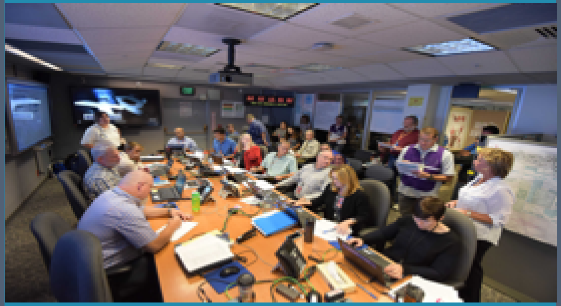
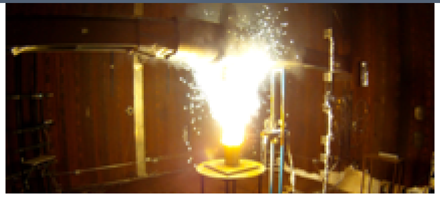
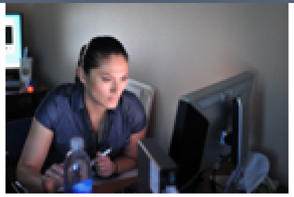
SAND2022-11168C

Adapting Multi-Grid-in-Time to Train Deep Neural Networks







Eric C. Cyr, Sandia National Laboratories

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

Neural Networks

A neural network is a parameterized model:

Neural Network $\longrightarrow$ $$\mathcal{N}\mathcal{N}(x;\Theta) \to y$$ $\longleftarrow$ Output

Input Parameters

It is composed of multiple layers*

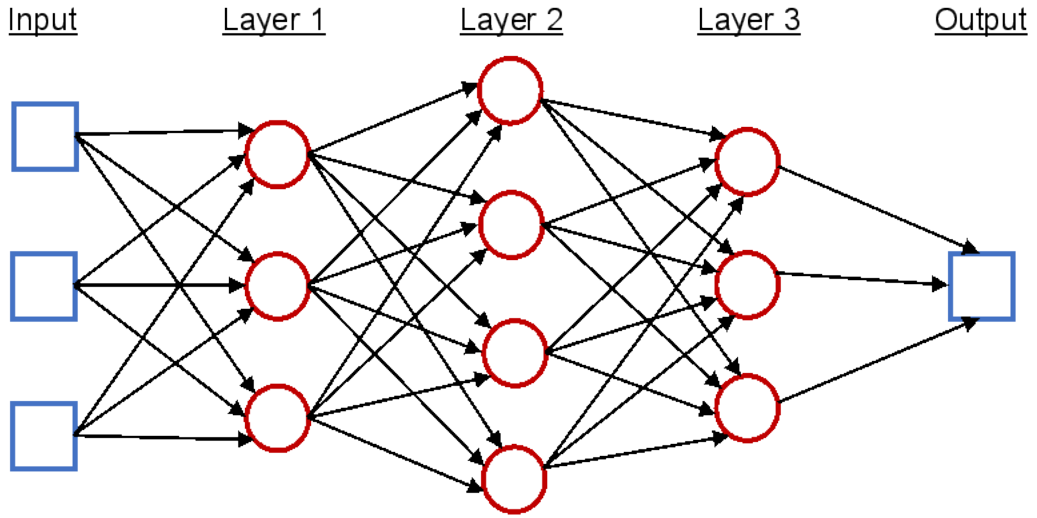
Feature Vectors

$$u_1 = A_0 x + b_0,$$
$$u_{i+1} = f(u_i; \{A_i, b_i\}) \quad i = 1 \ldots L-1,$$
$$y = A_L u_L;$$
$$\Theta = \{A_i, b_i\}_{i=0}^{L-1} \cup \{A_L\}$$

*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** :

$$\{(\underset{\text{Input}}{x_i}, \underset{\text{Output}}{y_i})\}$$

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

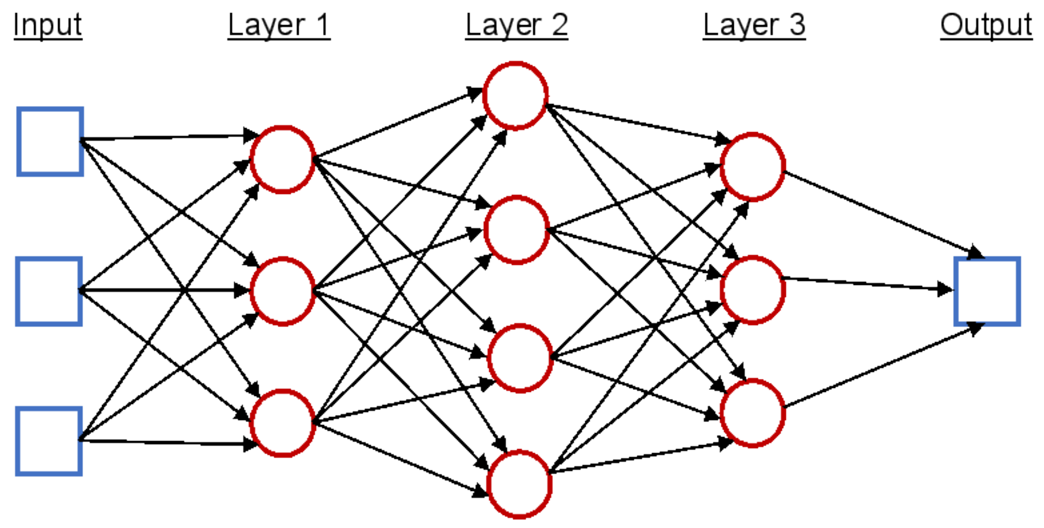
$$\underset{\text{Parameters}}{\min_{\Theta}} \sum_{n=1}^{N} \text{Loss}\left(\mathcal{NN}(x_n;\Theta), y_n\right)$$

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:



$$u_0 \longrightarrow u_1 \longrightarrow u_2 \longrightarrow u_3 \longrightarrow u_4$$

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

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent Algorithm:

```
# initialize the weights/biases
w_W = initialize_W()
w_b = initialize_b()

for epochs in [1,max_epochs]:
  # sample the data in batches
  for y_batch in data.get_batches(samps_per_batch()):
    # inference step - forward propagation
    x = forward_prop(y_batch,w_W,w_b)

    # compute gradient - backward propagation
    g_W,g_b = backward_prop(x,y_batch,w_W,w_b)

    # update the weights/biases
    w_W = w_W - learning_rate * g_W
    w_b = w_W - learning_rate * g_W
```

Each step computes gradient from a subset (batch) of the data selected at random:

$$\nabla_\Theta \left(\frac{1}{\underset{\uparrow}{N_b}} \sum_{n=1}^{N_b} \text{Loss}(\mathcal{N}\mathcal{N}(x_{b,n}, \Theta), y_{b,n}) \right)$$

Batch Size

Randomly selected data

Batched SGD samples from the data space defining the global loss

- Reduces required memory footprint
- Uses samples more efficiently (see Bottou, Curtis, Nocedal)

Original SGD paper: Robbins, Monro. "A stochastic approximation method." *The annals of mathematical statistics* (1951): 400-407.

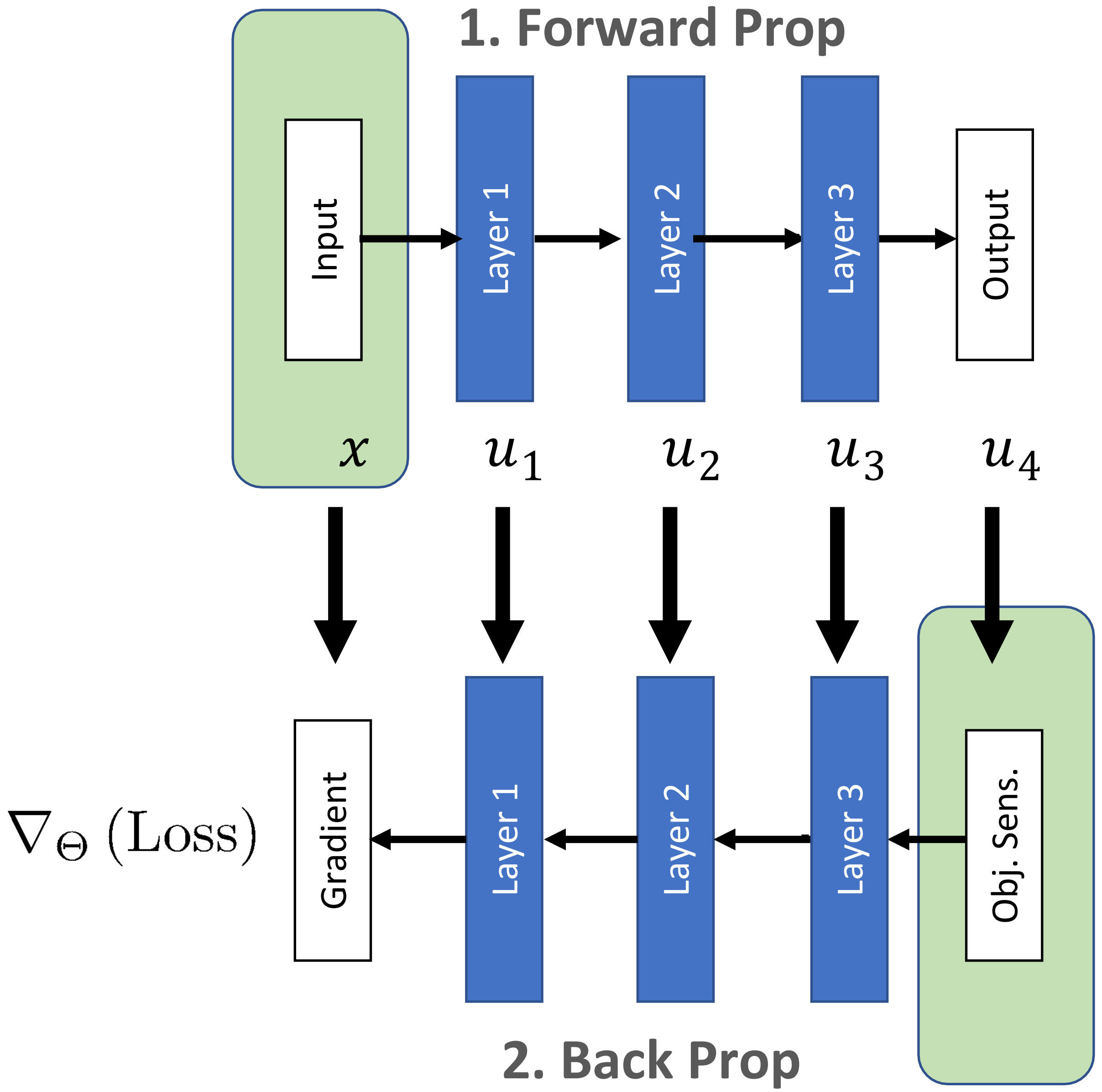
SGD: Forward and Backward Propagation

Stochastic Gradient Descent Algorithm:

```
# initialize the weights/biases
w_W = initialize_W()
w_b = initialize_b()

for epochs in [1,max_epochs]:
  # sample the data in batches
  for y_batch in data.get_batches(samps_per_batch()):
    # inference step - forward propagation
    x = forward_prop(y_batch,w_W,w_b)

    # compute gradient - backward propagation
    g_W,g_b = backward_prop(x,y_batch,w_W,w_b)

    # update the weights/biases
    w_W = w_W - learning_rate * g_W
    w_b = w_W - learning_rate * g_W
```

Batched SGD samples from the data space defining the global loss

- Reduces required memory footprint
- Uses samples more efficiently (see Bottou, Curtis, Nocedal)



Original SGD paper: Robbins, Monro. "A stochastic approximation method." *The annals of mathematical statistics* (1951): 400-407.

SGD Works

Assumptions

Lower bounded Objective:

$$F^* \leq F(\Theta) \quad \forall \Theta$$

Unbiased gradient estimator:

$$\mathbb{E}[g(\Theta_k, \xi_k) | \Theta_k] \overset{a.s.}{=} \nabla F(\Theta_k)$$

Lipshitz Cont. Gradient:

$$\|\nabla F(\Theta_u) - \nabla F(\Theta_v)\| \leq L \|\Theta_u - \Theta_v\|$$

Gradient estimator has bounded variance:

$$\mathbb{E}\left[\|g(\Theta_k, \xi_k) - \nabla F(\Theta_k)\|^2 | \Theta_k\right] \overset{a.s.}{\leq} \nu^2$$

Theorem (Ghadimi,Lan,2013;Lan,2020): For a nonconvex objective, with the above assumptions, and $\alpha_k < 2/L$, then

$$\sum_{k=1}^{K} (\alpha_k - \frac{1}{2} L \alpha_k^2) \mathbb{E}[\|\nabla F(\Theta_k)\|^2] \leq F(\Theta_1) - F^* + \frac{1}{2} L \nu^2 \sum_{k=1}^{K} \alpha_k^2$$

Take Home: SGD leads to small expected gradients

1. Ghadimi, Lan, “Stochastic first- and zeroth-order methods for nonconvex stochastic programming,” *SIAM J. Optim.*, 2013.
2. Lan, “First-order and Stochastic Optimization Methods for Machine Learning,” *Springer Series in the Data Sciences*, 2020.

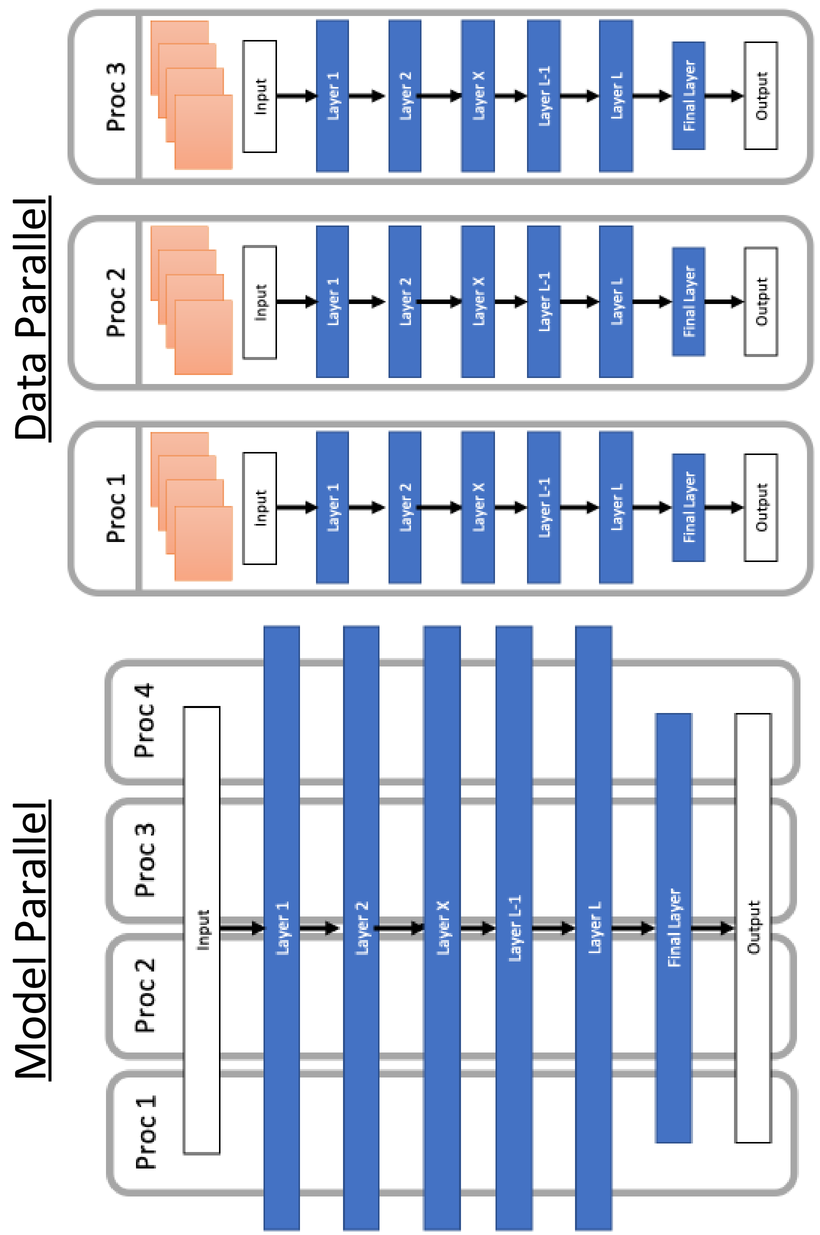
Use parallelisms to accelerate training

Parallel computing is important to training

- Focused on GPU level parallelism
- Relatively small clusters

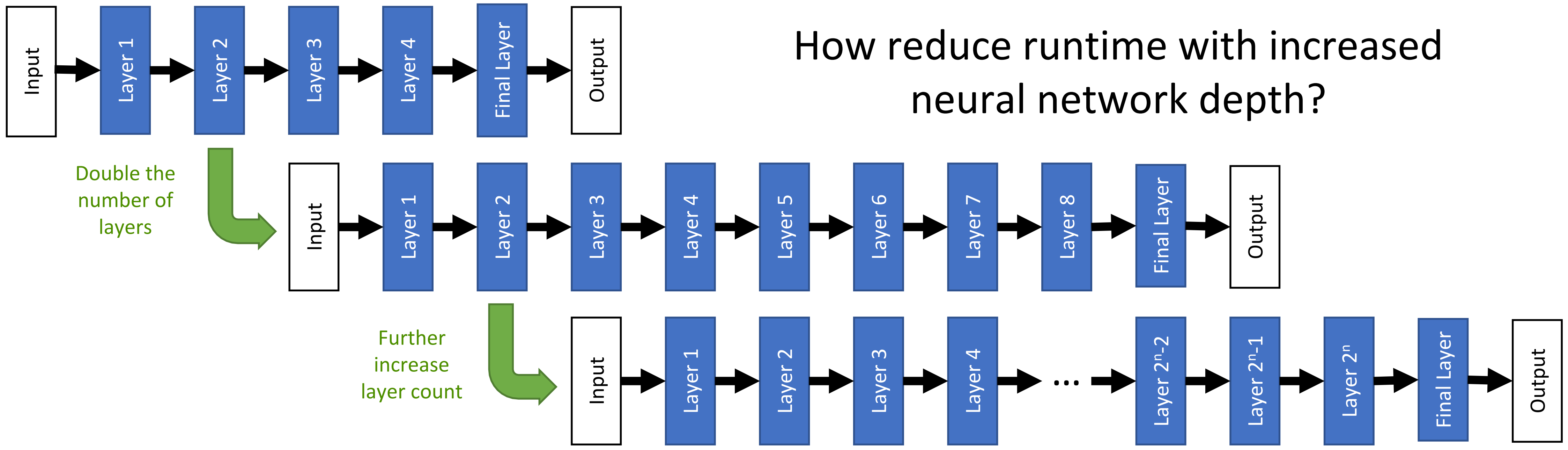
Parallel scalability relies on

- Spatial/Model parallelism
- Data parallelism
- Parallelism handles increased data set size and network width



Review of model/data parallelism: Ben-Nun, Hoefler. "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis." *ACM Computing Surveys (CSUR)* 52, 2019.

What about depth?



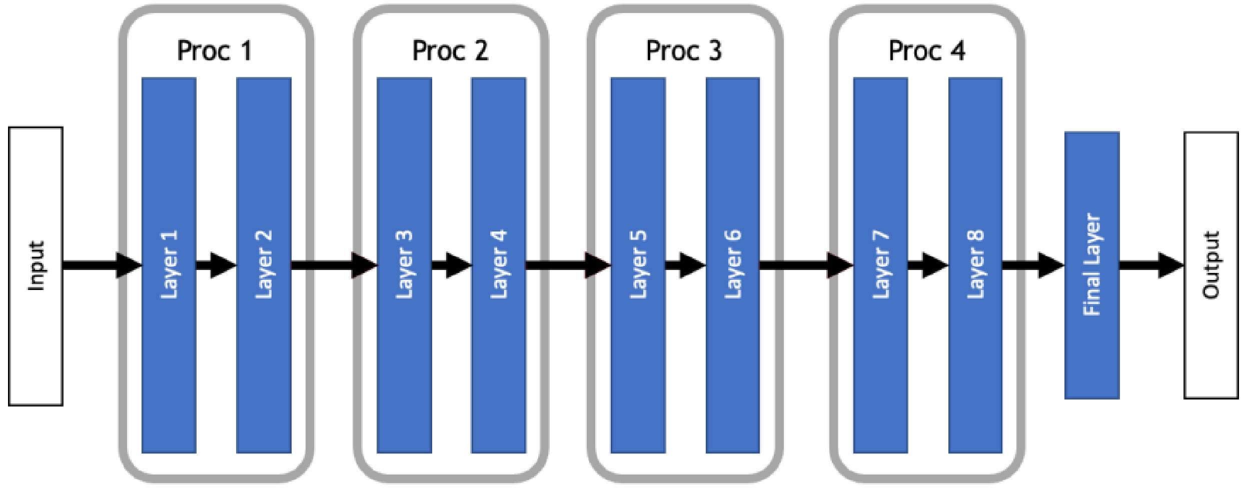
How reduce runtime with increased neural network depth?

Potential Architectures:
- ResNet’s
- NeuralODEs
- Recurrent Neural Networks

Current parallelism mitigates increasing data set sizes and network width:

Increased runtimes with depth are not reduced by traditional approaches!

Our New Approach: Layer-Parallel Training



A simple idea:

- Process each layer in parallel
- This will distribute computation

But it won't work:

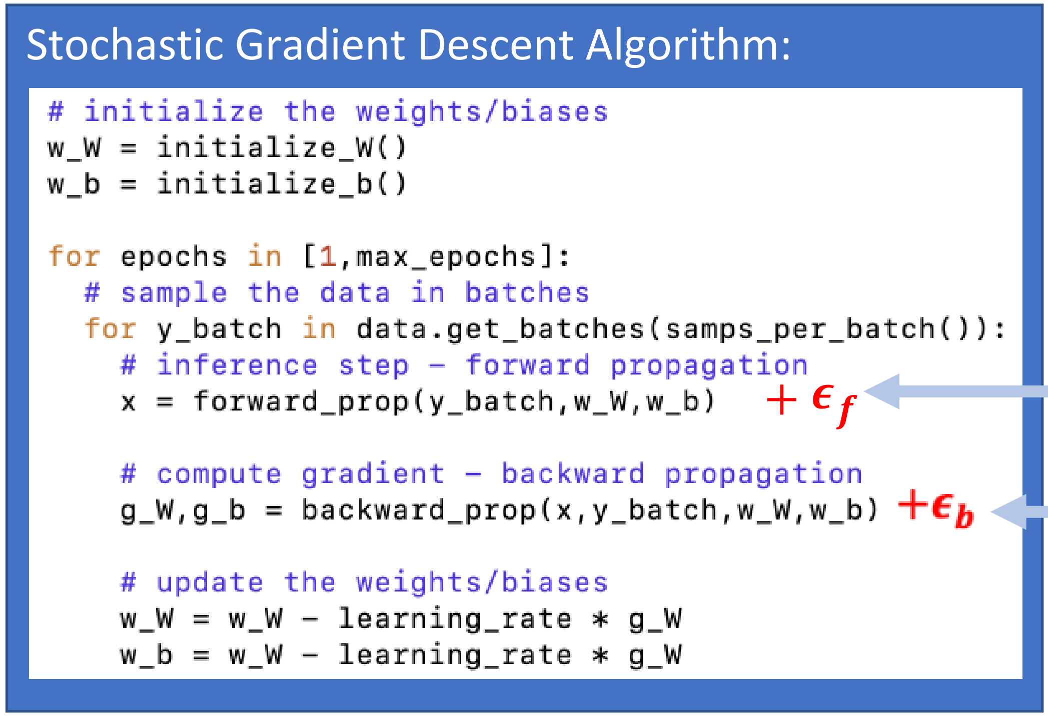
- Forward/Back prop are serial
- Distributing the layers does not lead to acceleration

Stochastic Gradient Descent Algorithm:

```
# initialize the weights/biases
w_W = initialize_W()
w_b = initialize_b()

for epochs in [1,max_epochs]:
  # sample the data in batches
  for y_batch in data.get_batches(samps_per_batch()):
    # inference step - forward propagation
    x = forward_prop(y_batch,w_W,w_b)

    # compute gradient - backward propagation
    g_W,g_b = backward_prop(x,y_batch,w_W,w_b)

    # update the weights/biases
    w_W = w_W - learning_rate * g_W
    w_b = w_W - learning_rate * g_W
```

Critical Assumption: Exactness of propagation

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



Stochastic Gradient Descent Algorithm:

```
# initialize the weights/biases
w_W = initialize_W()
w_b = initialize_b()

for epochs in [1,max_epochs]:
  # sample the data in batches
  for y_batch in data.get_batches(samps_per_batch()):
    # inference step - forward propagation
    x = forward_prop(y_batch,w_W,w_b)   + ϵ_f

    # compute gradient - backward propagation
    g_W,g_b = backward_prop(x,y_batch,w_W,w_b) +ϵ_b

    # update the weights/biases
    w_W = w_W - learning_rate * g_W
    w_b = w_W - learning_rate * g_W
```

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

SGD With Inexact Gradients

Assumptions

Objective Constraints:

$$F^* \le F(\Theta) \quad \forall \Theta$$
$$\|\nabla F(\Theta_u) - \nabla F(\Theta_v)\| \le L\|\Theta_u - \Theta_v\|$$

Summable conditional biases:

$$\sum_{k=1}^{K} \alpha_k \mathbb{E}\left[\|\mathbb{E}[\tilde{g}(\Theta_k, \xi_k)|\Theta_k] - \nabla F(\Theta_k)\|\right] \le B$$

Biased gradient estimator:

$$\|\mathbb{E}[\tilde{g}(\Theta_k, \xi_k)|\Theta_k] - \nabla F(\Theta_k)\| \overset{a.s.}{\le} \Delta$$

Gradient estimator has bounded variance:

$$\mathbb{E}\left[\|\tilde{g}(\Theta_k, \xi_k) - \nabla F(\Theta_k)\|^2 |\Theta_k\right] \overset{a.s.}{\le} \nu^2$$

Theorem (Lin, 2022): For a nonconvex objective, with the above assumptions, and $\alpha_k < 1/L$, then

$$\sum_{k=1}^{K} \left(\alpha_k - L\alpha_k^2\right) \mathbb{E}[\|\nabla F(\Theta_k)\|^2] \le 2(F(\Theta_1) - F^*) + \Delta B + L\nu^2 \sum_{k=1}^{K} \alpha_k^2$$

Take Home: SGD with inexact gradients leads to small expected gradients

Neural ODEs: Layers as Time Dimension

We make one more transformation: from ResNets to NeuralODEs

$$u_{i+1} = u_i + f(u_i; \{A_i, b_i\}) \quad \Rightarrow \quad \frac{\partial}{\partial t} u = f(t, u; \{A, b\})$$

See for instance:

1. Chen, Rubanova, Bettencourt, Duvenaud. "Neural ordinary differential equations." *arXiv preprint arXiv:1806.07366* (2018).
2. Haber, Ruthotto. "Stable architectures for deep neural networks." *Inverse Problems* 34, no. 1 (2017).

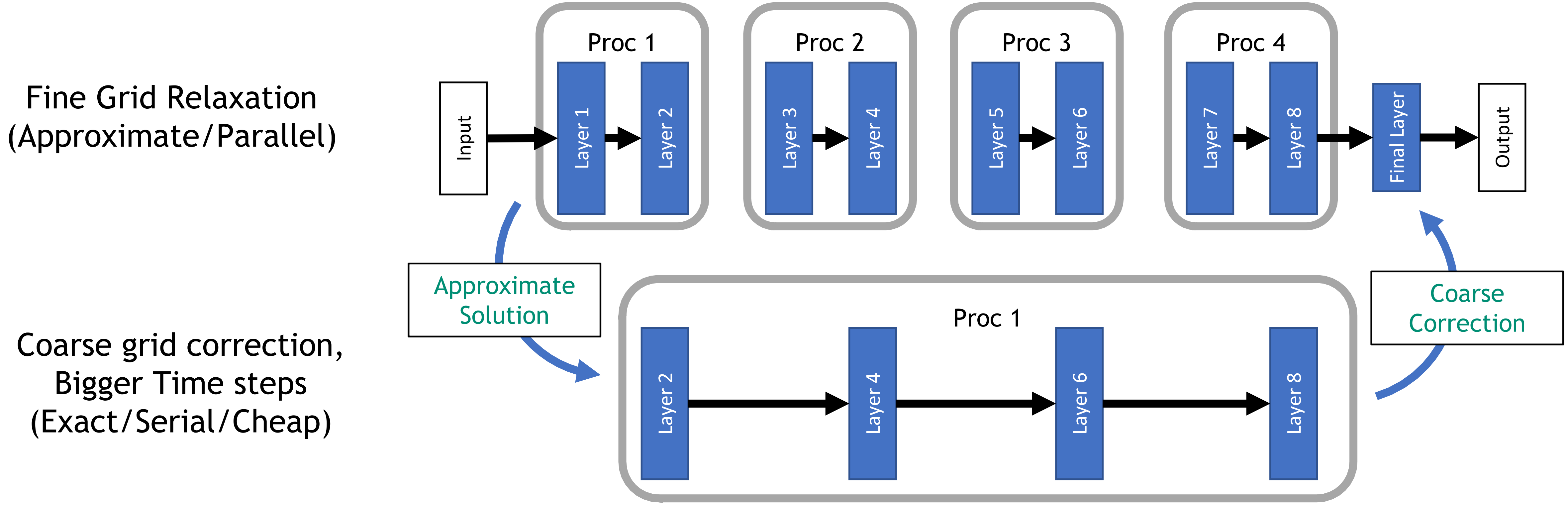
With this formulation we can develop a “Layer-Parallel” Algorithm using ”Parallel-in-Time” methods

- Parareal: *Lions, Maday, Turinici, Résolution d’EDP par un schéma en temps “pararéel”, C. R. Acad. Sci. Paris Sér. I Math. 332 2001.*
- PFASST: *Emmett, Minion. "Toward an efficient parallel in time method for partial differential equations." Communications in Applied Mathematics and Computational Science 7, 2012.*
- **MGRIT: *Falgout, Friedhoff, Kolev, MacLachlan, Schroder. "Parallel time integration with multigrid." SIAM SISC 2014.***

Layer Parallel Training – A Multigrid Approach*

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

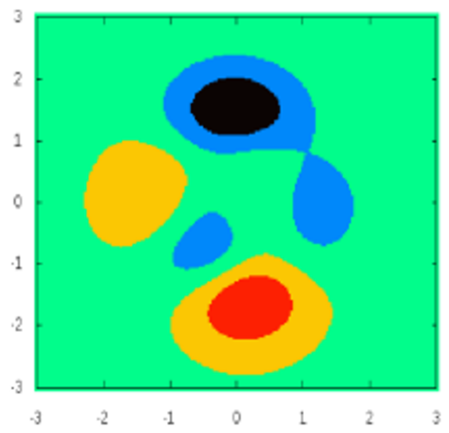
- “Fine grid relaxation”: Fixes local errors between layers – embarrassingly parallel
- “Coarse grid correction”: Fixes global errors – serial inference on smaller network



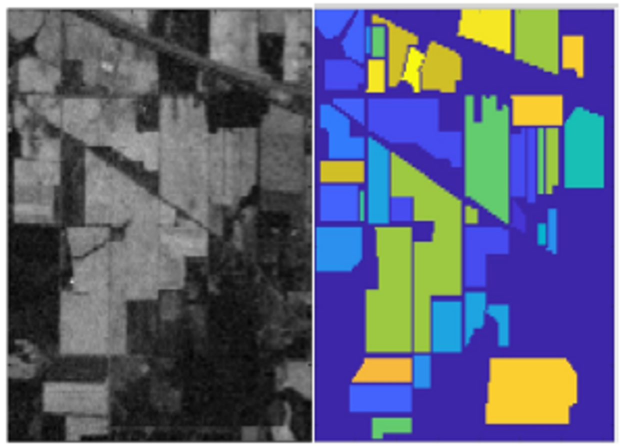
Multigrid is applied for both forward and back propagation

*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 Scaling Results



(a) Peaks



(b) Indian Pines

(c) MNIST

Three different classification problems

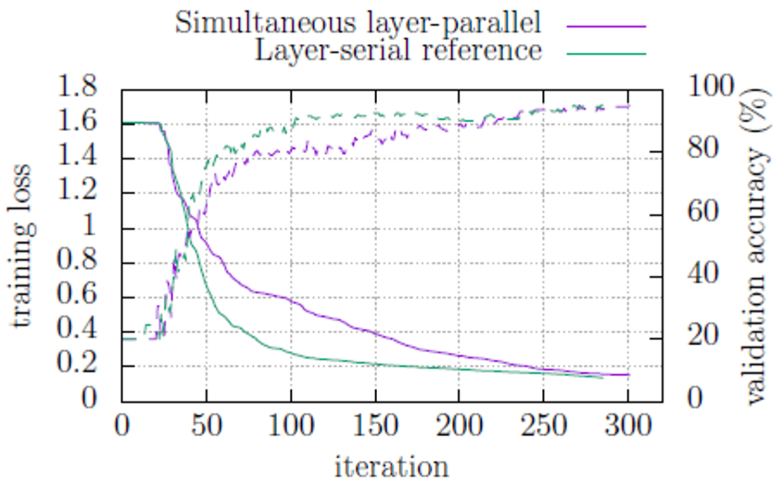
1. Peaks: Put particle position into one of 5 different classes
2. Indian Pines: Hyperspectral imaging, what crop? Soy, corn, etc…
3. MNIST: Handwritten digit classification

A comment on the code:

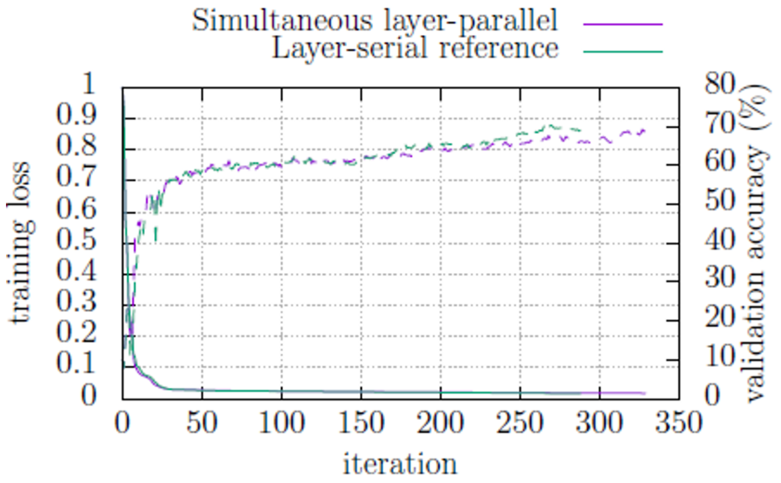
- Neural network code using Xbraid (LLNL) parallel-in-time library
- Code is not optimized: e.g. MNIST uses hand coded convolutions
- Neural networks architectures not optimized, simple ODENets

Layer Parallel Scaling Results

Peaks



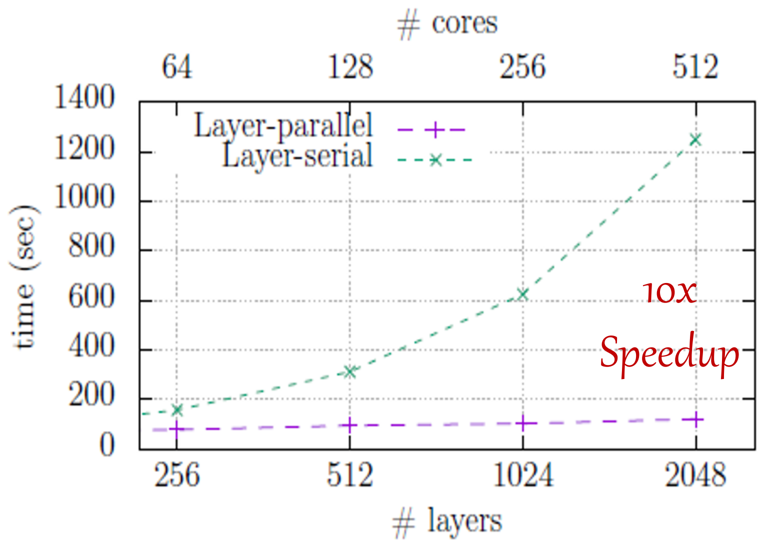
Indian Pines

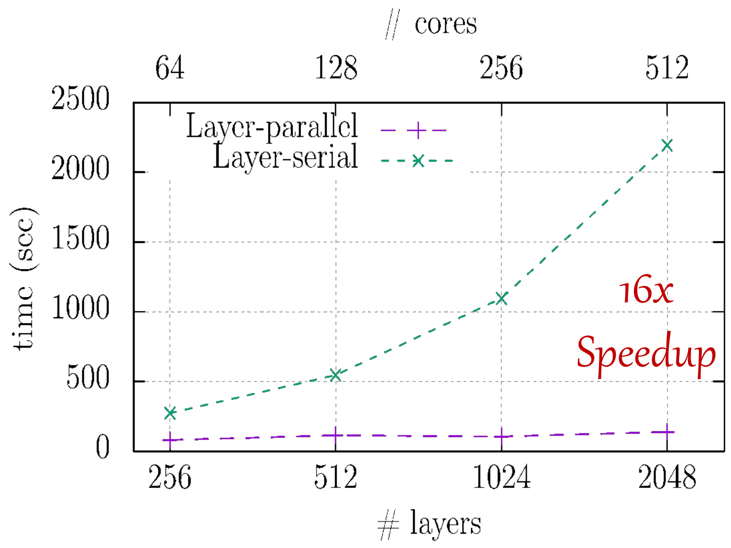


Indian Pines

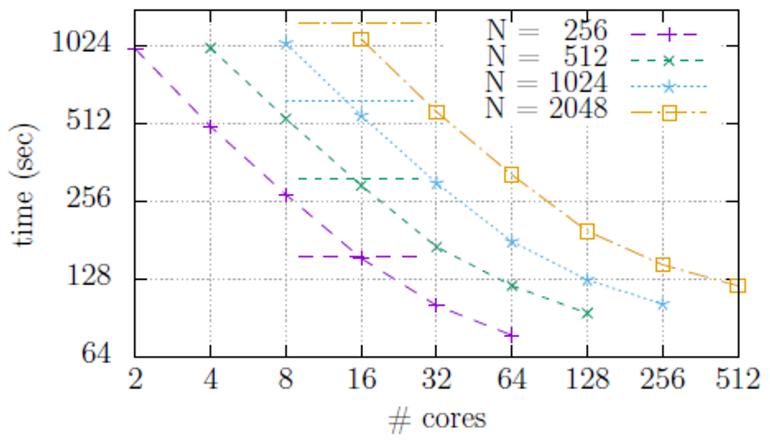
MNIST

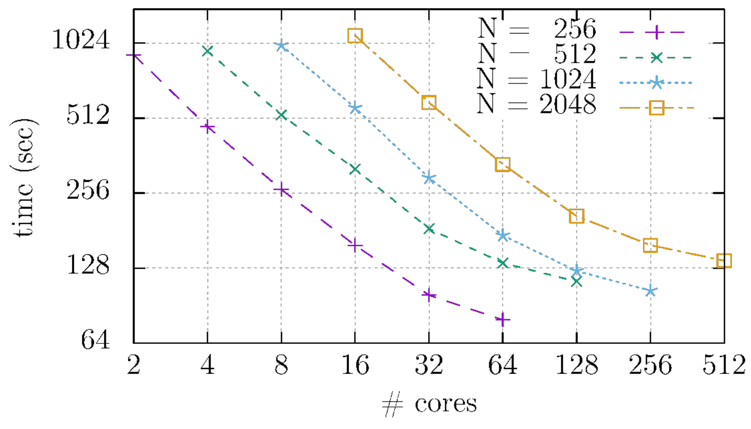
Weak Scaling





Strong Scaling

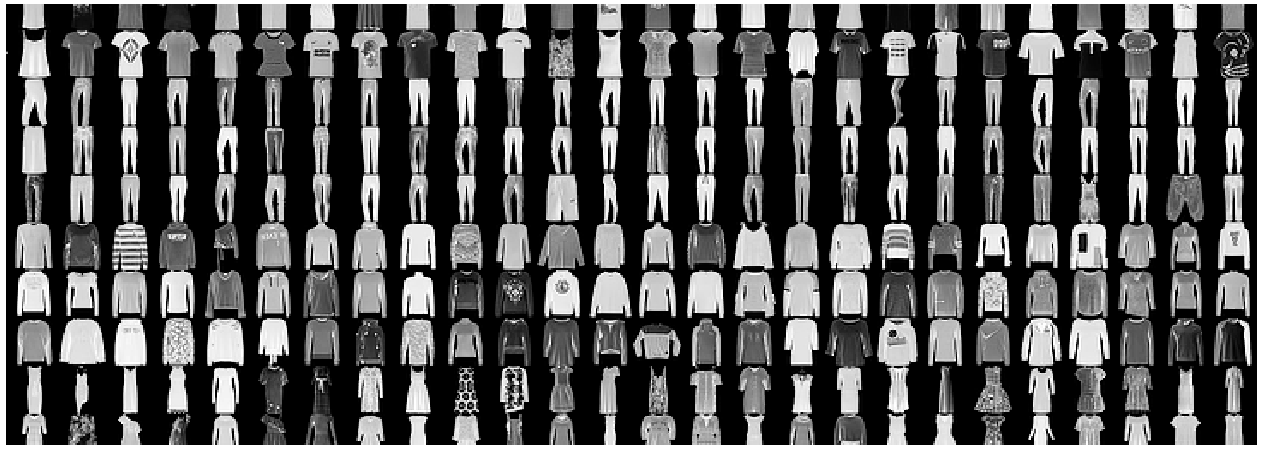


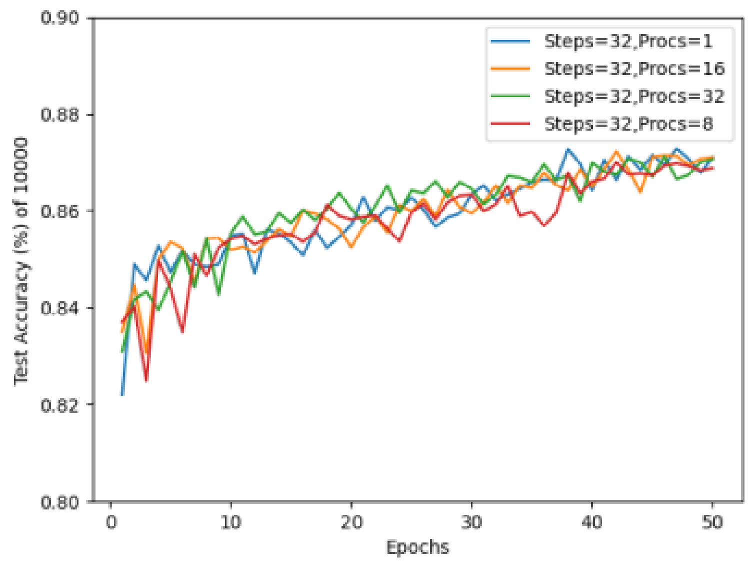


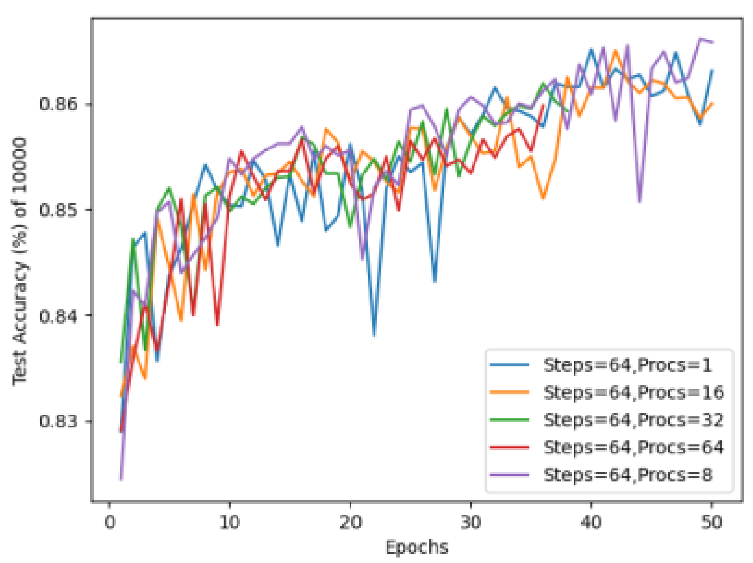
Using Stochastic Gradient Descent (SGD)

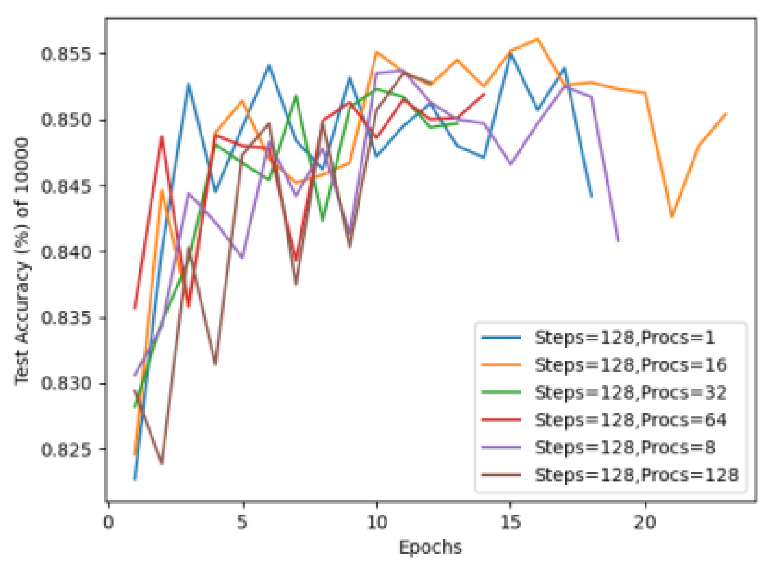
Workhorse of ML is SGD optimizer

- 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

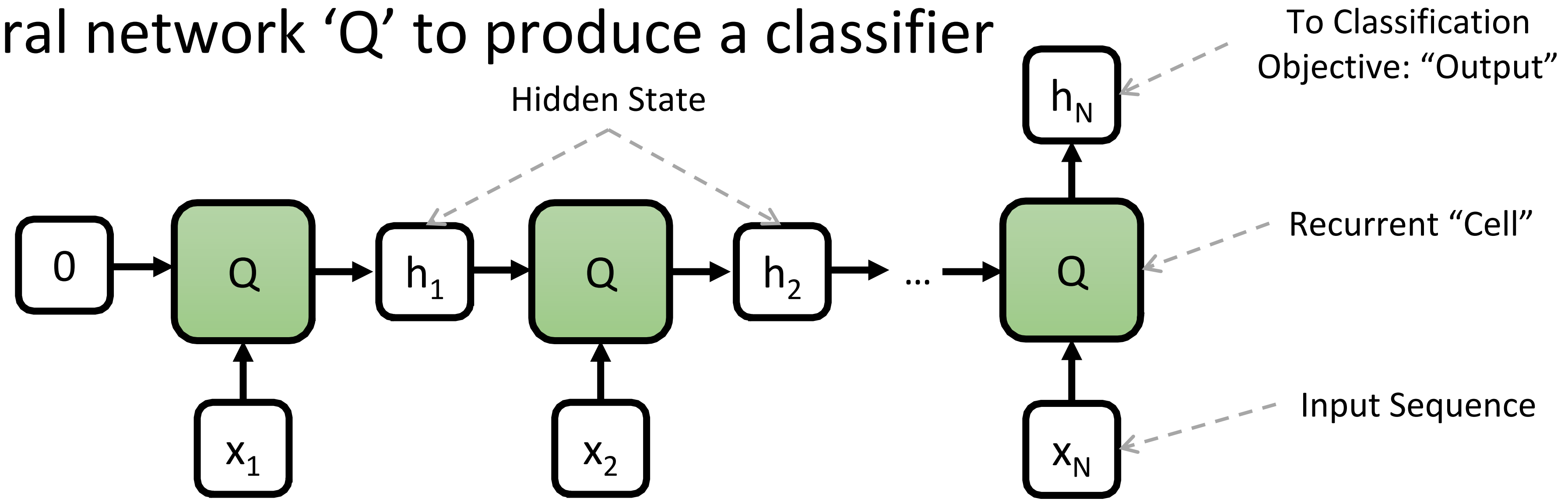
Recurrent Neural Nets (RNN)

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

$$\Phi(\underbrace{x_1, x_2 \ldots, x_N}_{\text{Sequence of N items}}) \rightarrow \underbrace{\{1, C\}}_{\text{One of C classes}}$$

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

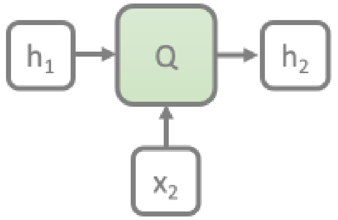
- Historically RNNs are hard to train (my read is they were unstable)
- "memory": remembers important features in the sequence
- "forget" gates: eliminates some redundant/irrelevant from the sequence

Generalized Recurrent Units:

- **h**$_*$: Hidden State,
- x$_*$: Input Sequence,
- W$_*$ and b$_*$: Learnable Network Parameters

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

Hadamard Product



GRUs to ODEs

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

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

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

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

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!
- ➢ In the NeuralODE case we took bigger time steps on coarse grids
- ➢ Here we will take $\Delta t = 1$, coarse grid will likely be unstable!

Remedy: a new "Implicit GRU", 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 MGRIT solver

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

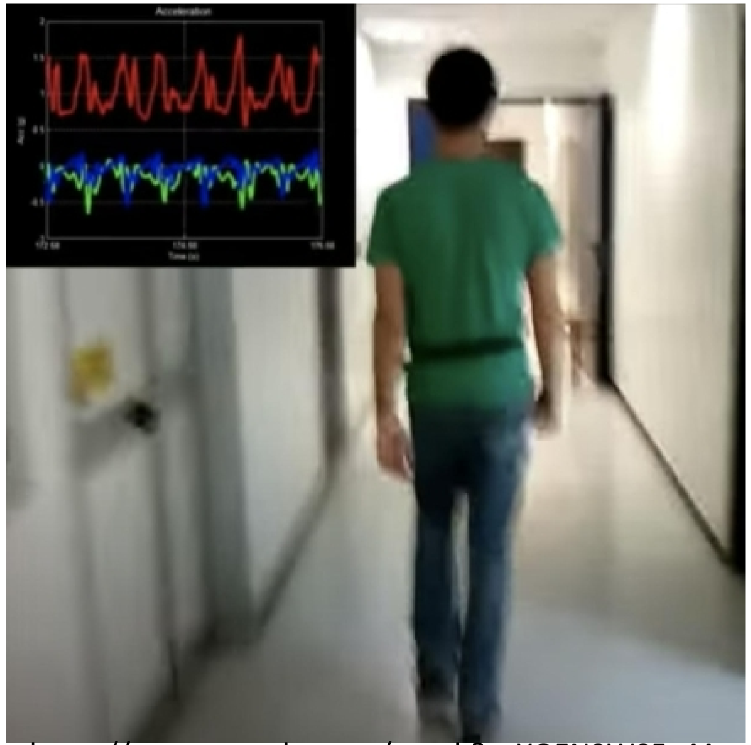
Dataset Details:

- 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 (Very Small)

- 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)

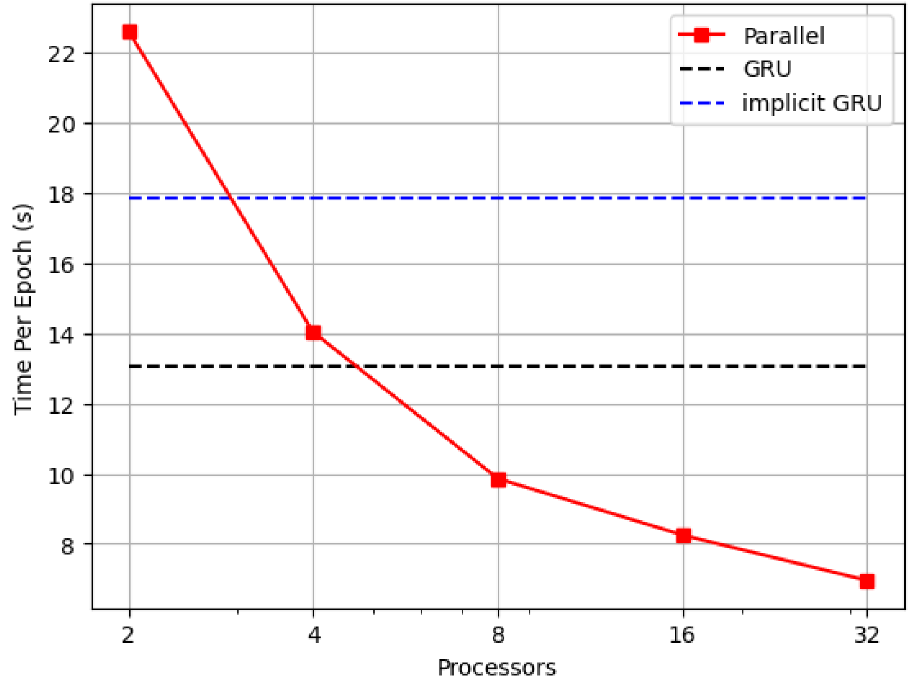


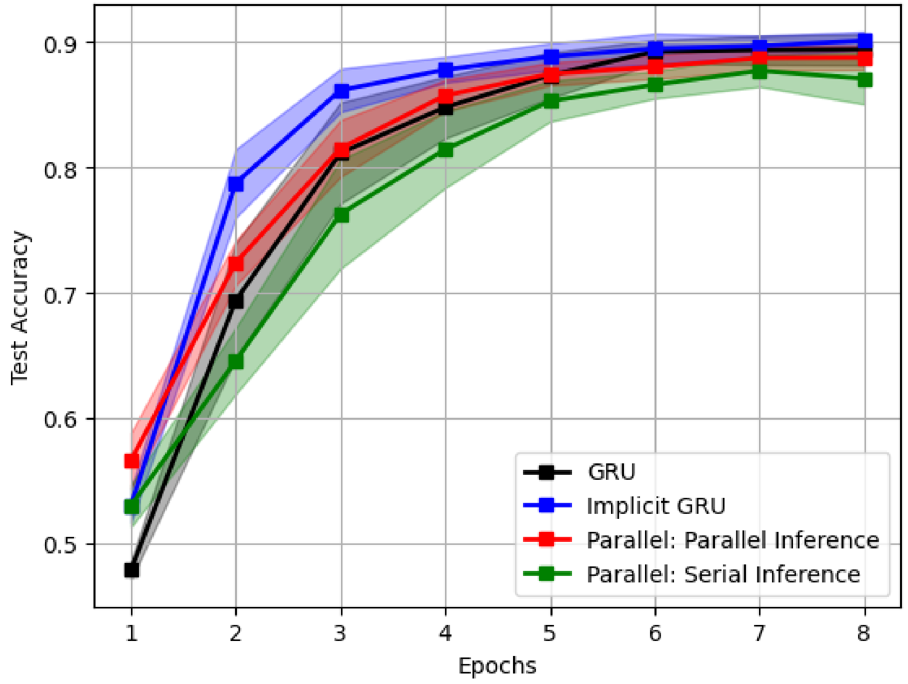
https://www.youtube.com/watch?v=XOEN9W05_4A

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





Take Home:

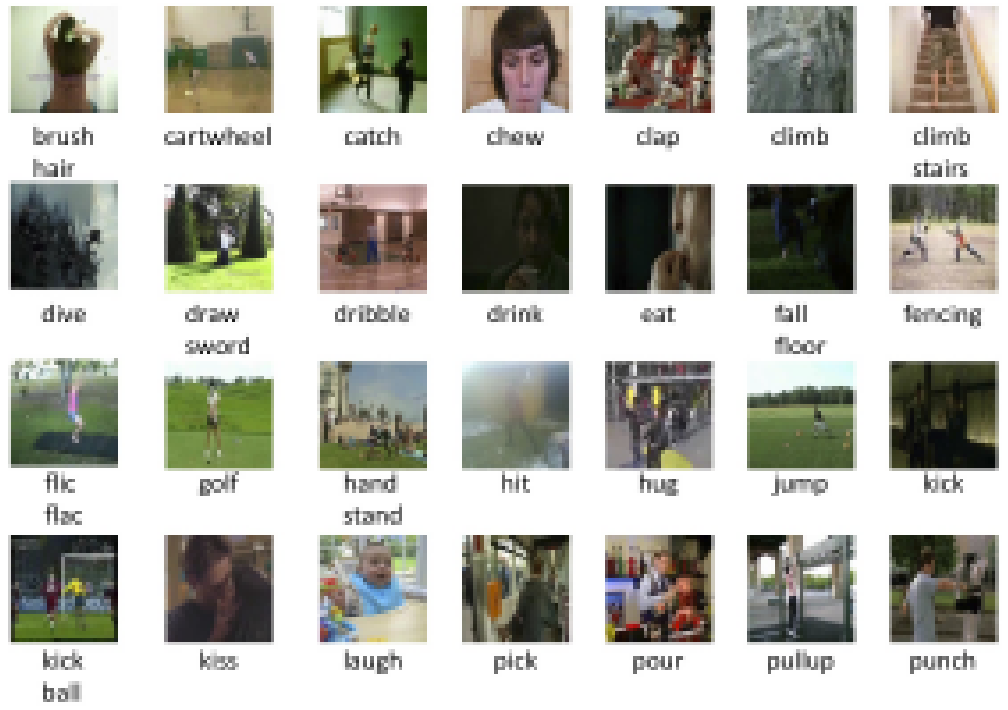
1. **Implicit GRU has accuracy is competitive with "classic"**
2. **Training in parallel has modest impact on serial inference**

HMDB51: A Large Human Motion Database[1,2]

Task: Classify human activity in each video

- Full Database:
  - ~6700 Clips Distributed in 51 Classes
  - Train/Test Split: 6053/673
  - Frame count ranges from 20 to more than 200
  - We truncate/pad each video to 128 frames
  - We use 240x240 pixels in each frame
- Subset Database (we run this):
  - 6 Classes: chew eat jump run sit walk
  - Train/Test Split: 1157/129
  - Frame count ranges from 20 to more than 200
  - We truncate/pad each video to 128 frames
  - We use 240x240 pixels in each frame

Representatives of 28 Classes



http://serre-lab.clps.brown.edu/wp-content/uploads/2012/08/HMDB_snapshot1.png

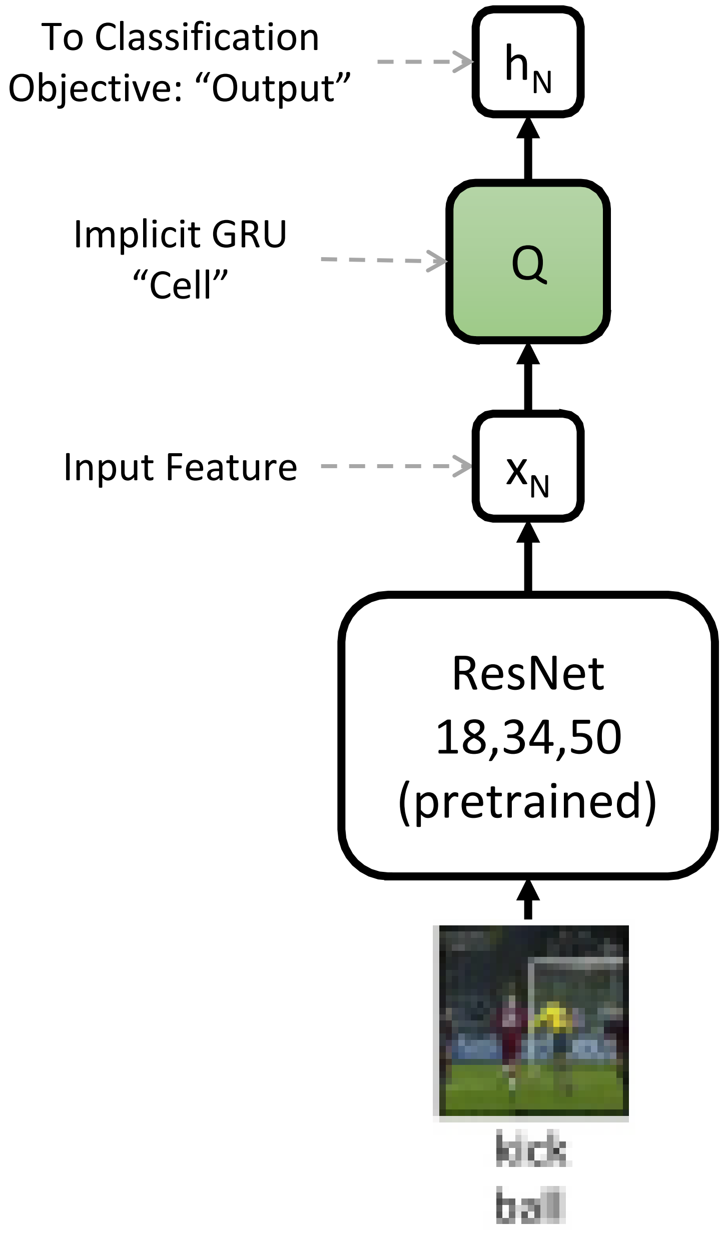
1. H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre. HMDB: A Large Video Database for Human Motion Recognition. ICCV, 2011.
2. https://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database/

Implicit GRU RNN

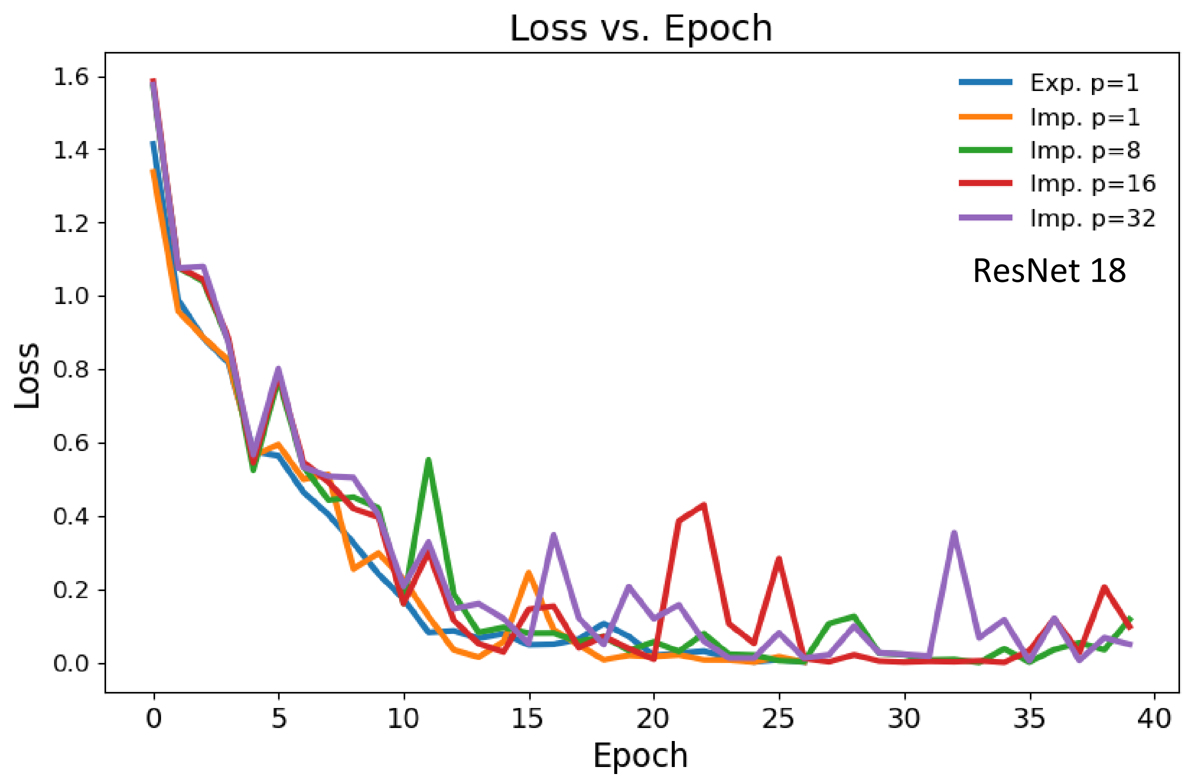
Implicit GRU with ResNet Preprocessor:
- ResNet 18, 34 or 50 (pretrained) computes 1000 features per frame
- Implicit GRU uses a hidden size of 1000, with two layers

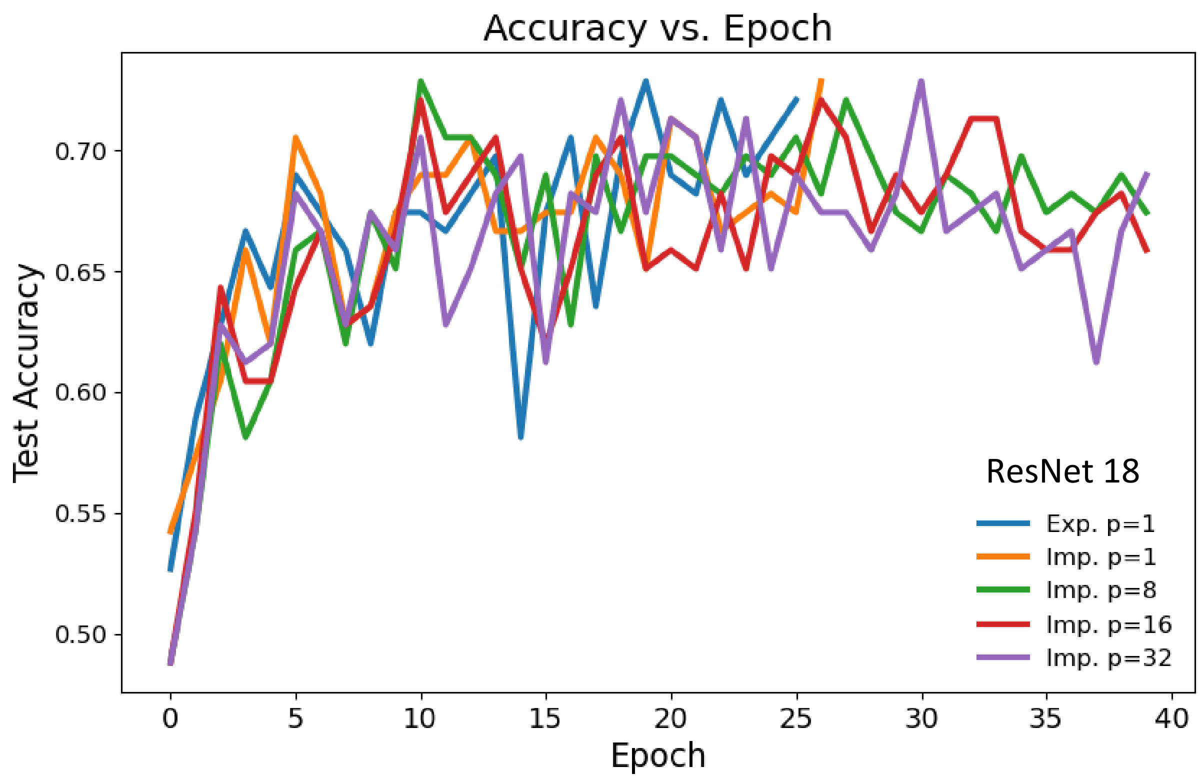
Computing Platform (Sandia's Attaway machine):
- 2.3 GHz Intel Xeon, 2 Sockets, 18 Cores each: 36 cores per node
- Run with 9 OpenMP threads per MPI rank (4 ranks per node)

Training Details:
- Batch Size of 100
- ADAM optimizer with learning rate of 10^{-3}
- ResNet18 is not applied on coarse grids
- Image feature is computed once


To Classification Objective: "Output"
h_N
Implicit GRU "Cell"
Q
Input Feature
x_N
ResNet 18,34,50 (pretrained)
kick ball

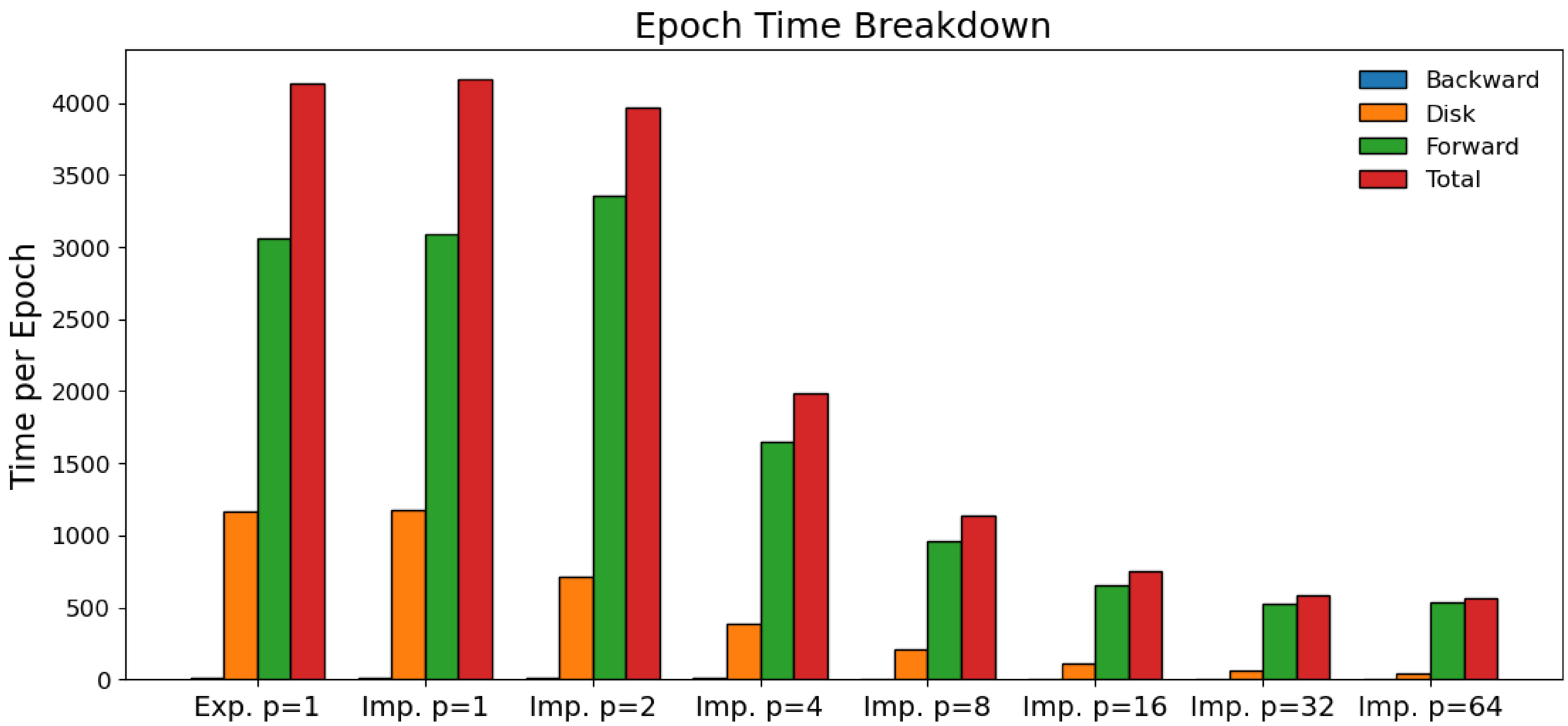
Subset: Training Loss and Test Accuracy



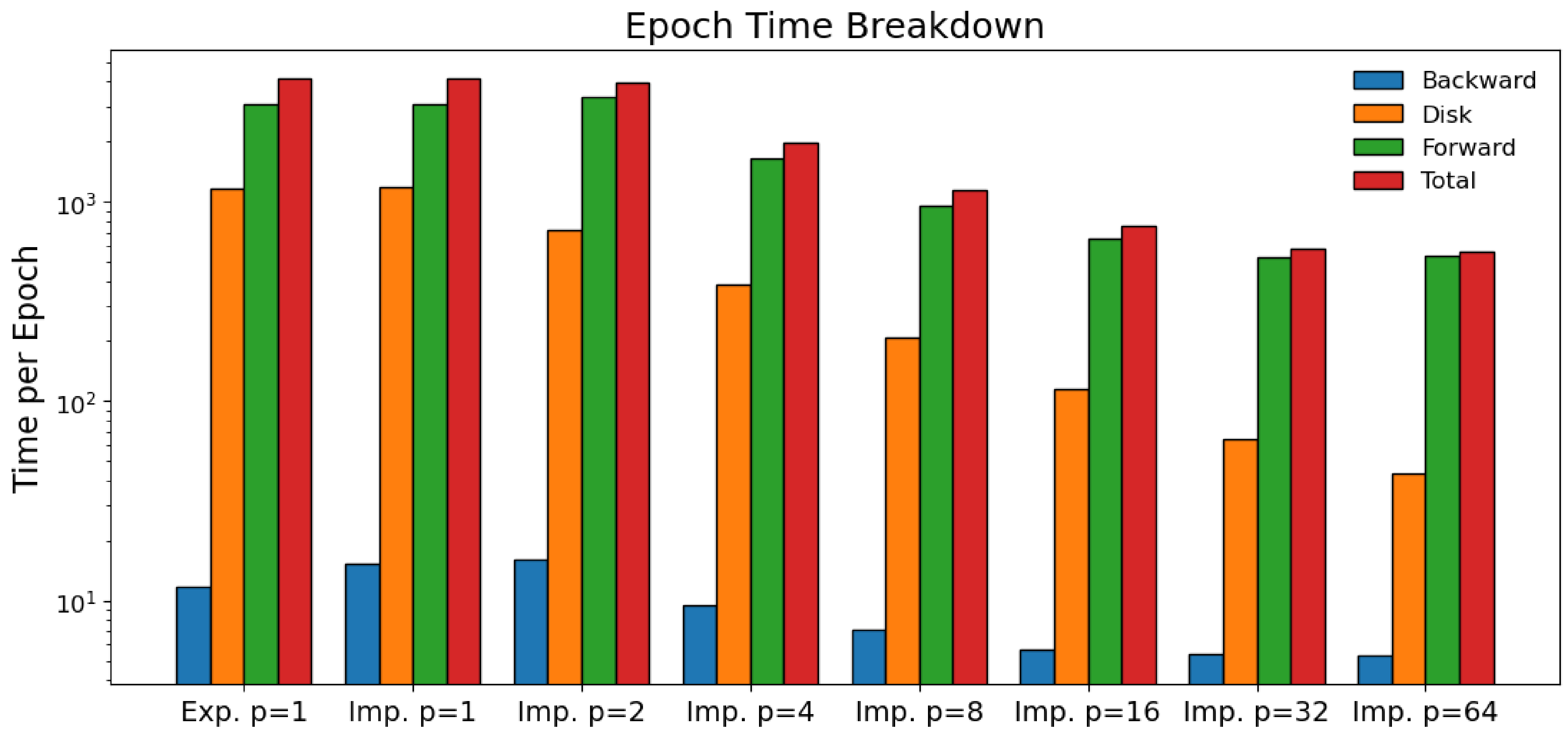


Parallel Training and Implicit vs. Classic GRU makes little difference in training loss and accuracy

Runtime: Timings Breakdown

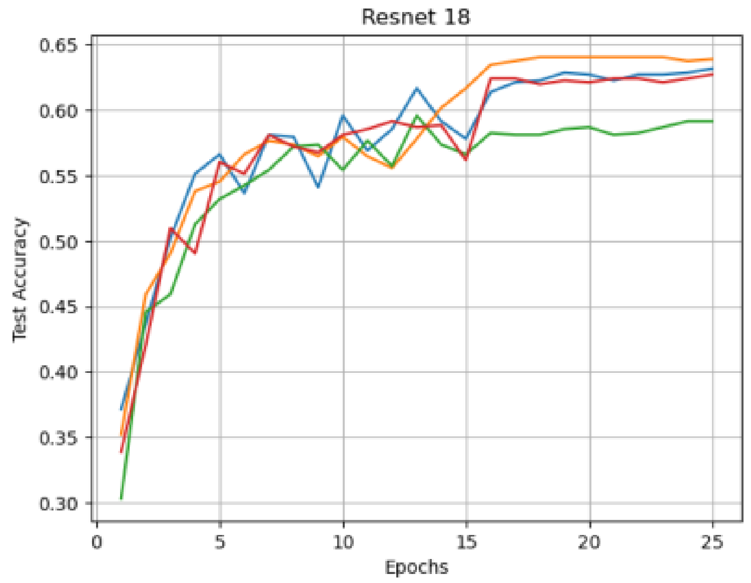


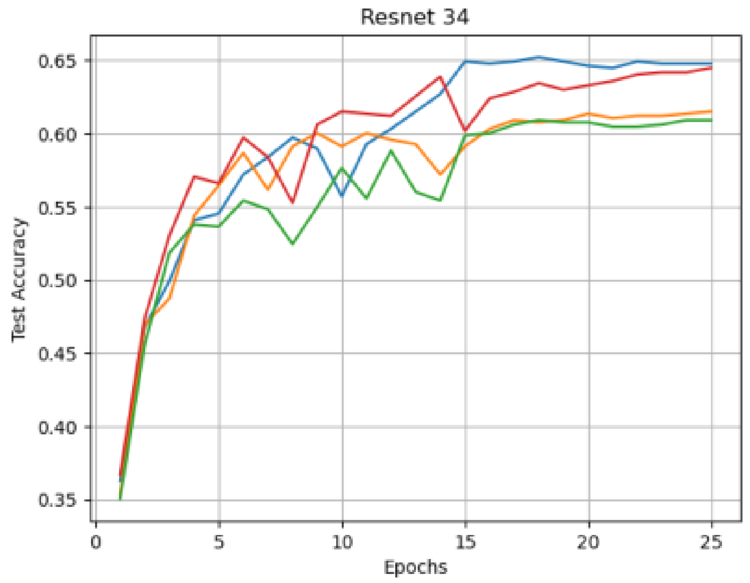
Runtime: Timings Breakdown



Speedups obtained in forward, backward, and disk time

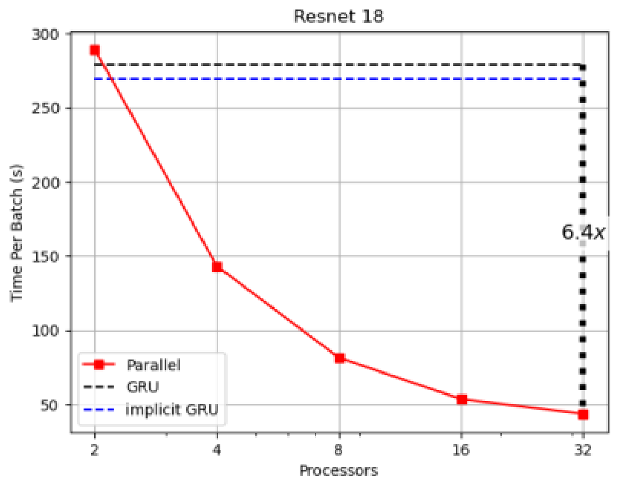
ResNet 18,34,50: Full dataset

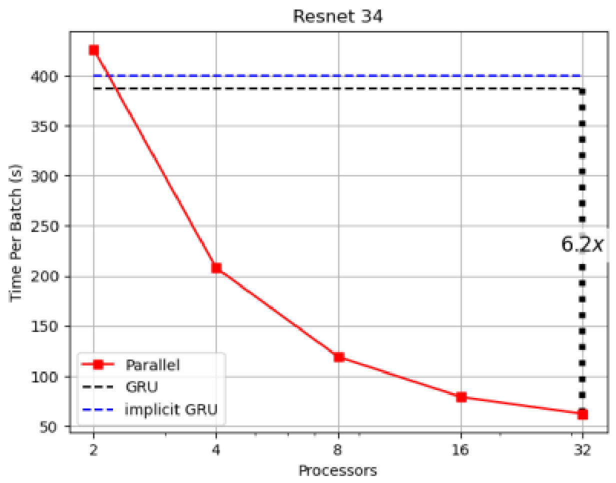


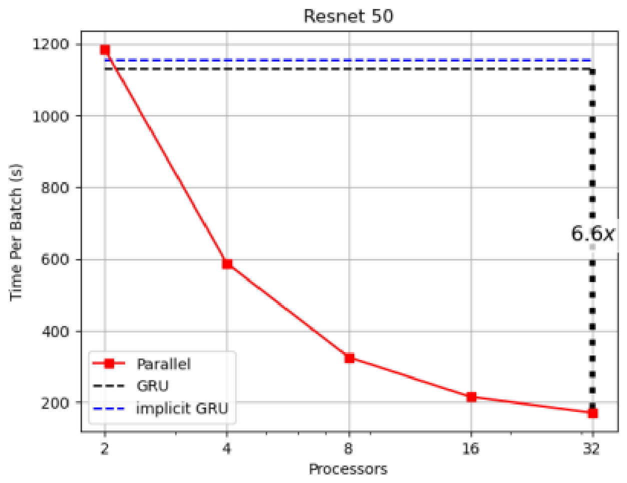


Accuracy on 32 MPI ranks

- Relatively insensitive to the starting condition
- Learning rate scheduling possible-not explored
- MGRIT iteration scheduling also possible







Good strong scaling speedups

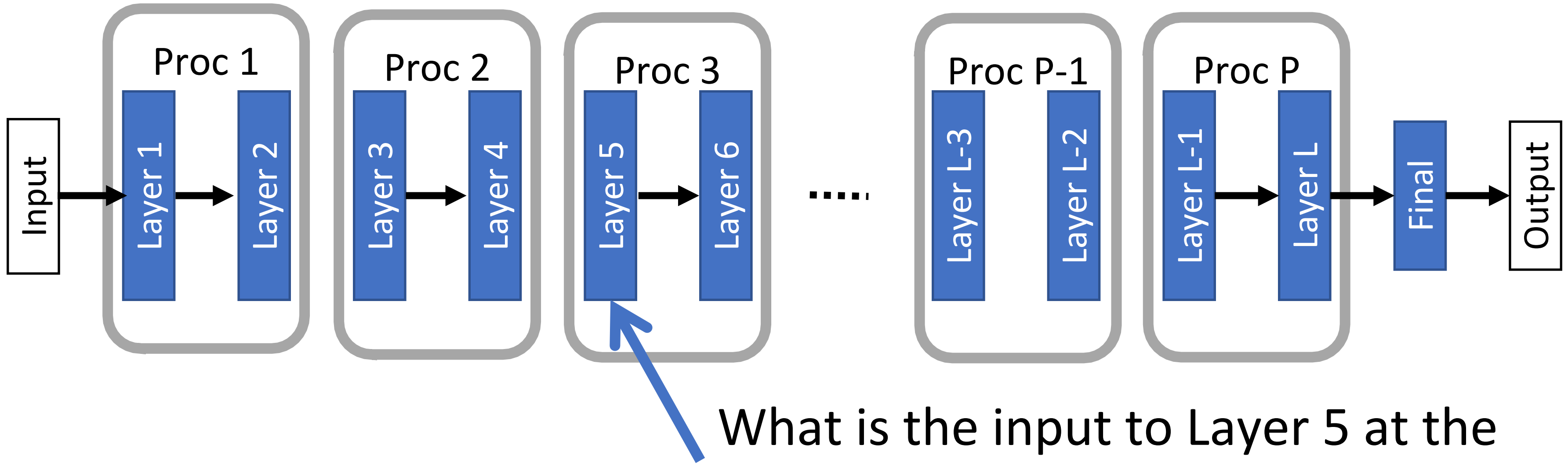
What about initial guesses?

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?

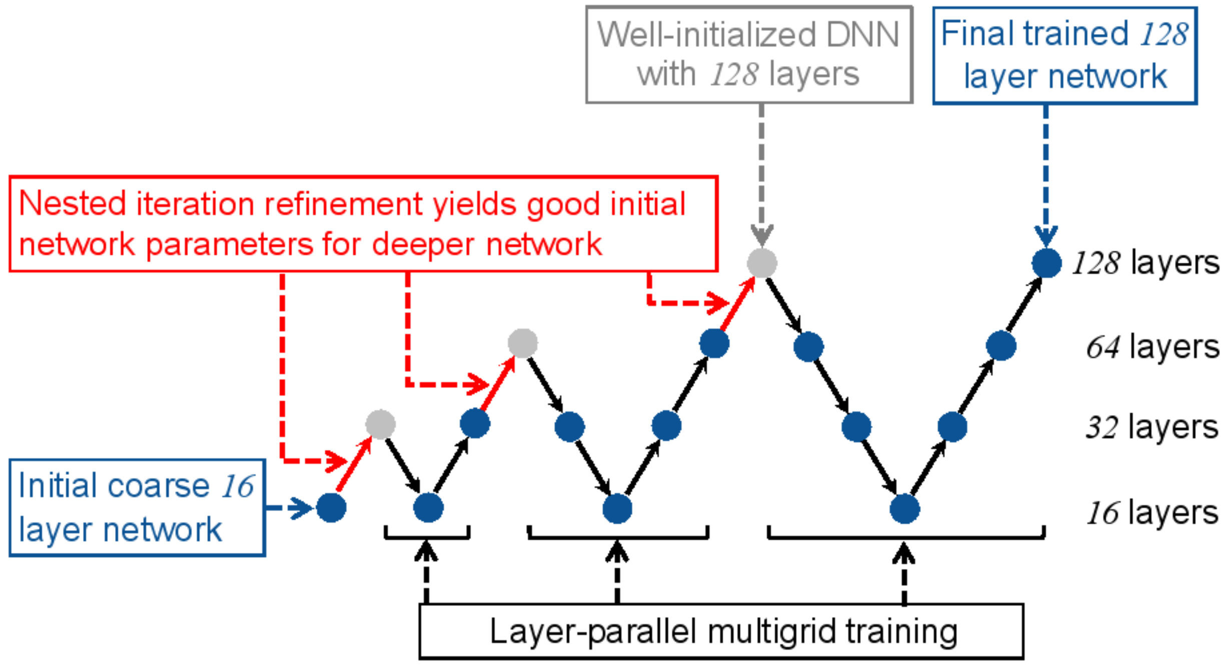


What is the input to Layer 5 at the beginning of the Layer-Parallel algorithm?

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


Well-initialized DNN with 128 layers
Final trained 128 layer network
Nested iteration refinement yields good initial network parameters for deeper network
128 layers
64 layers
32 layers
16 layers
Initial coarse 16 layer network
Layer-parallel multigrid training

Layer-Parallel Initialization: Nested Iteration

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

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

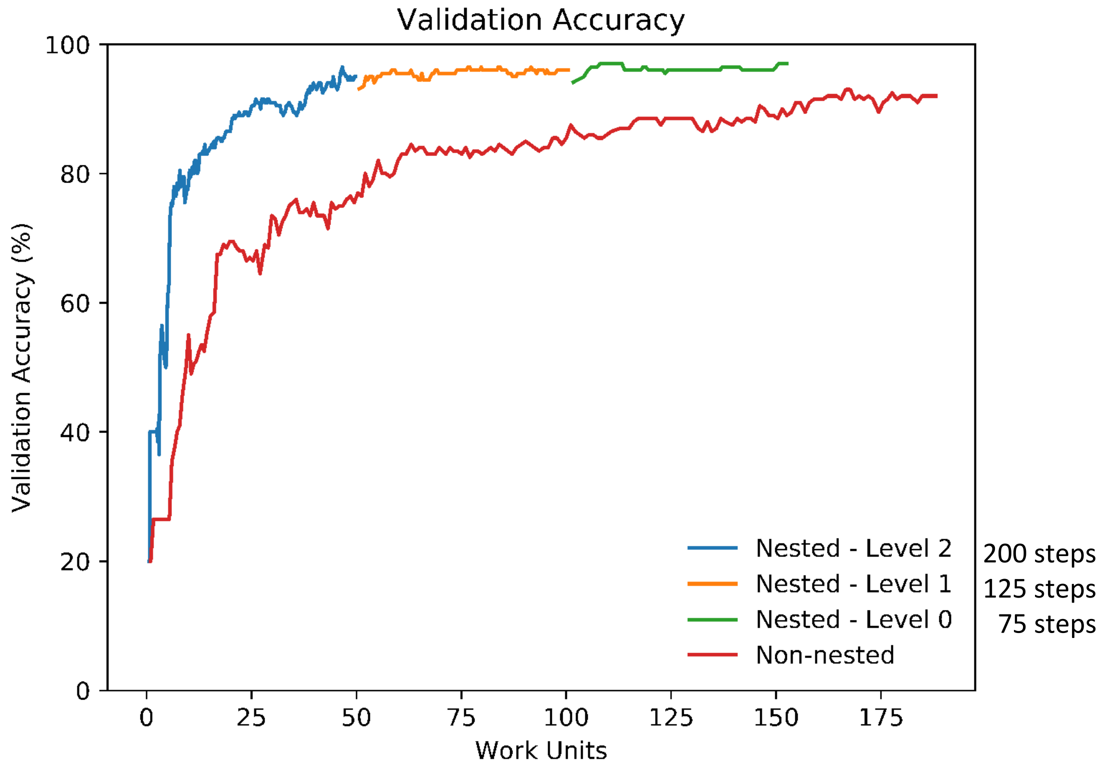
Initialization on the coarse level:

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

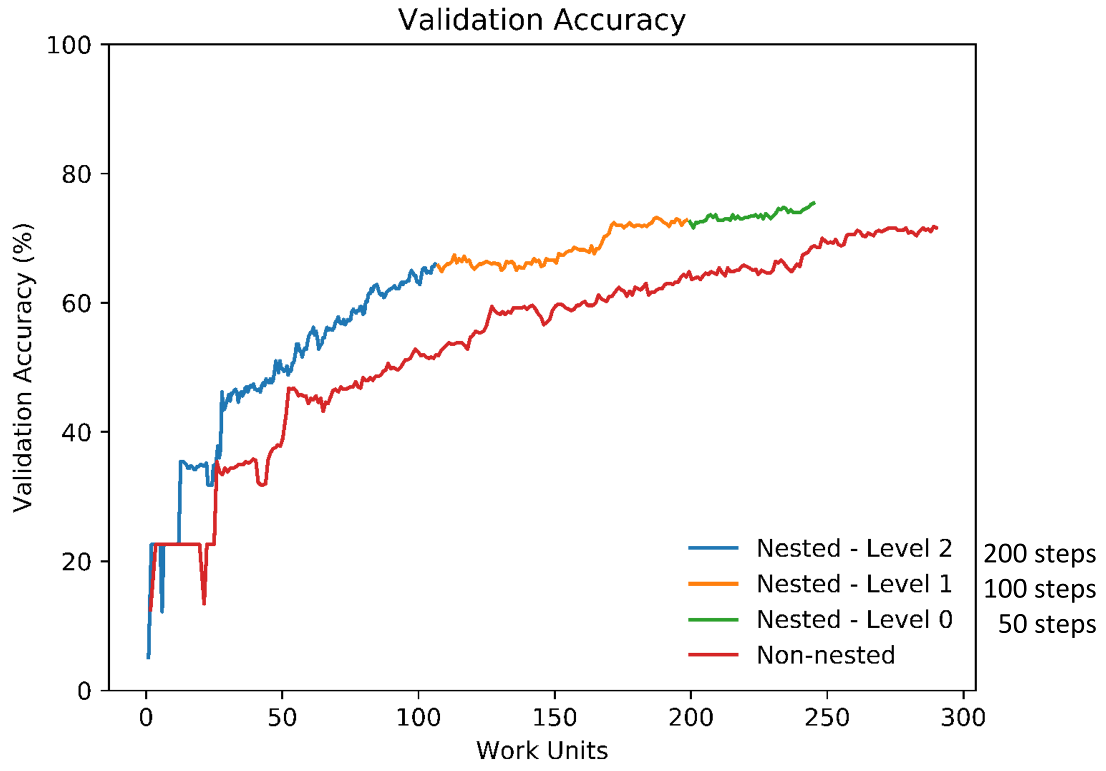
Nested Iteration: Indian Pines and Peaks

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

Peaks



Indian Pines



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)

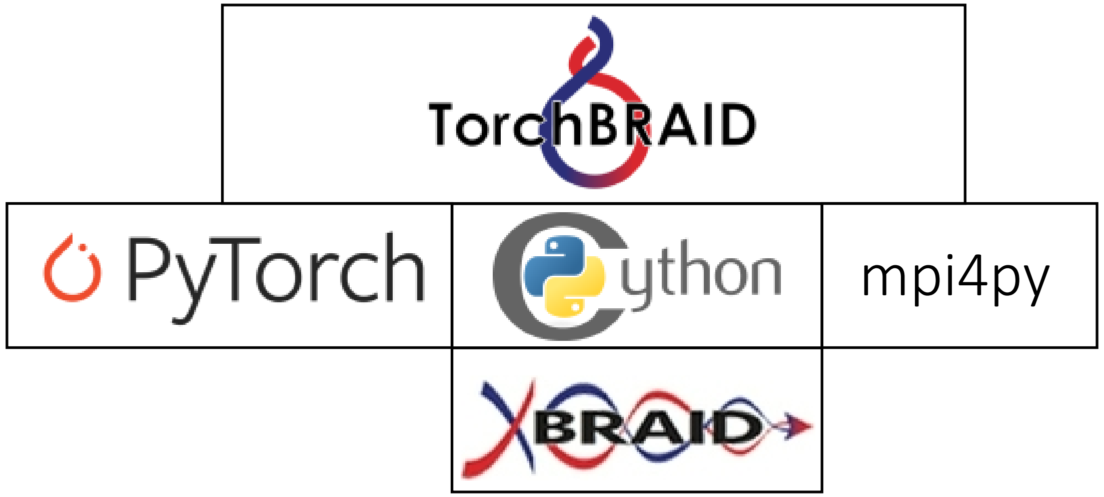
Torchbraid: Adding Layer-Parallel module to PyTorch

- Leverage more developers
- Uses automatic differentiation
- Support for ODENets and GRU-RNNs

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
```

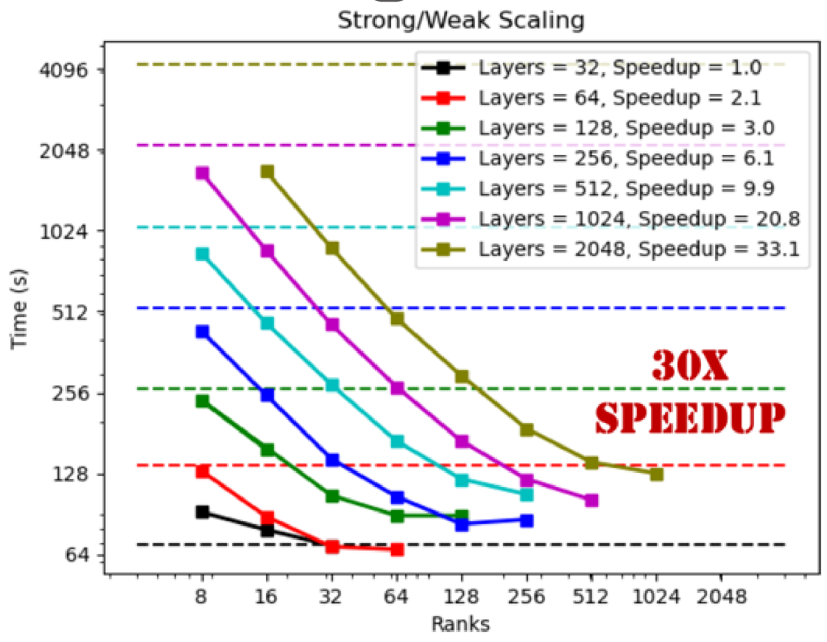
Layer-Parallel/Parallel-in-Time Training

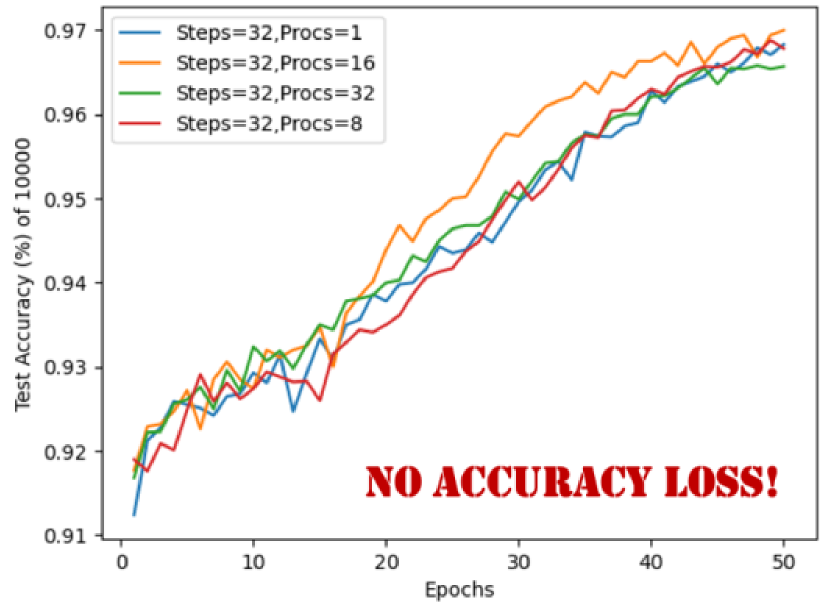
Running forward propagation:

- MNIST Training with ConvNets
- ODE Network with N Steps
- Each step contains 2 convolutional layers
- Dashed lines: pyTorch serial

TorchBRAID

<u>Take Home</u>: Torchbraid `LayerParallel` gives to speedups against PyTorch serial time





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
- Presented "TorchBraid" result: faster training

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!

References (An Incomplete List)

Layer-Parallel (multigrid modified SGD)

- Gunther, Stefanie, Lars Ruthotto, Jacob B. Schroder, Eric C. Cyr, and Nicolas R. Gauger. "Layer-parallel training of deep residual neural networks." *SIAM Journal on Mathematics of Data Science* 2, no. 1 (2020): 1-23.
- Moon, Gordon Euhyun, and Eric C. Cyr. "Parallel Training of GRU Networks with a Multi-Grid Solver for Long Sequences." ICLR, 2022 (*arXiv preprint arXiv:2203.04738*).
- Eric C Cyr, Stefanie Gu¨nther, and Jacob B Schroder. Multilevel initialization for layer-parallel deep neural network training. *arXiv preprint arXiv:1912.08974*, 2019.
- Kirby, Andrew, Siddharth Samsi, Michael Jones, Albert Reuther, Jeremy Kepner, and Vijay Gadepally. "Layer-parallel training with gpu concurrency of deep residual neural networks via nonlinear multigrid." In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1-7. IEEE, 2020.

Multigrid for Training

- Gaedke-Merzhäuser, Lisa, Alena Kopaničáková, and Rolf Krause. "Multilevel minimization for deep residual networks." *ESAIM: Proceedings and Surveys* 71 (2021): 131-144.
- von Planta, Cyrill, Alena Kopanicáková, and Rolf Krause. "Training of deep residual networks with stochastic MG/OPT." *arXiv preprint arXiv:2108.04052* (2021).

Other

- Ben-Nun, Hoefler. "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis." *ACM Computing Surveys (CSUR)* 52, 2019.
- Eliasof, Moshe, Jonathan Ephrath, Lars Ruthotto, and Eran Treister. "Multigrid-in-Channels neural network architectures." (2020).