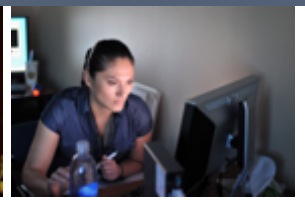
SAND2021-6984PE

A Layer-Parallel Approach for Training Deep Neural Networks



Eric C. Cyr, Sandia National Laboratories

Authors

Stefanie Guenther (LLNL), Lars Ruthotto (Emory), Jacob B. Schroder (UNM), Nico R. Gauger (TU Kaiserslautern), Gordon Moon (KAU), Ravi Patel (SNL)

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Neural Networks

A neural network is a parameterized model:

Neural Network → $\mathcal{NN}(x;\Theta) \rightarrow y$ ← Output

Input: x — Parameters: Θ

It is composed of multiple layers*

Feature Vectors

$$u_1 = A_0 x + b_0,$$

$$u_{i+1} = f(u_i; \{A_i, b_i\}) \quad i = 1 \ldots L-1,$$

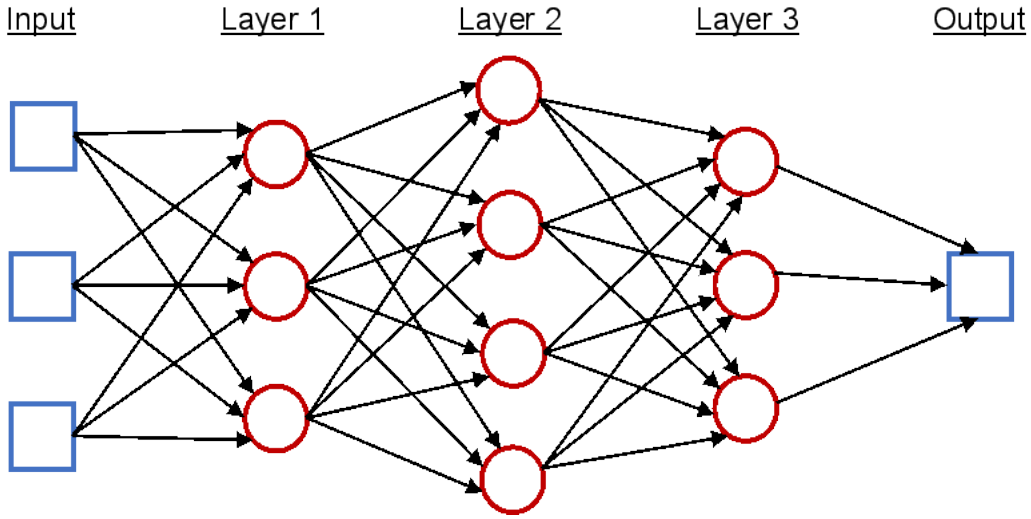
$$y = A_L u_L;$$

$$\Theta = \{A_i, b_i\}_{i=0}^{L-1} \cup \{A_L\}$$

*Your mileage may vary, there are so many possible architectures, this is our starting point

Neural Network Architectures*

	Update Rule: $f(u_i; \{A_i, b_i\})$
Feed Forward	$u_{i+1} = g(A_i u_i + b_i)$
ResNet	$u_{i+1} = u_i + g(A_i u_i + b_i)$
ODENet	$u_{i+1} = u_i + \Delta t g(A_i u_i + b_i)$ $\partial_t u = g(Au + b)$



Weighting Matrix

Bias Vector

Activation Function: nonlinear componentwise

*Your mileage may vary, there are so many possible architectures, this is our starting point

Determining the Parameters

Neural network should map data according to the sampled **training set** :

$$\{(x_i, y_i)\}$$

Input — Output

Find Θ minimizing the **loss** in the model over the **training set:**

Parameters

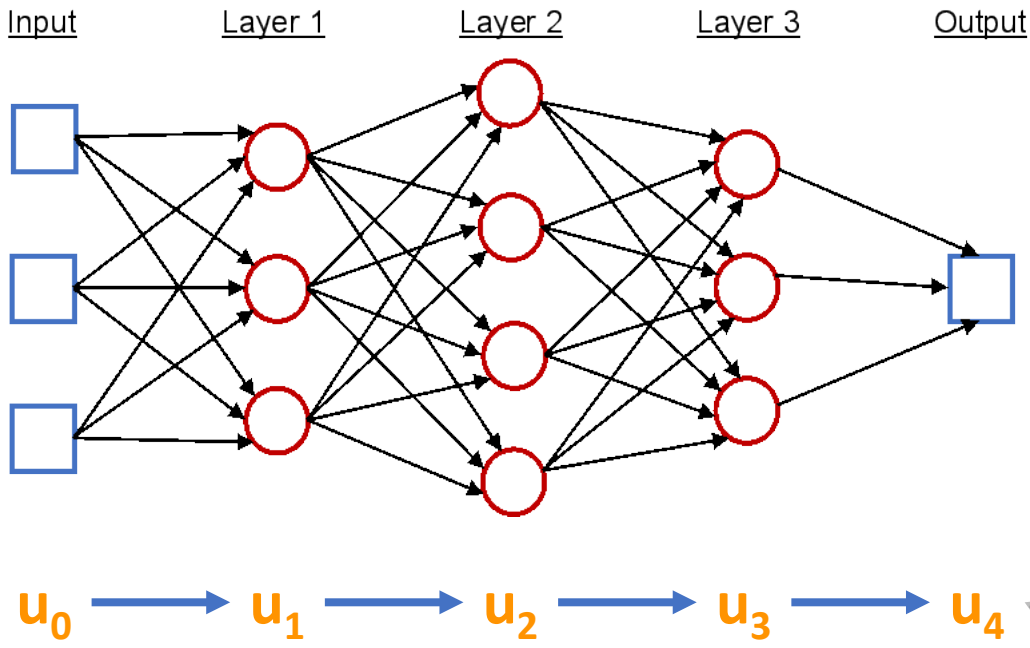
$$\min_{\Theta} \sum_{n=1}^{N} \text{Loss}\left(\mathcal{NN}(x_n; \Theta), y_n\right)$$

Loss function is model/data difference:

- $\text{Loss}(y^{model}, y^{data}) = \|y^{model} - y^{data}\|^2$
- $\text{Loss}(\vec{y}^{\,model}, \vec{y}^{\,data}) = \sum_{c=1}^{N_c} y_c^{data} \log\left(y_c^{model}\right)$

Neural Network Training as Constrained Optimization

Forward Inference:



Neural networks are a model that transform input u_0 to output u_4 by ”evolving” through layers

Training:

Solve optimization problem constrained by evolutionary models

- Supervised Training: Determine parameters that give best match to data

$$\begin{aligned} \underset{u_l, z_l}{\text{minimize}} \quad & \text{Loss}(u_L, z_1 \ldots z_L) \\ \text{subj. to} \quad & u_l = F(u_{l-1}, z_l) \end{aligned}$$

Feature Vectors

Parameters

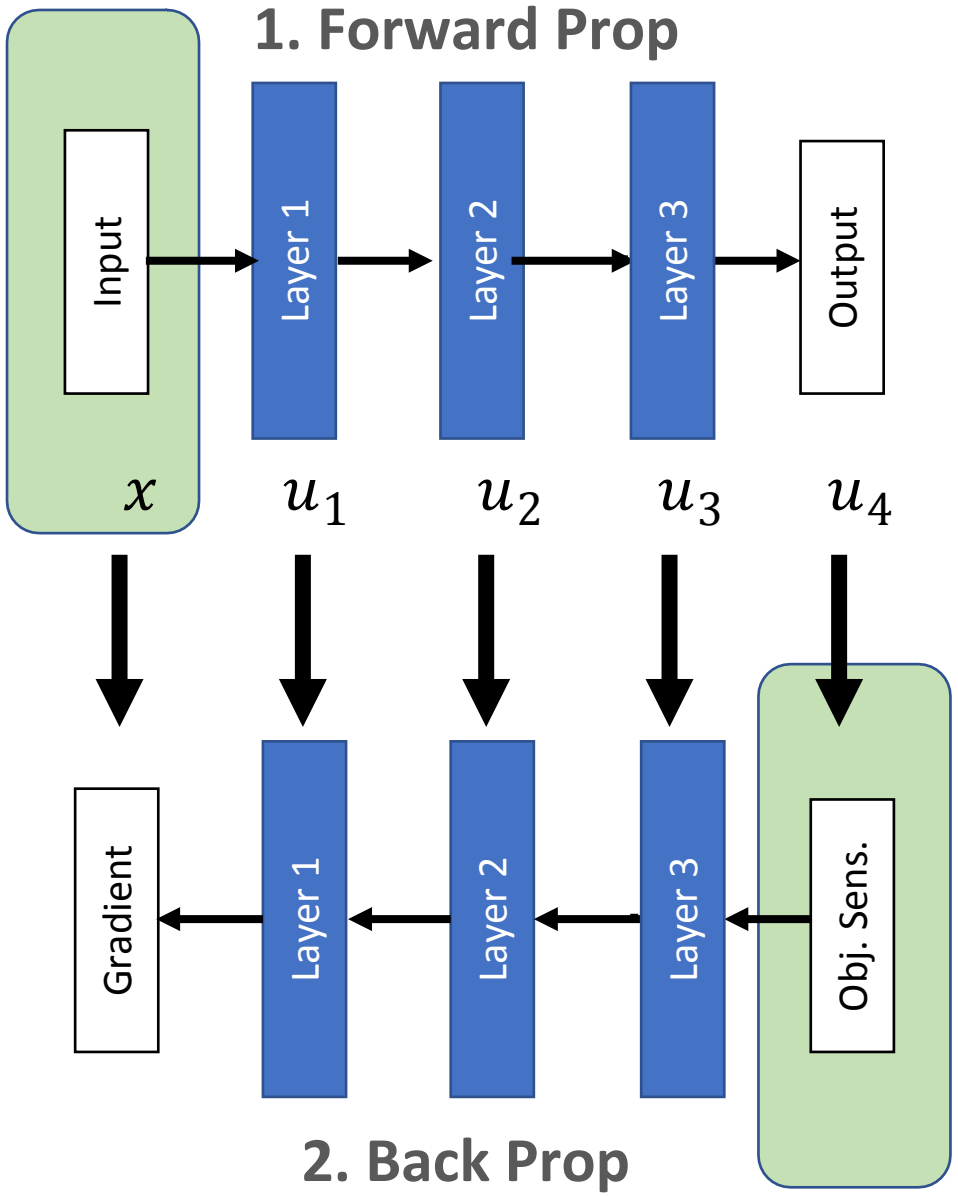
Training means optimize: Gradient Descent

Training "solves" an optimization problem for weights and biases:

$$\underset{\Theta}{\operatorname{argmin}} \sum_{n=1}^{N} \operatorname{Loss}(\mathcal{NN}(x_n, \Theta), y_n)$$

- Gradient descent:
  1. Forward prop: Features
  2. Back prop: Loss grad
  3. Step in negative grad

$$\nabla_{\Theta}(\operatorname{Loss})$$



Review of training methods: Bottou, Curtis, Nocedal. "Optimization methods for large -scale machine learning." *SIAM Review* 60, 2018.

Constrained Optimization

We take a constrained optimization viewpoint of training

$$\underset{\Theta}{\text{argmin}} \sum_{n=1}^{N} \text{Loss}(\mathcal{NN}(x_n, \Theta), y_n)$$

$$\underset{\Theta}{\text{argmin}} \sum_{n=1}^{N} \text{Loss}(y_n^{\mathcal{NN}}, y_n)$$

$$\text{subject to} \quad u_{1,n} = A_0 x_n + b_0,$$
$$u_{i+1,n} = f(u_{i,n}; \{A_i, b_i\}) \quad i = 1 \ldots L-1,$$
$$y_n^{\mathcal{NN}} = A_L u_{L,n}$$

Expanded $y_n^{\mathcal{NN}} = \mathcal{NN}(x_n; \Theta)$

Useful transformation! We will relax constraint enforcement and

Trade exactness for parallelism!

How to Accelerate Training With Parallelism?

Training neural networks can be costly (weeks)

- Loads of data to look at
- Lots of weights and features to optimize
- Nonlinear interactions to differentiate through
- Rely on on gradient descent for optimization

Can parallel computing in general, and HPC specifically help here?

- Already multi-GPU codes are helping
- New optimization algorithms less sensitive to inaccurate gradients being developed

Our Goal: Develop a new dimension of parallelism to exploit!

Ravi Patel
Sandia

Stefanie Guenther
LLNL Fernbach Fellow

Jacob Schroder
UNM

Gordon Moon
KAU

Lars Ruthotto
Emory

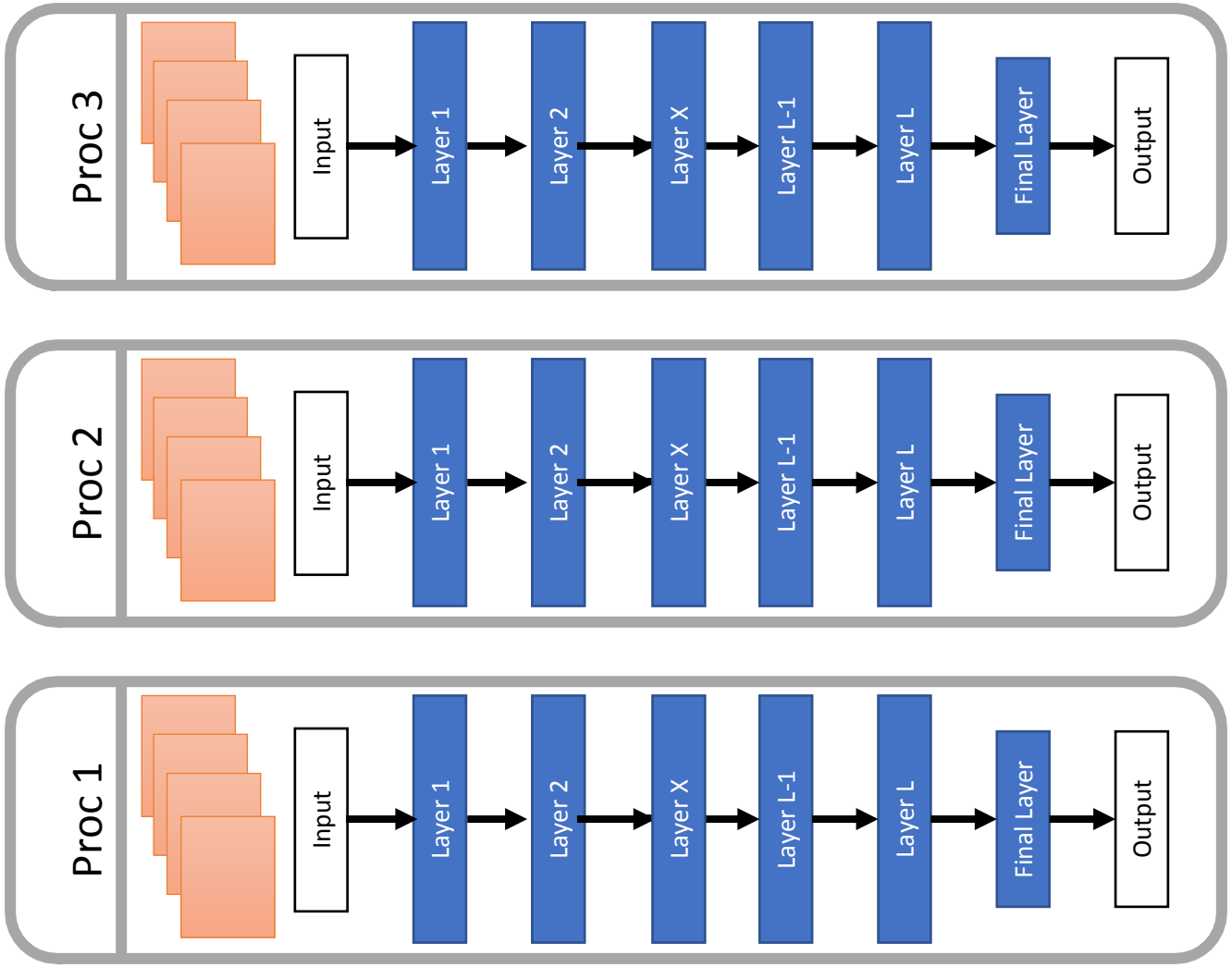
Nico Gauger
TU Kaiserslautern

Parallelization strategies: Data Parallel

Data Parallelism:

- Distribute a batch of samples over processors
- Replicate neural network across all processors

Problem: Stochastic gradient descent performance degrades with increased data size



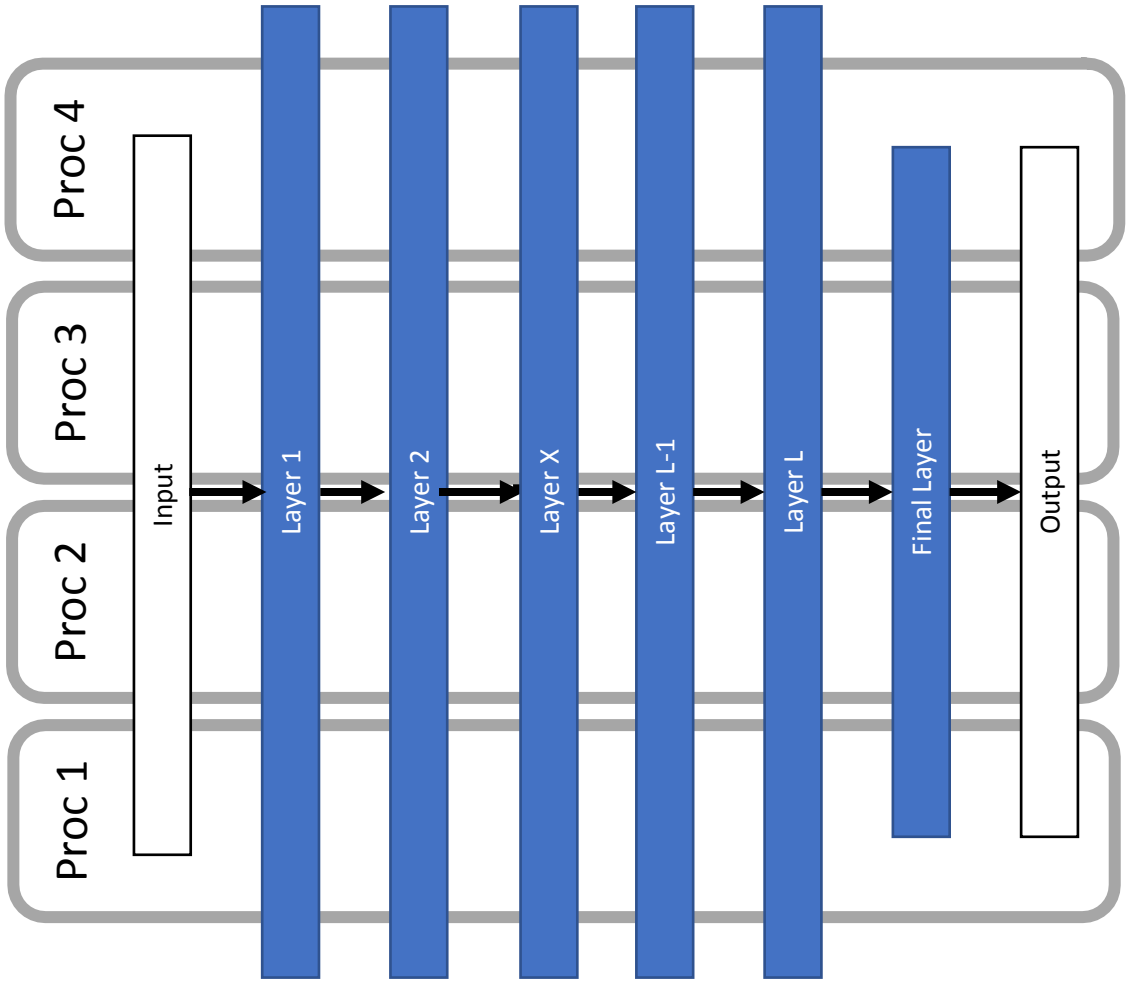
Parallelization strategies: Model Parallel

Model Parallelism:

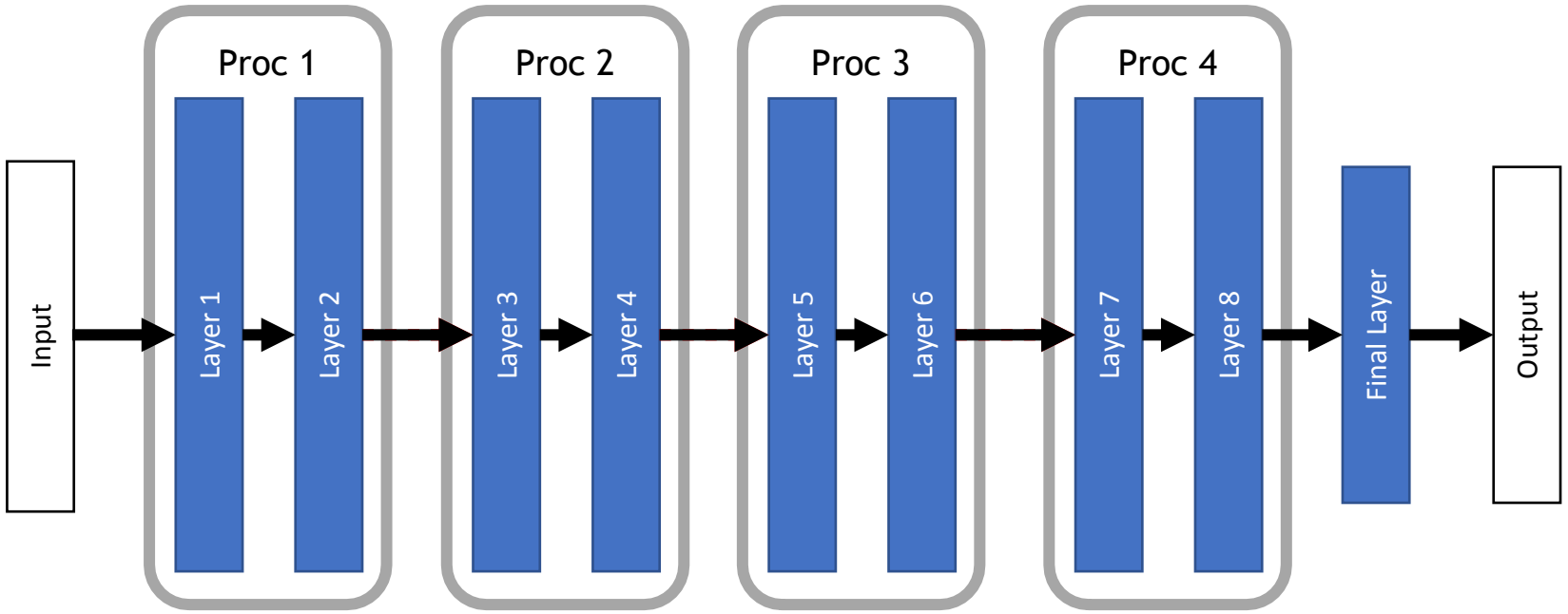
- Distribute network across processors
- Distribute data accordingly

Problem: Forward and backward propagation are serial bottlenecks. Increased depth leads unreasonable computation times

- Using a bigger computer will not solve this!



Our New Approach: Layer-Parallel Training



Inexact Evolution: Relax satisfaction of evolution constraints, trade accuracy for performance

Stitch Together: Constraint continuity required only at convergence of optimization

Wait, what? (Number one response)

Layer-Parallel makes no sense they say:

Gradient Descent Algorithm:

```
# initialize the solution
w_W = initialize_W()
w_b = initialize_b()

y0 = data
for iter in [1,max_iter]:
  # do forward propagation inference step
  x = forward_prop(y0,w_W,w_b)

  # do backward propogationj to compute the gradient
  g_W,g_b = backward_prop(x,y0,w_W,w_b)

  # update the solution with gradient descent
  w_W = w_W - learning_rate * g_W
  w_b = w_b - learning_rate * g_b
```

These serialize across the layers. A forward and then a backward sweep! How can you parallelize

- Forward and backward propagation are serial!
- Distributing the layers across processors still serializes!
- It doesn't make a whole lot of sense does it?

Critical Assumption: Exactness of propagation

We can relax the exactness of propagation, and trade for parallelism!

Gradient Descent Algorithm:

```
# initialize the solution
w_W = initialize_W()
w_b = initialize_b()

y0 = data
for iter in [1,max_iter]:
  # do forward propagation inference step
  x = forward_prop(y0,w_W,w_b)  + ϵ_f

  # do backward propogationj to compute the gradient
  g_W,g_b = backward_prop(x,y0,w_W,w_b) +ϵ_b

  # update the solution with gradient descent
  w_W = w_W - learning_rate * g_W
  w_b = w_b - learning_rate * g_b
```

Introduce a small error

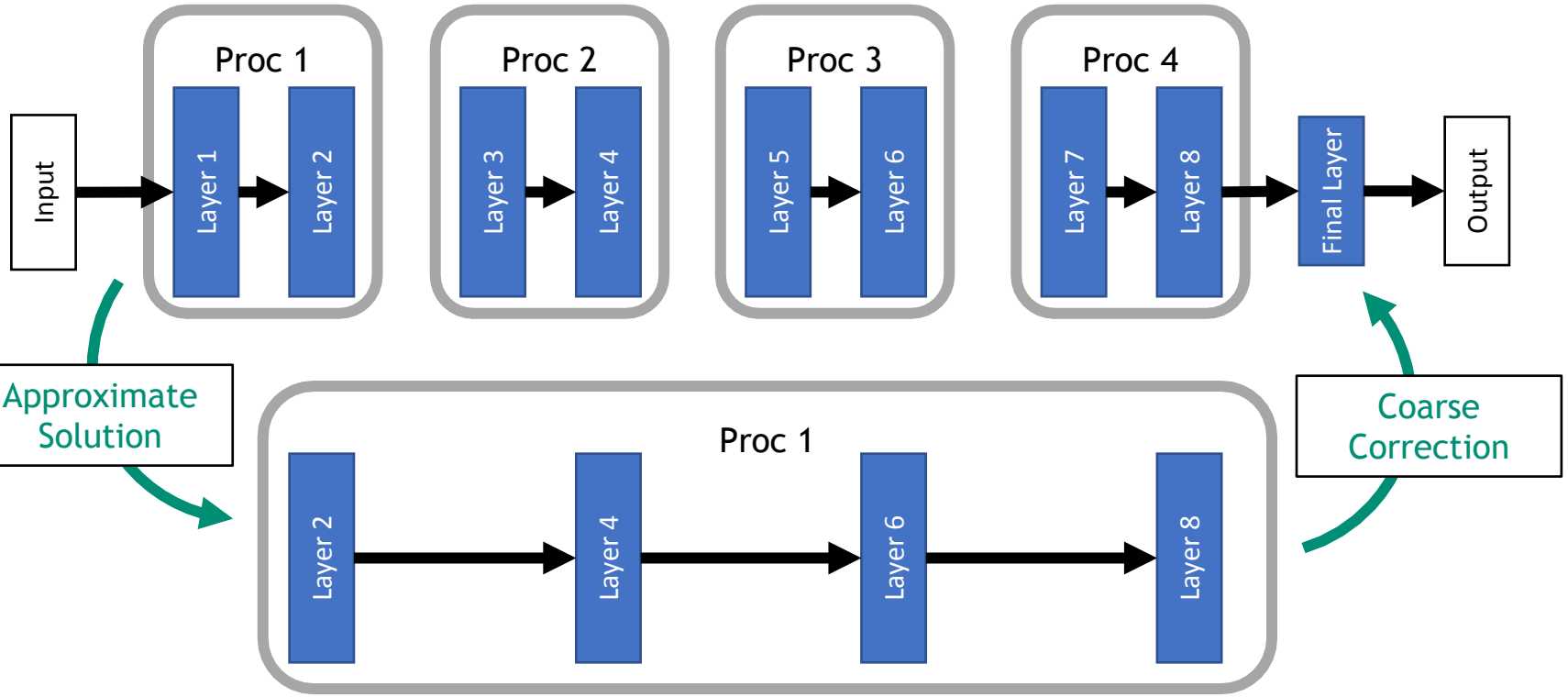
- If we can control the error we introduce, we can use it to get parallelism!
- We introduce this error through a multigrid algorithm, and get parallelism as a result

Layer Parallel Training – A Multigrid Approach*

Multi-grid algorithm uses “divide and conquer” approach to inference

- “Fine grid relaxation”: Fixes local errors between layers – embarrassingly parallel
- “Coarse grid correction”: Fixes global errors – serial inference on smaller network

Fine Grid Relaxation
(Approximate/Parallel)



Coarse grid correction
(Exact/Serial/Cheap)

Multigrid is applied for both forward and back propagation

*Based on Multigrid-In-Time: Collaboration with J. Schroder (UNM), S. Günther (LLNL), L. Ruthotto (Emory), N. Gauger (TU Kasierslatern)
Multi-grid in time reference: Falgout, Friedhoff, Kolev, MacLachlan, Schroder. "Parallel time integration with multigrid." SIAM SISC 2014.

Layer-Parallel Algorithm: Details

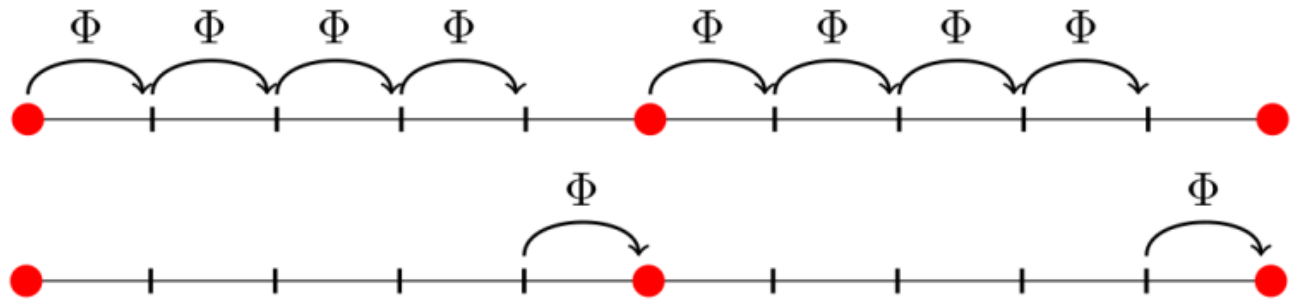
1. Uses ODE Networks (time=layers)
   - Think ResNet as an ODE
   - Theory from multigrid-in-time
   - Questions about regularity required

$$x_{k+1} = x_k + \Delta t \sigma(W_k x_k + b_k)$$

Discretize

$$\partial_t x(t) = \sigma(W(t)x(t) + b(t))$$

2. Fine-Coarse-Fine (FCF) relaxation with FAS multigrid:
   1. Relax fine points
   2. Relax on coarse points
   3. Relax on fine points again

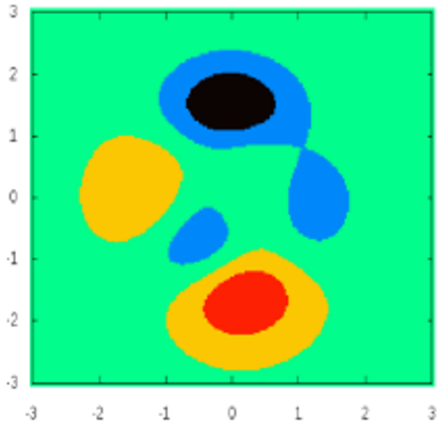


3. Using one-shot optimization
   - No batching like SGD
   - Probably suboptimal
   - Using L-BFGS Hessian

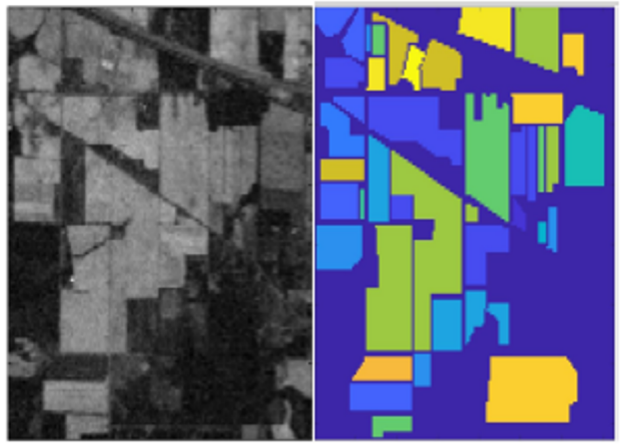
$$\square^{n} \quad \square^{n} - \square H_k^{-1} \left[\square \frac{d\text{Loss}_k}{d\square^{n}}\right]$$

with $H \square @^2_{\square}\left(L + \square ky - H(y, \square)k^2\right)$

Layer Parallel Scaling Results



(a) Peaks



(b) Indian Pines

(c) MNIST

Three different classification problems

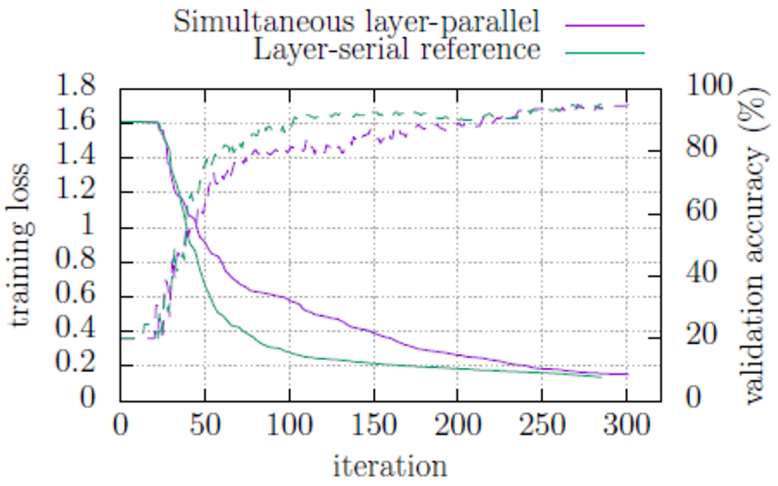
1. Peaks: Put particle position into one of 5 different classes
2. Indian Pines: Hyperspectral imaging, what crop? Soy, corn, etc…
3. MNIST: Handwritten digit classification

A comment on the code:

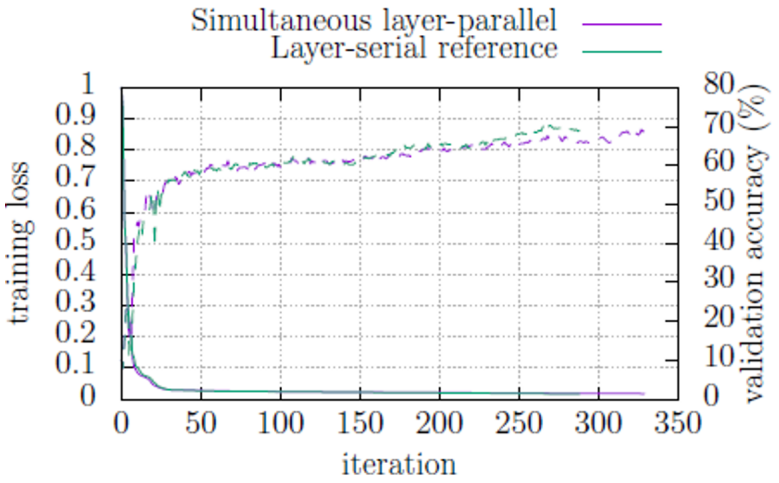
- Neural network code using Xbraid (LLNL) parallel-in-time library
- Code is not optimized: e.g. MNIST uses hand coded convolutions
- Neural networks architectures not optimized, simple ODENets

Layer Parallel Scaling Results

Peaks

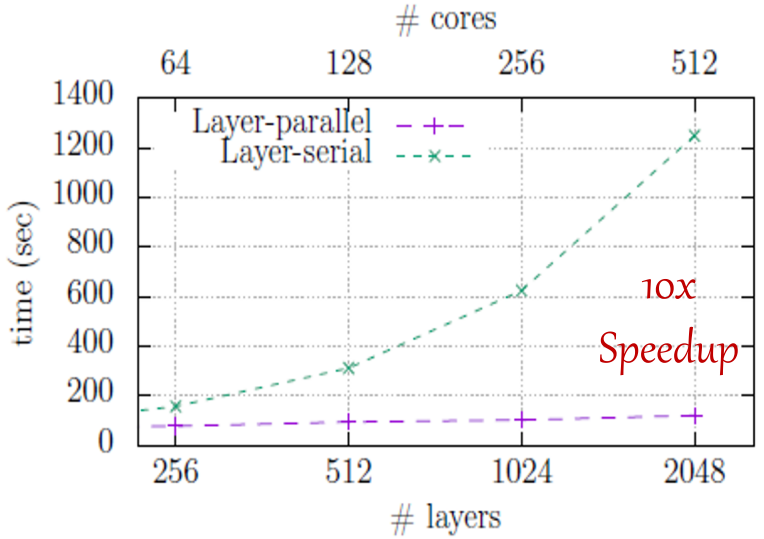


Indian Pines

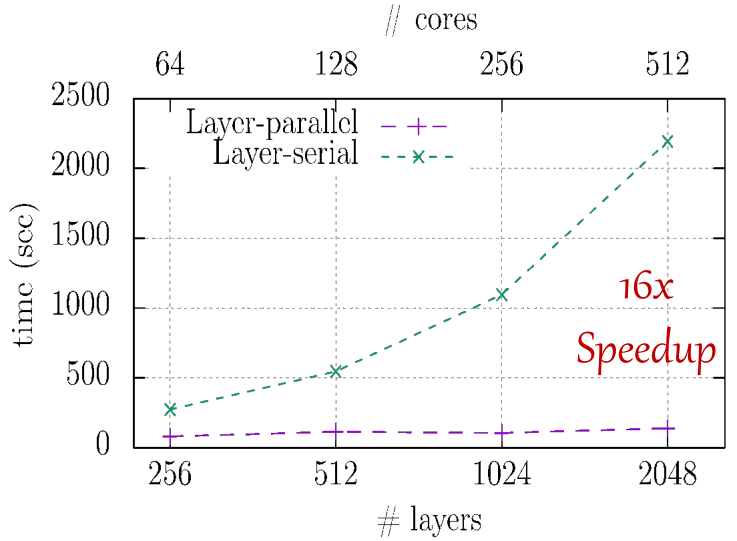


Weak Scaling

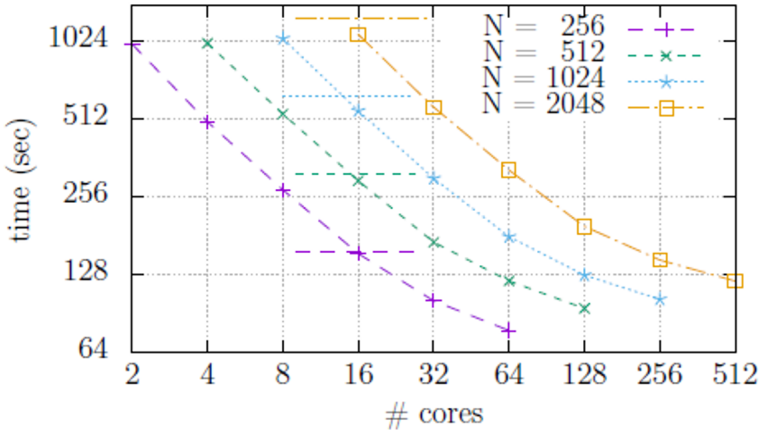
Indian Pines

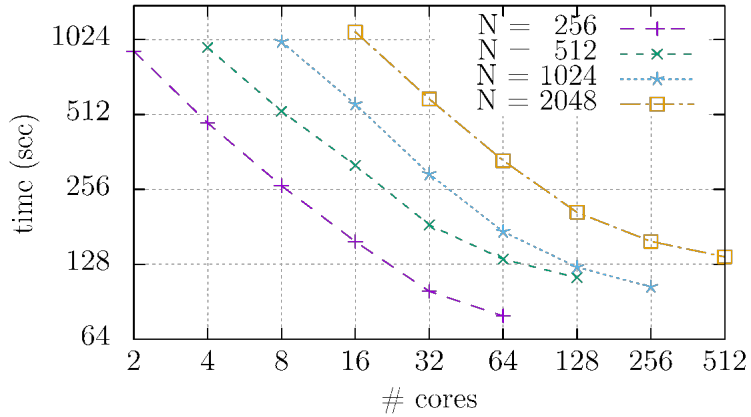


MNIST



Strong Scaling

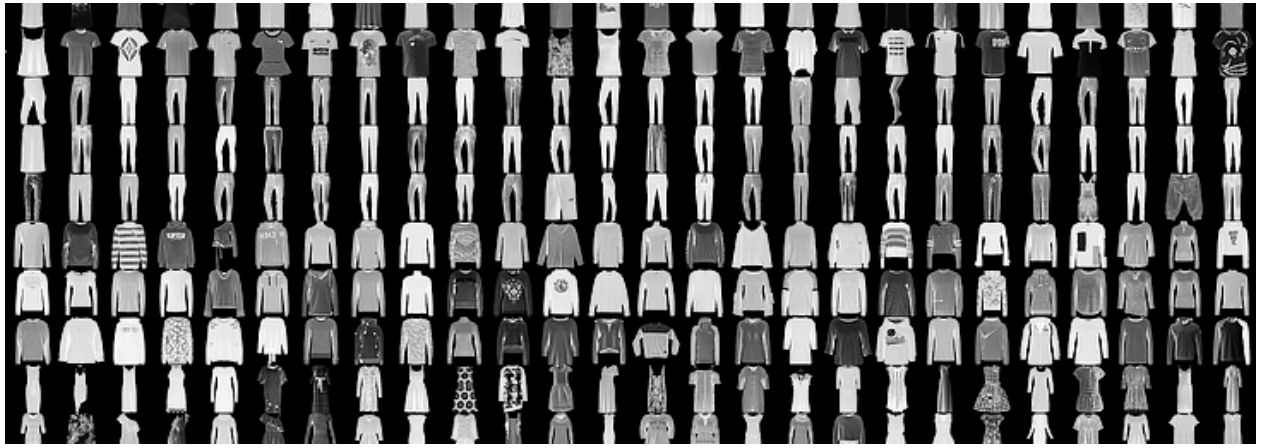


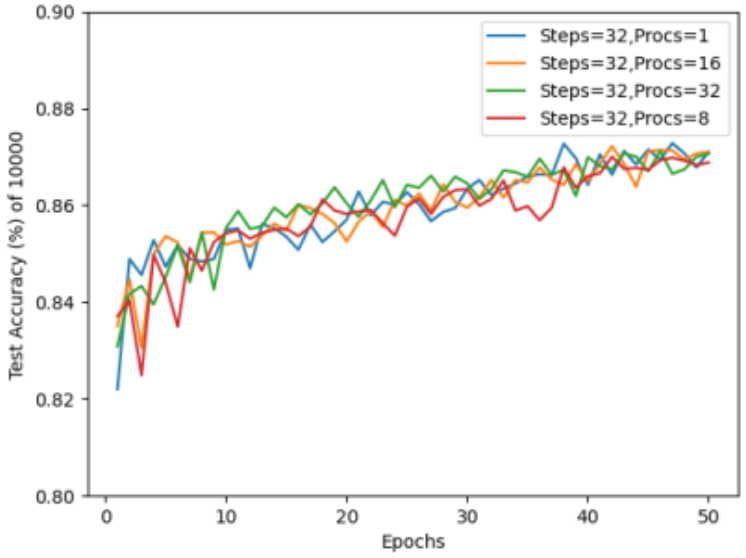


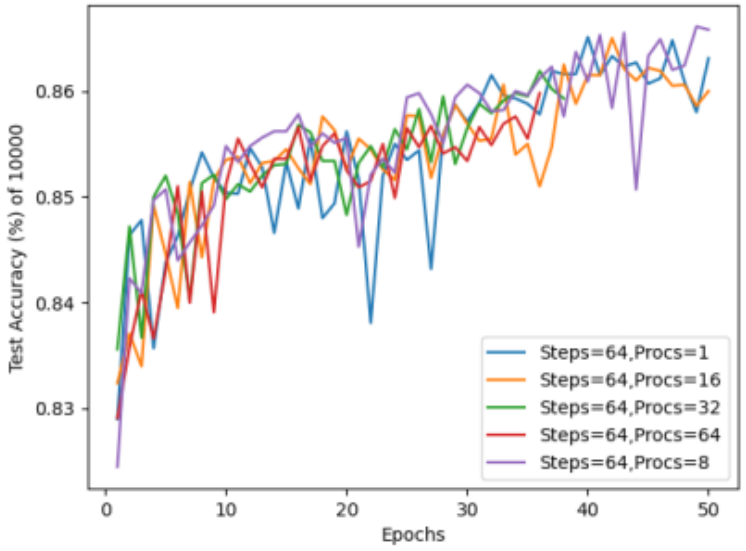
Using Stochastic Gradient Descent (SGD)

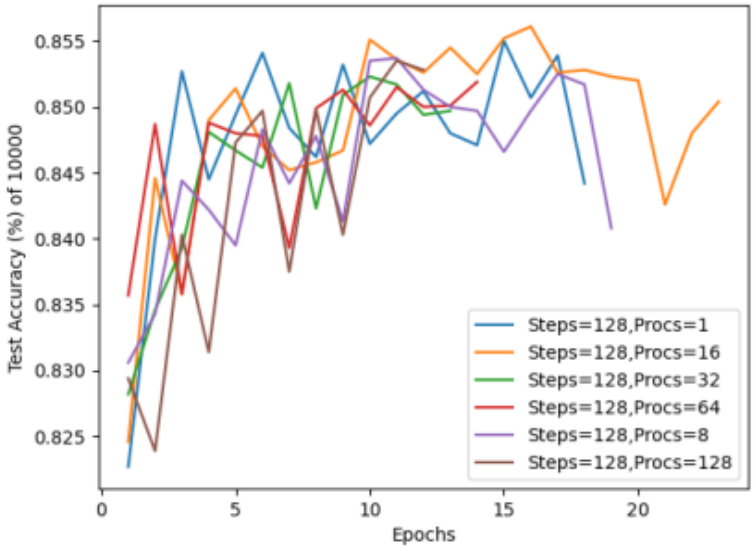
Workhorse of ML is SGD optimizer

- How does Layer-Parallel perform
- Compare networks trained with SGD
- Using “harder” fashion MNIST data set
- Similar speedups as seen previously









No loss of accuracy from layer-parallel compared to serial algorithm

Beyond ResNets: Recurrent Neural Nets (RNN)

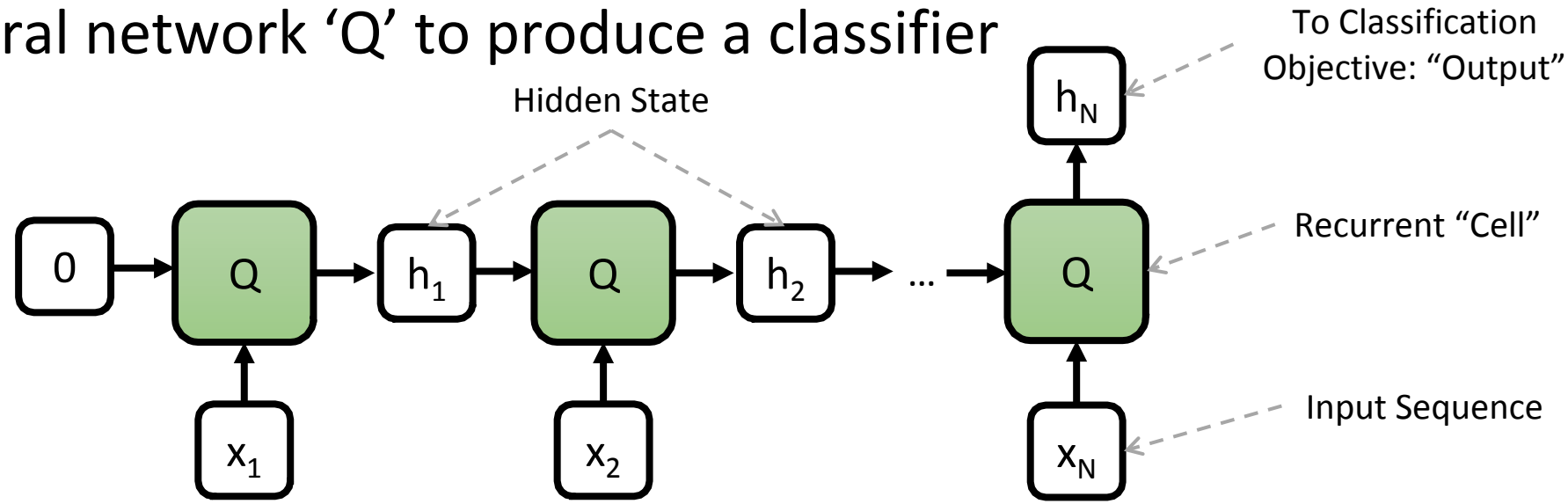
Problem: Classify a sequence, e.g. learn the mapping

$$\Phi(\underbrace{x_1, x_2 \ldots, x_N}) \rightarrow \underbrace{\{1, C\}}$$

Sequence of N items — One of C classes

Solution: Recurrent neural network

Learn a neural network 'Q' to produce a classifier



See "Colah's Blog" for a really great discussion (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

Generalized Recurrent Units (GRUs)

LSTMs and GRUs are two trainable types of RNNs

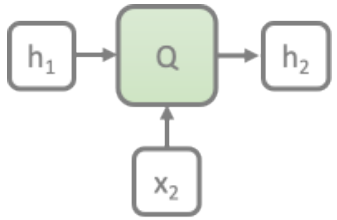
- Historically RNNs are hard to train (my read is they were unstable)
- “memory”: remembers important features in the sequence
- “forget” gates: eliminates some redundant/irrelevant from the sequence

Generalized Recurrent Units:

- $\mathbf{h}_*$: Hidden State,
- x_*: Input Sequence,
- W_* and b_*: Learnable Network Parameters

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}\boldsymbol{h_{t-1}} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}\boldsymbol{h_{t-1}} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}\boldsymbol{h_{t-1}} + b_{hn})) \\
\boldsymbol{h_t} &= z_t \odot \boldsymbol{h_{t-1}} + (1 - z_t) \odot n_t
\end{aligned}
\quad \Rightarrow \quad \boldsymbol{h_t} = Q(\boldsymbol{h_{t-1}}, x_t; \xi)
$$

Hadamard Product



Generalized Recurrent Units (GRUs)

We rewrite the update with a time step update (assume $\Delta t = 1$)

$$h_t = Q(h_{t-1}, x_t; \xi) = z_t \odot h_{t-1} + (1 - z_t) \odot n_t$$
$$= h_{t-1} + \Delta t \left((z_t - 1) \odot h_{t-1} + (1 - z_t) \odot n_t \right)$$

Taking $\Delta t \to 0$, we arrive at an ODE form

$$\partial_t h(t) = \underbrace{-(1 - z(t)) \odot h(t)}_{\text{Stiff mode: Collapsing onto multi-rate asymptotic (this is a stabilizing dissipation term!)}} + \underbrace{(1 - z(t)) \odot n(t)}_{\text{Introduction of new sequence information}}$$

Implicit GRUs

Stiff mode suggests a problem for traditional GRU's with large Δt:

➢ This will be a problem for coarse grids in layer-parallel!

We introduce a new "Implicit GRU" as a result, default to $\Delta t = 1$:

$$(1 + \Delta t(1 - z_t)) \odot h_t = h_{t-1} + \Delta t(1 - z_t) \odot n_t$$

- Because stiff mode is implicit, this new formulation will be stable for "large" Δt
- We will leverage this in a multi-level solver

Multi-Level GRU

There are several forms of GRU that we could consider for layer-parallel

1. Classic GRU (explicit with $\Delta t = 1$) – All multi-grid levels

   When you take bigger time steps, the result is instability on the coarse grid!

2. Classic GRU on fine levels, Implicit GRU on coarse

   With MGRIT this results in a mismatch that doesn't converge!

3. Variable Δt GRU (implicit or explicit)

   I have finite time ☹

4. Implicit GRU on coarse with $\Delta t = 1$ ← We try this one

Human Activity Recognition Using Smartphones Dataset (v1.0)[1,2]

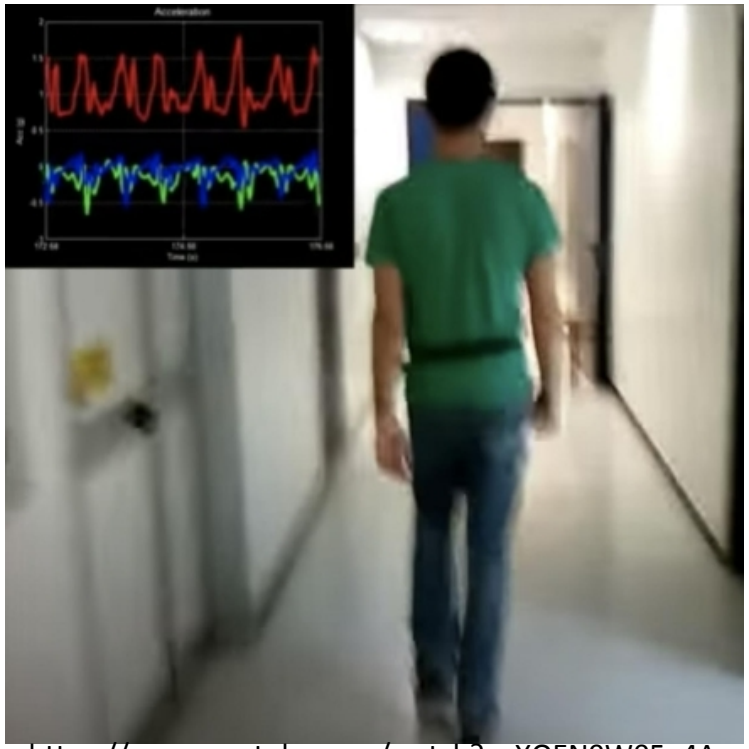
Dataset Details:

- 30 Volunteers performed six activities: WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING
- Smartphone accelerometers measured three different types of motion, yielding 9 features per sample
- Times windows of 2.56s composed of 128 time samples are labeled with activity
- 70% of volunteers selected for training data (7352 sequences), and 30% for test (2947 sequences)

Short Story: Supervised Classification Problem

- 6 labels
- Sequence of 128 steps, with 9 features
- Training set of 7352 sequences
- Testing set of 2947 sequences

PyTorch GRU and LSTM Implementations get to 90% test accuracy in 5-10 epochs with Adam (e.g. its not a really difficult problem)

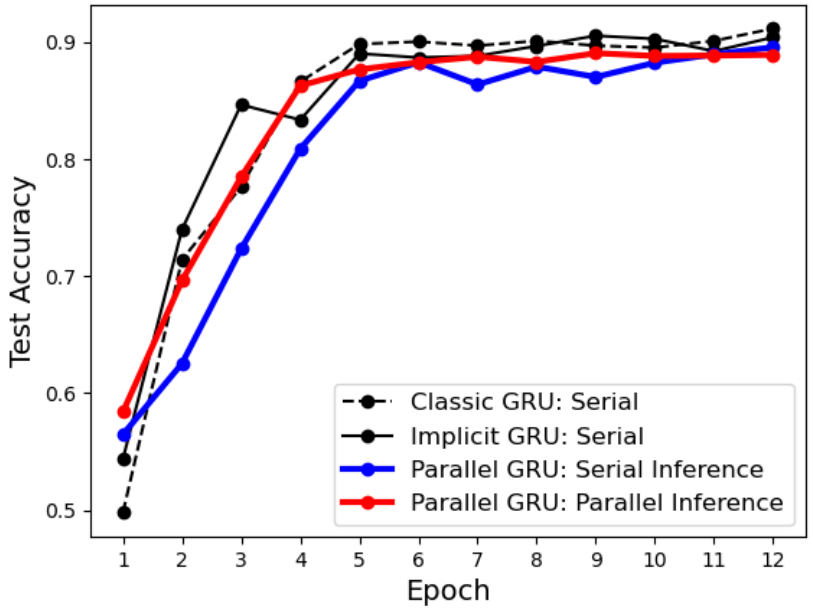


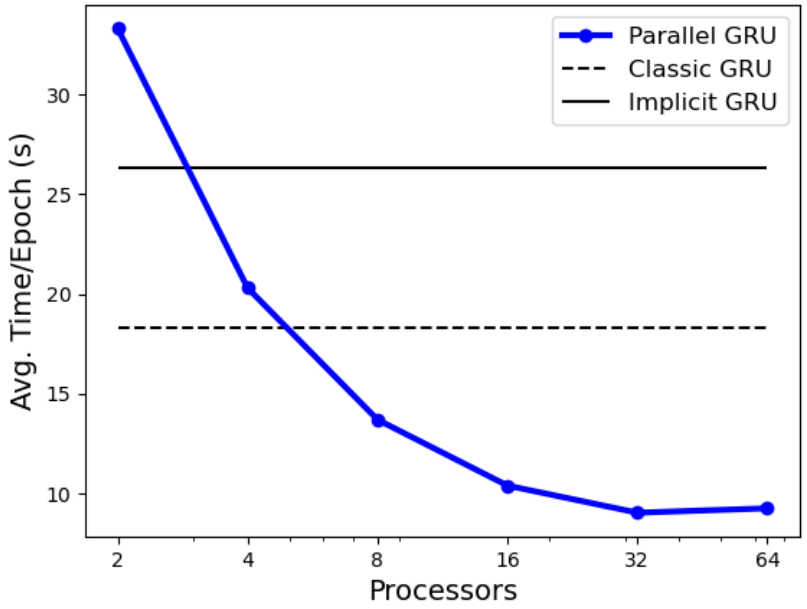
https://www.youtube.com/watch?v=XOEN9W05_4A

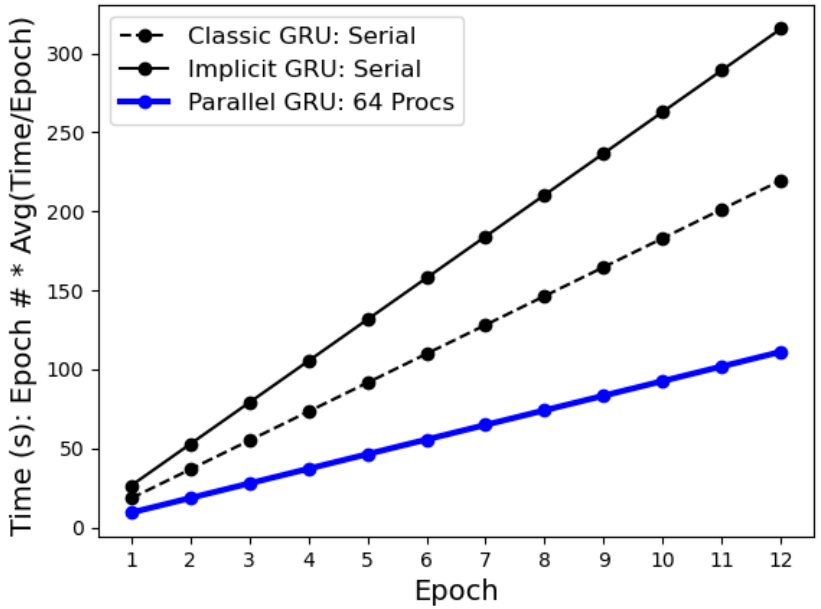
1. Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra and Jorge L. Reyes-Ortiz. A Public Domain Dataset for Human Activity Recognition Using Smartphones. 21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2013. Bruges, Belgium 24-26 April 2013.
2. https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones

Classic/Implicit/Parallel GRU Comparisons

- Parallel speedup of 2x
  - ➢ **Very small** problem, Amdahl Law limited
- All three methods have reasonable accuracy
  - ➢ Slight degradation for Implicit, and Parallel
- Comparing inference serial (blue) and parallel inference (red) for a network trained in parallel
  - ➢ Similar forward accuracy







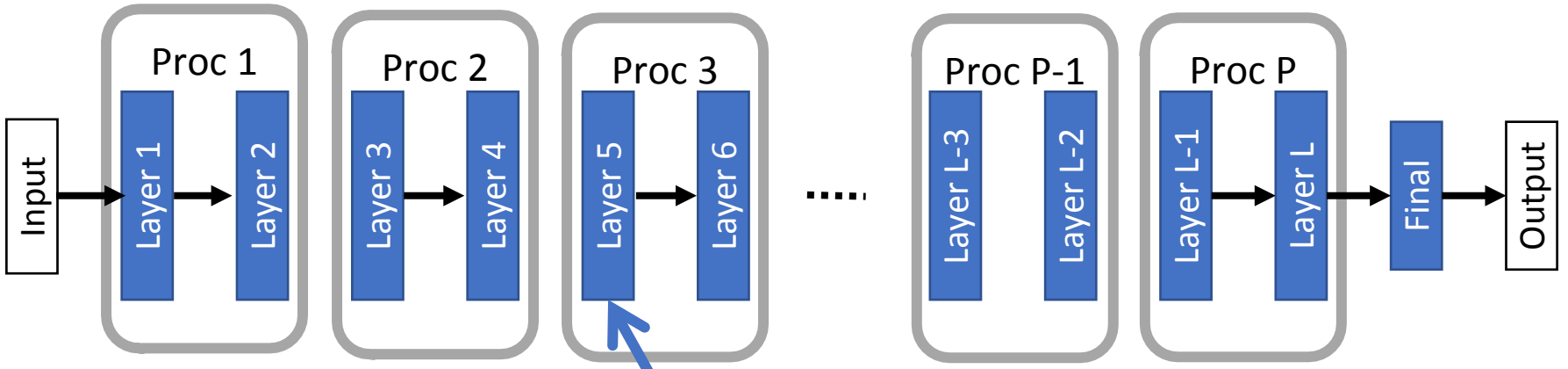
What about initial guesses?

Serial Training:

- Weights: Many different ways – Glorot 2010, He 2016, “Box” 2020
- Features: Defined by evolution both backward and forward

Layer-Parallel:

- Weights: Same way as in serial? Is there something “better”
- Features: Tricky, what is natural guess? What about for backprop?



What is the input to Layer 5 at the beginning of the Layer-Parallel algorithm?

Glorot, Bengio, 2010; He, Zhang, Ren, Sun, 2015; Cyr, Gulian, Patel, Perego, Trask, “Box”, 2020

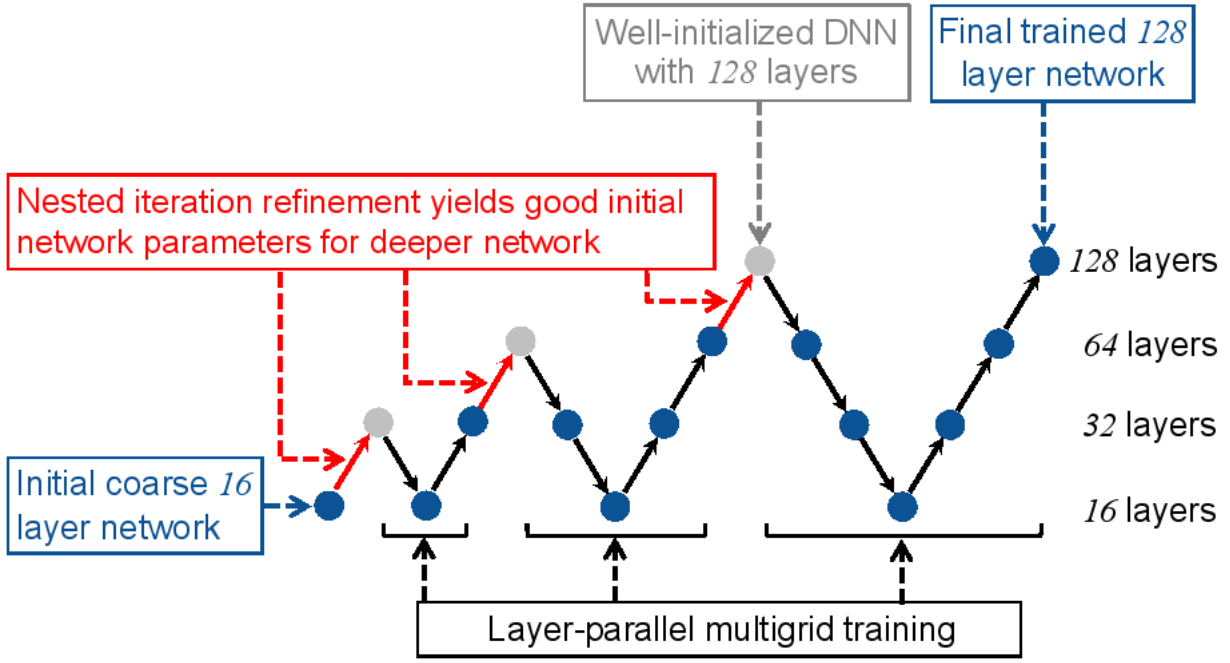
Layer-Parallel Initialization: Nested Iteration

Initialization of Layer-Parallel is complex

- Initialize weights and biases
- Initialize state and adjoint

To overcome this, we have developed a nested iteration

- Like full multigrid
- Train on the coarse network first, then upscale



Layer-Parallel Initialization: Nested Iteration

Algorithm 1 nested_iter($u^{(L-1)}$, $\square^{(L-1)}$, L, $\{m^{(l)}\}$)

1: . *Loop over nested iter. levels, then optimization iter.*
2: Initialize $u^{(L-1)}$, $\square^{(L-1)}$
3: for $l = L - 1,\ l > 0,\ l \mathrel{-}= 1$ do
4: for $i = 0,\ i < m^{(l)},\ i \mathrel{+}= 1$ do
5: $u^{(l)}, \square^{(l)} \leftarrow LPT(u^{(l)}, \square^{(l)}, d)$. *LPT: Layer-parallel training*
6:
7: end for
8: $\square^{(l-1)} = P^{(l)} \square^{(l)}$. *Interpolate*
9: end for
10: return $\square^{(0)}$. *Return finest level weights*

Initialization on coarse level (see below)

For each level (L=0 is fine)

$m^{(l)}$ optimization iterations

Layer-Parallel Iteration: Forward/Backward (Computational Kernel)

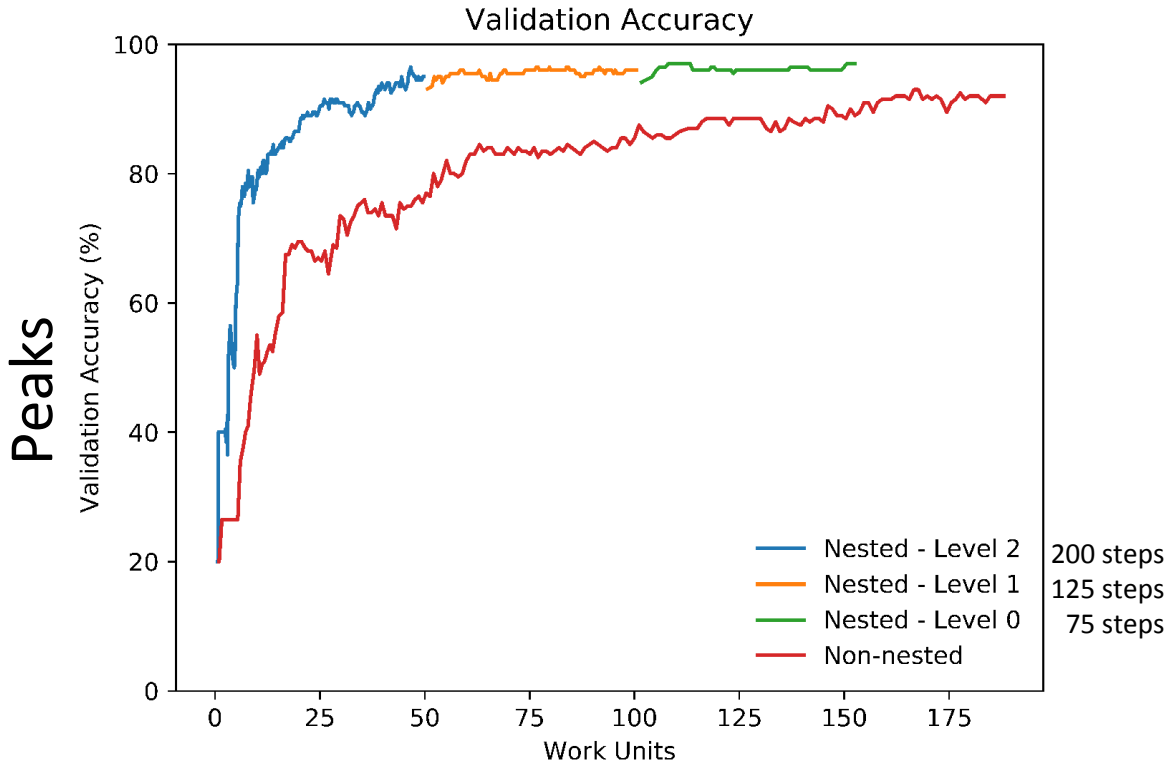
Piecewise Constant transfer to finer level

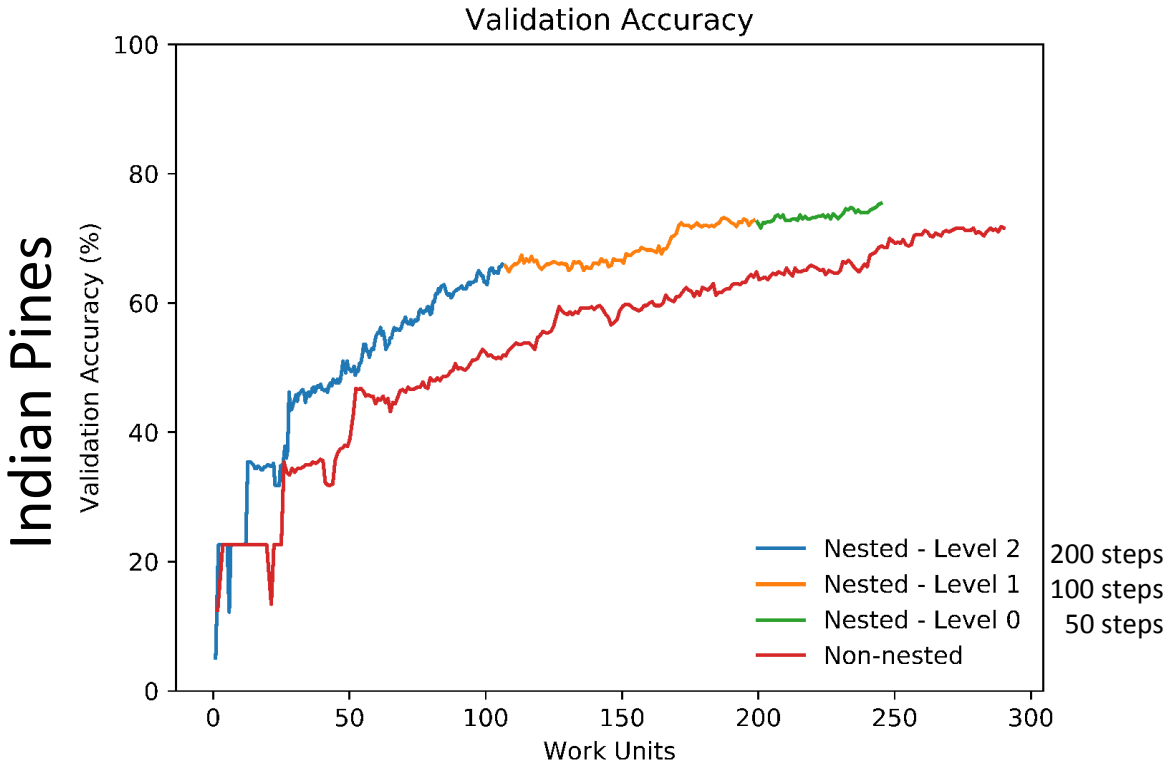
Initialization on the coarse level:

- Weights: Random
- Features: Coarse level runs serially, no initialization is necessary

Nested Iteration: Indian Pines and Peaks

- 3 level example with Indian Pines and Peaks data sets
- Work Unit = Average Fine Level forward/adjoint gradient computation





Nested iteration yields better validation accuracy in less time

Nested Iteration: Regularization

To understand the regularization impact of nested iteration

- 4 different values for hyper parameters, chosen to give good results

Tikanov Regularization	10^{-5}	10^{-7}
Initial Weights	0.0	10^{-6}

- 12 independent runs for each hyper parameterization (48 total runs)

Nested Iteration validation accuracy less sensitive than non-nested iteration

- Promising improvement to robustness (not definitive)
- **Hypothesis**: nested iteration applies implicit regularization

Peaks Validation Accuracy

	5 Channel	
	Nested	Non-Nested
Mean	86.7%	85.0%
Median	88.0%	88.5%
Max	97.0%	95.0%
Min	66.0%	20.0%
Std. Dev	7.69%	11.7%

	8 Channel	
	Nested	Non-Nested
Mean	92.3%	90.7%
Median	94.0%	91.8%
Max	99.0 %	96.5%
Min	72.5 %	57.0%
Std. Dev	5.18 %	6.08 %

Introducing Torchbraid (v0.1)

Original “Layer-Parallel” code was C++ with hand rolled kernels

- Effective research code (thanks Stefanie)
- Performance of convolutional kernels is suspect (blame me)
- Hard to do apples-to-apples comparisons with state of the art
- Not as easy to extend as PyTorch (and TensorFlow)

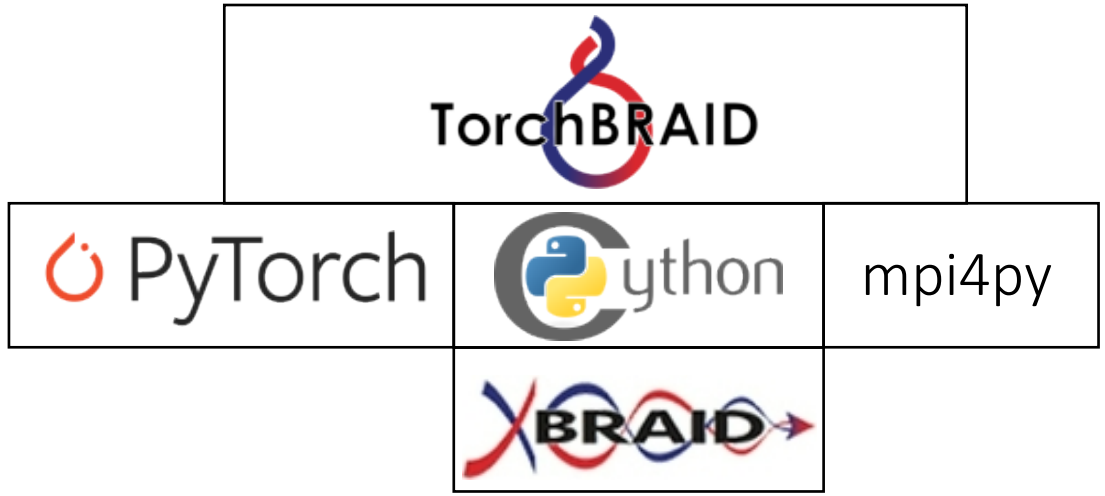
Torchbraid: Adding Layer-Parallel module to PyTorch

- Leverage more developers
- Uses automatic differentiation
- Currently has support for ODENets
- Recurrent networks under development

Torchbraid Arch. (v0.1): An Evolving Library

`LayerParallel` – A PyTorch `Module` for parallel training

- Follows ODENet and ResNet (He 2016) nomenclature
- Supports automatic differentiation
- Memory/performance tradeoffs under study
- Limited testing of different problems

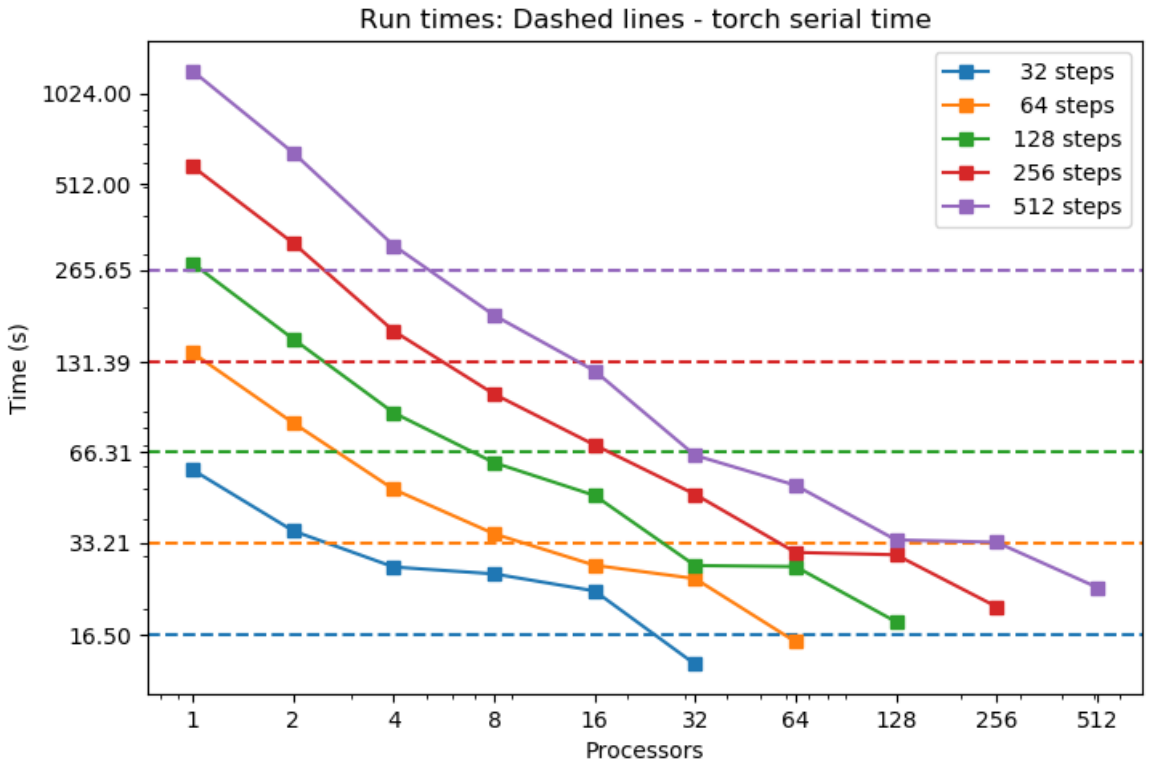



```
from torchbraid import LayerParallel
...
parallel_nn = LayerParallel(comm,                # mpi4py Communicator
                            basic_block,         # Lambda building a PyTorch module
                            local_num_steps,     # Processor local number of steps
                            Tf)                  # Final time value
```

Layer-Parallel Forward Prop

Running forward propagation:

- ODE Network with N Steps
- Each step contains 2 convolutional layers
- 3x3 convolutions on 256x256 layers
- 16 convolutions per layer
- Batch size of 16 images
- 2 Layer-Parallel Sweeps
- Dashed lines: pyTorch serial



Take Home: Torchbraid `LayerParallel` gives to speedups against PyTorch serial time

TorchBRAID

Closing Thoughts

Presented a Layer-Parallel algorithm for training deep NNs

- **Parallelism is exposed by permitting inexact propagation**
- We trade inexactness for performance with multigrid algorithms
- Developed new recurrent neural network parallel training procedure
- Initialization of state and weights using nested iteration
- Presented first “TorchBraid” result: faster forward prop

Layer-Parallel Papers:

- Guenther, Ruthotto, Schroder, Cyr, Gauger, Layer-Parallel Training of DNNs, SIMODs, 2020
- Cyr, Guenther, Schroder, Nested Iteration Initialization of DNNs, Accepted to PinT Proceedings, 2020
- Moon, Cyr, Working Title: Parallel Training of GRU with a Multi-Grid Solver for Very Long Sequences, In Preparation, 2021

U.S. DEPARTMENT OF ENERGY | Office of Science

Thanks to the DOE Office of Science ASCR Early Career Research Program for supporting this work!