangle \\ &= -\nabla_{x_t}\Phi(f^\delta(X_t^k, U_t^k)) \\ &= -\nabla_{x_t}\Phi(X_T^k). \end{aligned}$$

Using the same argument recursively we conclude that,

$$P_t^k = -\nabla_{x_t}\Phi(X_T^k),\ 0 \le t \le T. \tag{10}$$

Using the preceding equation we obtain,

$$\begin{aligned} \partial_j J_k &= \nabla_{u_j}\Phi(X_T^k) + \nabla_{u_j} R^\delta(u_j^k) \\ &= \langle \nabla_{u_j}(X_{j+1}^k), \nabla_{x_{j+1}}\Phi(X_T^k)\rangle + \nabla_{u_j} R^\delta(u_j^k) \\ &= -\langle \nabla_{u_j} f^\delta(X_j^k, U_j^k), P_{j+1}^k\rangle + \nabla_{u_j} R^\delta(u_j^k). \end{aligned}$$

It follows that,

$$U_j^{k+1} = U_j^k - \eta\partial_j J_k = \mathcal{A}(U_j^k, X_j^k, P_{j+1}^k),$$

and we conclude that SGD with a learning rate of η produces the same iterations as Algorithm 3. □

7.4. Proofs for Section 4

Proof. (Of Theorem 4.1) Because $\mathcal{J}$ has a Lipschitz continuous gradient it follows from (8),

$$\begin{aligned} \mathcal{J}(U_{k+1}) \le& J(U_k) + \sum_{j=0}^{T_\delta - 1}\left(-\eta_k\langle \partial_j\mathcal{J}_k, G_j(U^k)\rangle + \frac{\eta_k^2 L}{2}\|G_j(U^k)\|^2\right) \\ =& \mathcal{J}(U_k) + \sum_{j=0}^{T_\delta - 1}\left(-\eta_k\langle \partial_j\mathcal{J}_k - G_j(U^k), G_j(U^k)\rangle + \frac{\eta_k^2 L - 2\eta_k}{2}\|G_j(U^k)\|^2\right) \\ =& \mathcal{J}(U_k) + \sum_{j=0}^{T_\delta - 1}\left(\eta_k\langle\langle \partial_j f_k^\delta, P_{j+1}^k - \widehat{P}_{j+1}^k\rangle, G_j(U^k)\rangle + \frac{\eta_k^2 L - 2\eta_k}{2}\|G_j(U^k)\|^2\right) \\ \le& \mathcal{J}(U_k) + \sum_{j=0}^{T_\delta - 1}\left(\left(\epsilon_p\eta_k^2 + \frac{\eta_k^2 L - 2\eta_k}{2}\right)\|\partial_j f_k^\delta\|^2\|\widehat{P}_{j+1}^k\|^2 + \left(\epsilon_p\eta_k^2 + \eta_k^2 L - 2\eta_k\right)\|\partial_j f_k^\delta\|\|\widehat{P}_{j+1}^k\|\|\partial_j R_k^\delta\|\right. \\ &\left. + \frac{\eta_k^2 L - 2\eta_k}{2}\|\partial_j R_k^\delta\|^2\right) \end{aligned}$$

Since $\eta_k \le \frac{2}{2\epsilon_p + L}$ the result follows.

□

Next we discuss the case when Algorithm 4 is run using mini-batches. In this case, the iteration in (8) is replaced with the following,

$$U^{k+1} = U^k - \alpha_k G(U^k, \xi)$$

where $G(U^k, \xi)$ denotes a randomized version of $G(U^k)$. The following assumption is standard regrading the sampling process (see e.g. (Bottou et al., 2018)),

$$\langle \nabla \mathcal{J}_k(U^k), \mathbb{E}[\nabla \mathcal{J}_k(U^k, \xi) \mid \mathcal{I}_k] \rangle \geq \mu \|\nabla \mathcal{J}_k(U^k)\|^2. \tag{11}$$

We now establish some technical lemmas that are needed for the proof of Theorem 4.2.

Lemma 7.1. *Suppose that*

$$\mathbb{E}[\|\nabla_u f^\delta(X^k, U^k, \xi)\| \|\hat{P}(\xi)\| \mid \mathcal{I}_k] \leq M_1 \|\nabla \mathcal{J}_k(U^k)\|$$

Then,

$$\langle \nabla \mathcal{J}_k(U^k), \mathbb{E}[G(U^k, \xi) \mid \mathcal{I}_k] \rangle \geq (\mu - \epsilon_p M_1) \|\nabla \mathcal{J}_k(U^k)\|^2$$

Proof. It follows from (11)

$$\begin{aligned}
\mu \|\nabla \mathcal{J}_k(U^k)\|^2 &\leq \langle \nabla \mathcal{J}_k(U^k), \mathbb{E}[-\langle \nabla_u f^\delta(X^k, U^k, \xi), P^k(\xi) \rangle + \nabla_u R^\delta(U^k) \mid \mathcal{I}_k] \rangle \\
&= \langle \nabla \mathcal{J}_k(U^k), \mathbb{E}[-\langle \nabla_u f^\delta(X^k, U^k, \xi), P^k(\xi) - \widehat{P}^k(\xi) \rangle - \langle \nabla_u f^\delta(X^k, U^k, \xi), \widehat{P}^k(\xi) \rangle + \nabla_u R^\delta(U^k) \mid \mathcal{I}_k] \rangle \\
&\leq \langle \nabla \mathcal{J}_k(U^k), \mathbb{E}[G(U^k, \xi) \mid \mathcal{I}_k] \rangle + \epsilon_p \|\nabla \mathcal{J}_k(U^k)\| \mathbb{E}[\|\nabla_u f^\delta(X^k, U^k, \xi)\| \|\hat{P}(\xi)\| \mid \mathcal{I}_k] \\
&\leq \langle \nabla \mathcal{J}_k(U^k), \mathbb{E}[G(U^k, \xi) \mid \mathcal{I}_k] \rangle + \epsilon_p M_1 \|\nabla \mathcal{J}_k(U^k)\|^2,
\end{aligned}$$

and by re-arranging the inequality above, the result follows. □

Lemma 7.2. *Suppose that*

$$\mathbb{E}[\|\langle \nabla_u f^\delta(X^k, U^k, \xi), \widehat{P}^k(\xi) \rangle\|^2 + \|\nabla_u R^\delta(U, \xi)\|^2 \mid \mathcal{I}_k] \leq M_2 + M_3 \|\nabla_u \mathcal{J}(U^k)\|^2 \tag{12}$$

Then,

$$\mathbb{E}[\mathcal{J}(U^{k+1}) \mid \mathcal{I}_k] - \mathcal{J}(U^k) \leq -\eta_k(\mu - \epsilon_p M_1 - \eta_k M_3 L) \|\nabla \mathcal{J}_k(U^k)\|^2 + L\eta_k^2 M_2.$$

Proof.

$$\mathbb{E}[\mathcal{J}(U^{k+1}) \mid \mathcal{I}_k] - \mathcal{J}(U^k) \leq -\eta_k \langle \nabla \mathcal{J}(U^k), \mathbb{E}[G(U^k, \xi) \mid \mathcal{I}_k] \rangle + \frac{L\eta_k^2}{2} \mathbb{E}[\|G(U^k, \xi)\|^2 \mid \mathcal{I}_k] \tag{13}$$

We can bound the first term using Lemma 7.1 We bound the second term as follows,

$$\begin{aligned}
\|G(U^k, \xi)\| &= \| - \langle \nabla_u f^\delta(X^k, U^k, \xi), \widehat{P}^k(\xi) \rangle + \nabla_u R^\delta(U^k, \xi)\| \\
&\leq \|\langle \nabla_u f^\delta(X^k, U^k, \xi), \widehat{P}^k(\xi) \rangle\| + \|\nabla_u R^\delta(U^k, \xi)\|
\end{aligned}$$

Taking squares on both sides, using the inequality $(a+b)^2 \leq 2(a^2 + b^2)$ and taking conditional expectations on both sides we find,

$$\begin{aligned}
\mathbb{E}[\|G(U^k, \xi)\|^2 \mid \mathcal{I}_k] &\leq 2 \left(\mathbb{E}[\|\langle \nabla_u f^\delta(X^k, U^k, \xi), \widehat{P}^k(\xi) \rangle\|^2 \mid \mathcal{I}_k] + \mathbb{E}[\|\nabla_u R^\delta(U^k, \xi)\|^2 \mid \mathcal{I}_k] \right) \\
&\leq 2(M_2 + M_3 \|\nabla_u \mathcal{J}(U^k)\|^2).
\end{aligned}$$

Using the bound above in (13), and Lemma 7.1 we obtain,

$$\mathbb{E}[\mathcal{J}(U^{k+1}) | \mathcal{I}_k] - \mathcal{J}(U^k) \leq -\eta_k(\mu - \epsilon_p M_1 - \eta_k M_3 L) \|\nabla \mathcal{J}_k(U^k)\|^2 + L\eta_k^2 M_2,$$

as claimed. □

Lemma 7.3. *Suppose that Algorithm 4 is run with a fixed step-size $\bar{\eta}$ such that,*

$$0 < \bar{\eta} \leq \frac{\mu - \epsilon_p M_1}{2M_3 L}, \tag{14}$$

then after N iterations the following holds,

$$\frac{1}{N}\sum_{i=1}^{N} \|\nabla \mathcal{J}(U^k)\|^2 \leq \frac{2L\bar{\eta}M_2}{\mu} + \frac{2(\mathcal{J}(U^N) - \mathcal{J}(U^1))}{\bar{\eta}N\mu}$$

Proof. Taking expectations on the bound obtained in Lemma 7.2 and using the law of total expectation we obtain,

$$\begin{aligned}\mathbb{E}[\mathcal{J}(U^{k+1})] - \mathbb{E}[\mathcal{J}(U^k)] &\leq -\bar{\eta}(\mu - \epsilon_p M_1 - \bar{\eta}_k M_3 L)\mathbb{E}[\|\nabla \mathcal{J}_k(U^k)\|^2] + L\bar{\eta}_k^2 M_2 \\ &\leq -\frac{\bar{\eta}\mu}{2}\mathbb{E}[\|\nabla \mathcal{J}_k(U^k)\|^2] + L\bar{\eta}^2 M_2.\end{aligned}$$

Summing from $k = 1$ to iteration N we obtain,

$$\mathcal{J}^\star - \mathcal{J}(U^1) \leq J(U^M) - J(U^1) \leq -\frac{\bar{\eta}\mu}{2}\sum_{k=1}^{N}\mathbb{E}[\|\nabla \mathcal{J}_k(U^k)\|^2] + NL\bar{\eta}^2 M_2.$$

Rearranging the inequality above we obtain the required result. □

Proof. (Of Theorem 4.2) This result can be established by observing that the conditions imposed on the step-size imply that $\eta_k \to 0$. Therefore, for k sufficiently large the assumption on the step-size in Lemma 7.3 holds. The rest of the proof is the same as the proof of Lemma 7.3. □

7.5. Additional Figures for Section 5

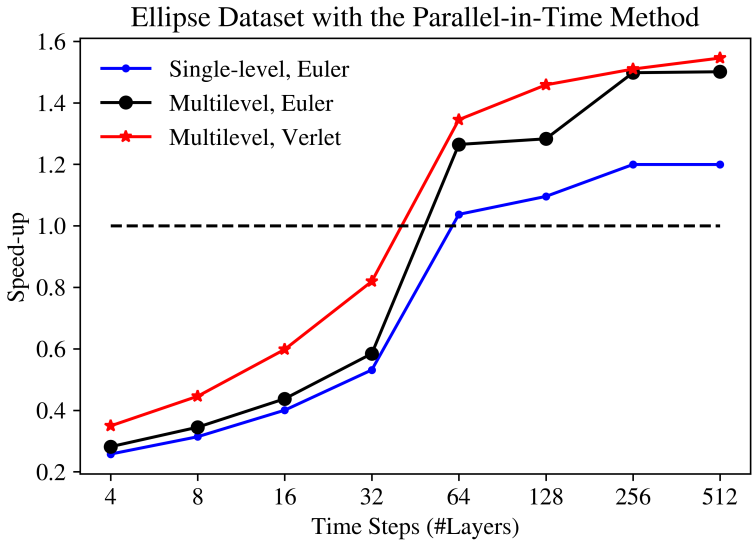

Figure 4. Speed-up for the Ellipse dataset over a data-parallel implementation of SGD. Like the Swiss-Roll dataset the results suggest an efficiency of 75% (for large networks) which is much greater than existing layer-parallel methods for training NNs.

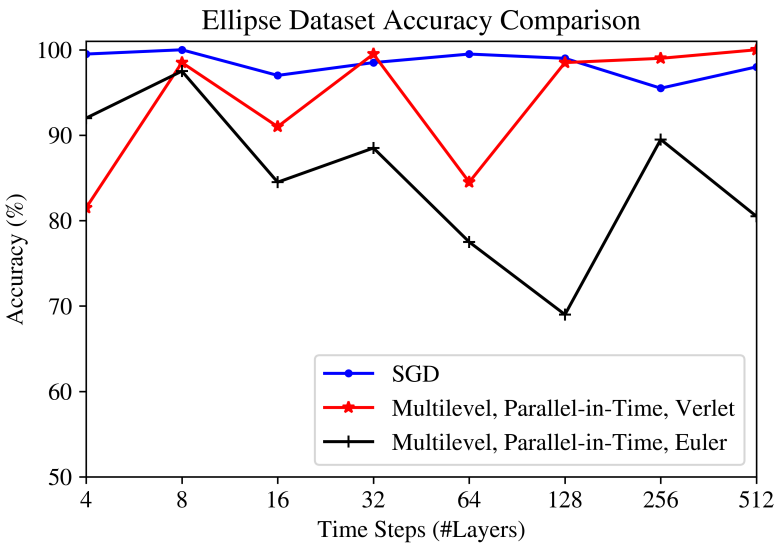

Figure 5. Accuracy for the Ellipse dataset when compared with a data-parallel implementation of Stochastic Gradient Descent (SGD).

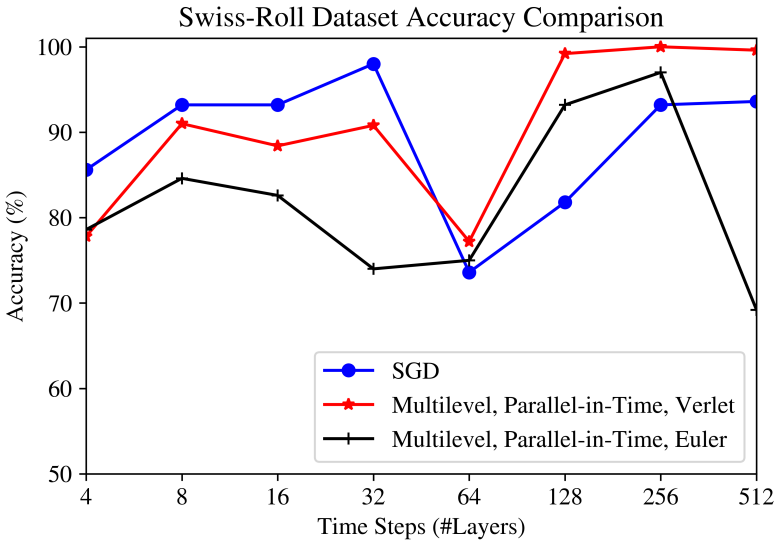

Figure 6. Accuracy for the Swiss-Roll dataset when compared with a data-parallel implementation of Stochastic Gradient Descent (SGD).

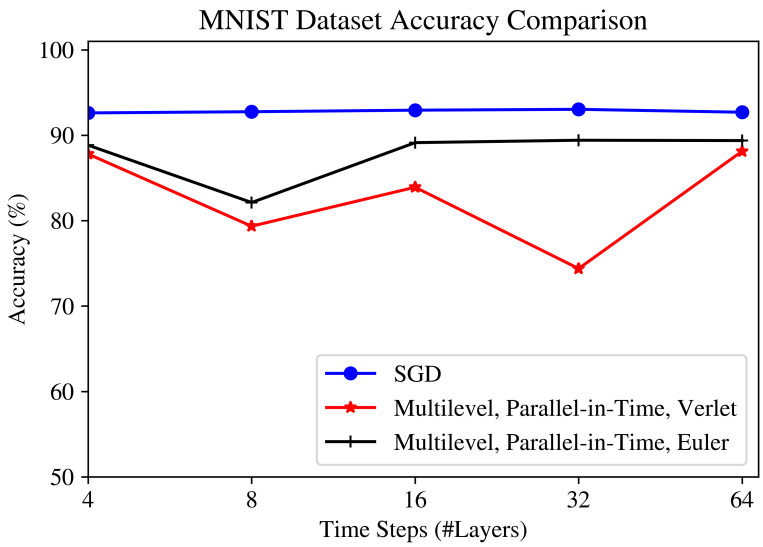

Figure 7. Accuracy for the MNIST dataset when compared with a data-parallel implementation of Stochastic Gradient Descent (SGD).