Analyzing Neural Networks with Gradient Tracing

Brian DuSell

Application

Neural network analysis

- Networks are organized into logical components
  - Recurrent gates, differentiable data structures, etc.
- Let's try to identify components that most facilitate learning

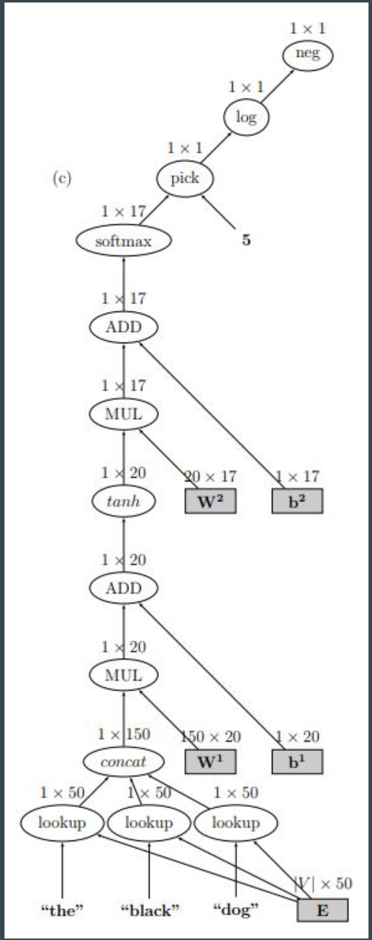
Kernel

"Gradient tracing"

- Finds the path in the computation graph through which the most gradient propagates
- Then finds components that the path intersects with
- Algorithmically very similar to "backpropagation"

Computation Graphs

- Any neural network can be expressed as a graph of mathematical operators
- Like an abstract syntax tree
- Vertices represent operators, constants, or parameters
- Edges are directed and represent assignments to function parameters
- Always a DAG
- “Components” are subgraphs of the computation graph

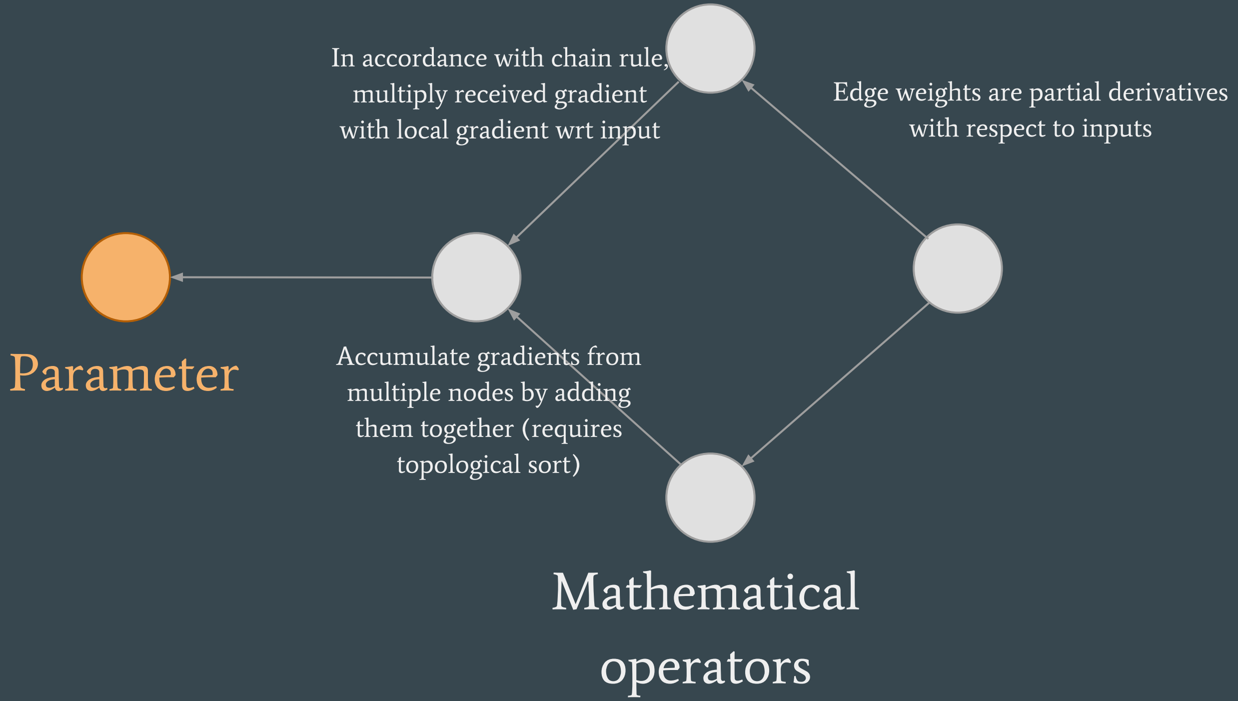


Backpropagation

- Computes the gradient of a loss function with respect to the network's parameters
- A necessary operation during training
- Can be expressed as a graph

Backpropagation as a Graph

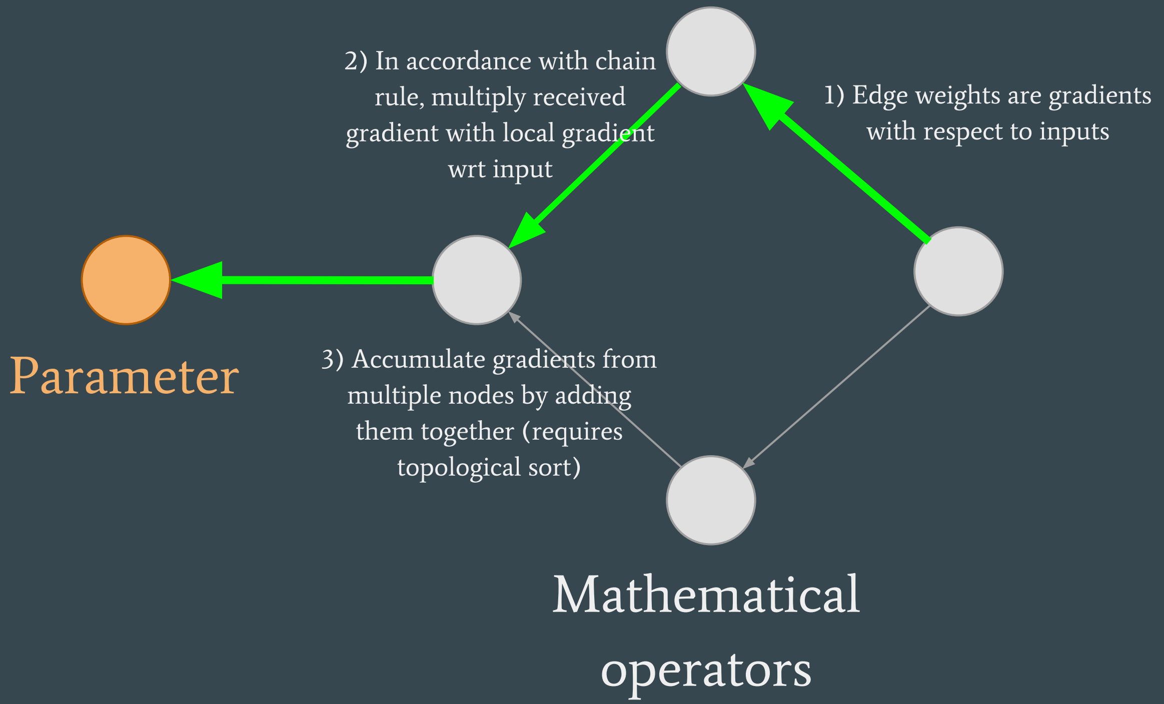
- Edge weights are partial derivatives
  - n.b. a “gradient” is a vector of partial derivatives
- Three simple rules
  - Multiply along paths
  - Sum incoming edges
  - Stop at parameters
- Can be seen as computing sum of all path weights leading into each vertex
  - Where path weight is the product of all edge weights along the path



Gradient Tracing as a Graph

Some paths are better than others!

Same procedure as backprop, different semiring (max/argmax of absolute value instead of sum)



Gradient Tracing

- How do we find the path with the highest weight?
- Answer: run backprop with a different semiring
- Instead of sum, take max of absolute value
  - Take argmax to preserve backpointers
- Analogous to the difference between the Forward Algorithm and the Viterbi Algorithm

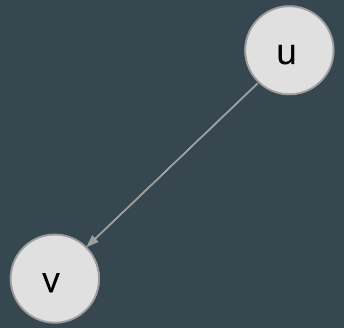
Backprop vs. Gradient Tracing

$$g_v = \frac{\partial L}{\partial U} = \begin{cases} \sum_{(u,v,k)\in E} w(u,v,k) g_u & v \neq \ell \\ 1 & v = \ell \end{cases}$$

Backpropagation (computing total incoming gradient)

$$g'_v = \begin{cases} \max\limits_{(u,v,k)\in E} |w(u,v,k) g'_u| & v \neq \ell \\ 1 & v = \ell \end{cases}$$

Gradient tracing (identifying path with biggest gradient)



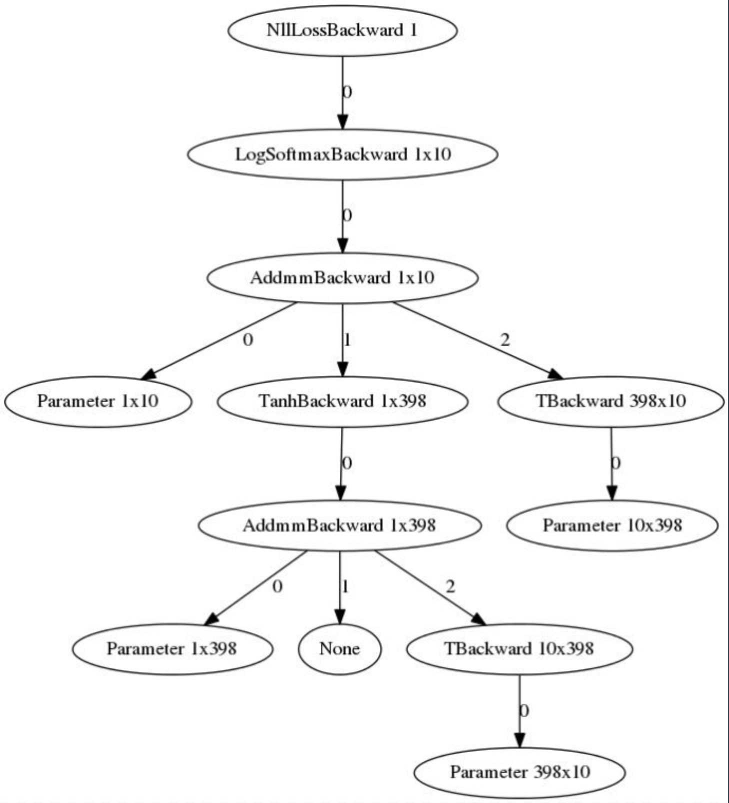
Complexity of Backprop and Gradient Tracing

- Every edge and vertex in the computation graph $G = (V, E)$ is visited a fixed number of times
  - Edge weights are summed or max-ed together
- Time complexity for both algorithms is linear: $O(|V| + |E|)$

Enhancement

- What I’ve described so far assumes that all inputs and outputs among operators are scalars
- But actual neural network implementations use tensor-level operations for performance
    - Tensor = multi-dimensional array
- Can use GPUs to accelerate tensor-level operations significantly
    - Both CPUs and GPUs take better advantage of parallelism and locality when values are grouped into contiguous “tensors”
- Note that the size of the graph is not the number of vertices and edges in the tensor graph, but in the equivalent scalar graph

Example Tensor-level Computation Graph



Implementation Details

- Language: Python
- Library: PyTorch
- Experiments show results on a Python implementation of the closely-related backpropagation algorithm, since it does not require digging into PyTorch primitives
- Problem: Computing the gradient (or gradient trace) for some operations (especially matrix multiplication) requires knowing the values of the inputs
  - PyTorch does not provide access to these through the Python API
- Given a PyTorch representing a loss function, traverses the computation graph, then topologically sorts using Kahn's Algorithm
  - Must use iterative rather than recursive code to avoid hitting recursion limit

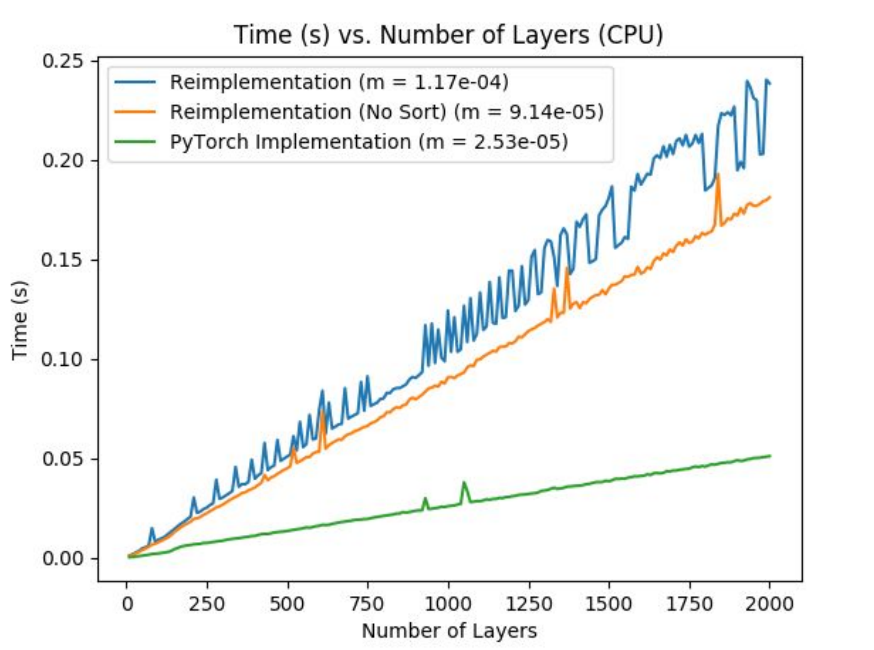
Scaling Results

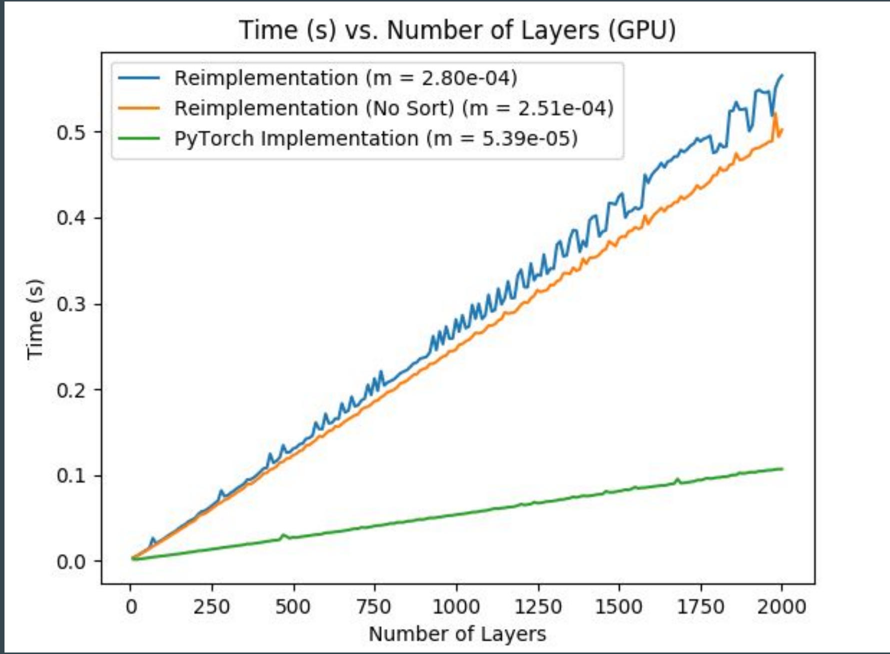
- Comparison of Python implementation of backprop (including and not including topological sort) and PyTorch's C++ implementation
  - CPU vs. GPU
  - vs. Number of Layers
  - vs. Batch Size
  - vs. Size of Weight Matrix
  - vs. Graph Size
- Gradient and gradient trace of matrix-vector multiplication
- CPU Model: Intel Core i7-4790, 3.60GHz, 8 cores
- GPU Model: NVIDIA Tesla K40c, 2880 CUDA cores

Time vs. Number of Layers

- Synthetically generated feed-forward neural network with N layers
- All layers have 20 hidden units, input and output are both 10 units
- Graph size is proportional to number of layers
- Scales linearly in number of layers
- Poor parallelization potential

Time vs. Number of Layers



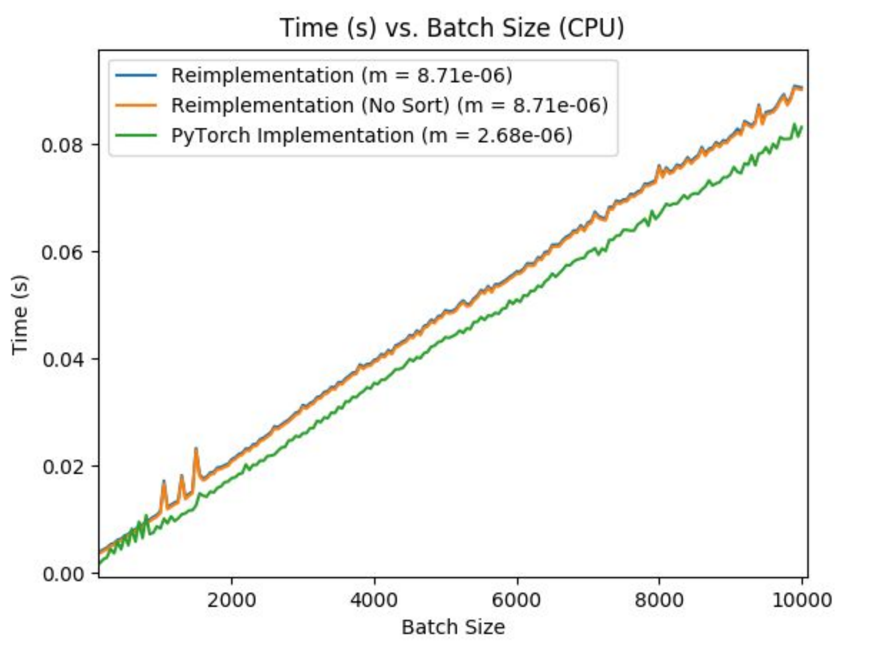


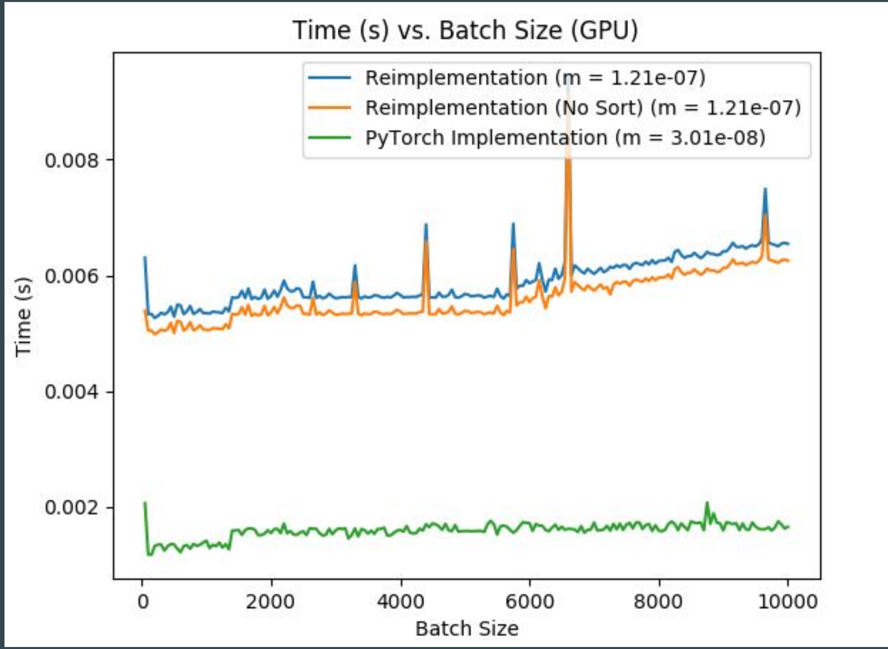
- GPU is twice as slow! -- serial layers are not parallelizable

Time vs. Batch Size

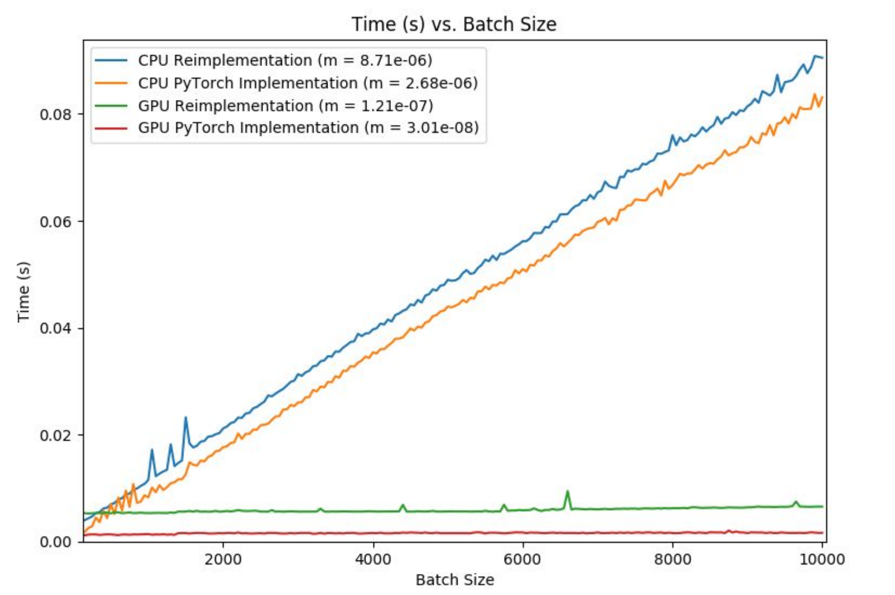
- Synthetically generated feed-forward network with varying batch size
- Higher batch size introduces greater opportunity for parallelism
- 20 layers, 20 hidden units each, input and output are 10 units
- Scales linearly in batch size

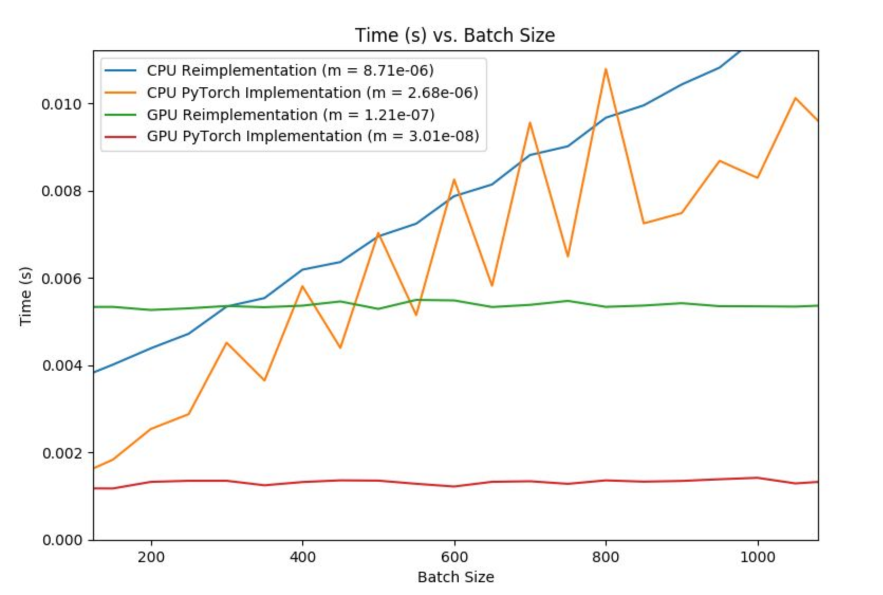
Time vs. Batch Size





Time vs. Batch Size

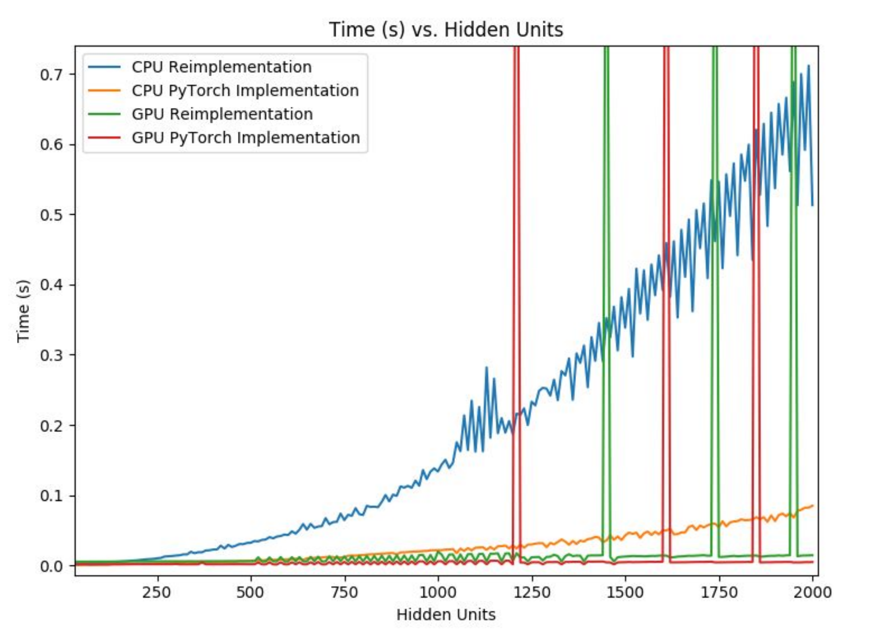


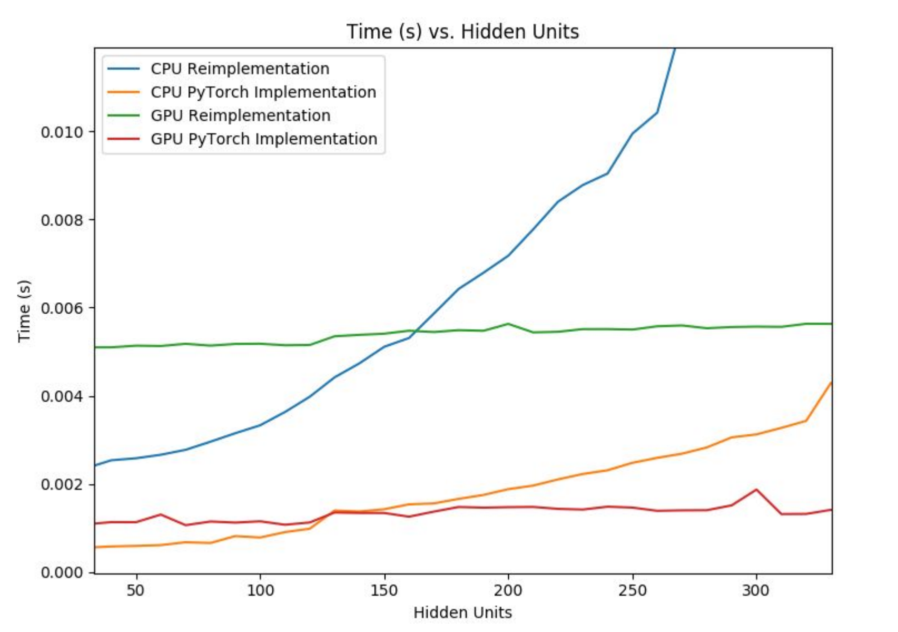


Time vs. Weight Matrix Size

- Network with 20 layers, 10 input and output units
- All layers have N hidden units
- Size of weight matrix is quadratic in N
- Time to do matrix-vector multiplication is quadratic in N
- Scales quadratically with N

Time vs. Weight Matrix Size



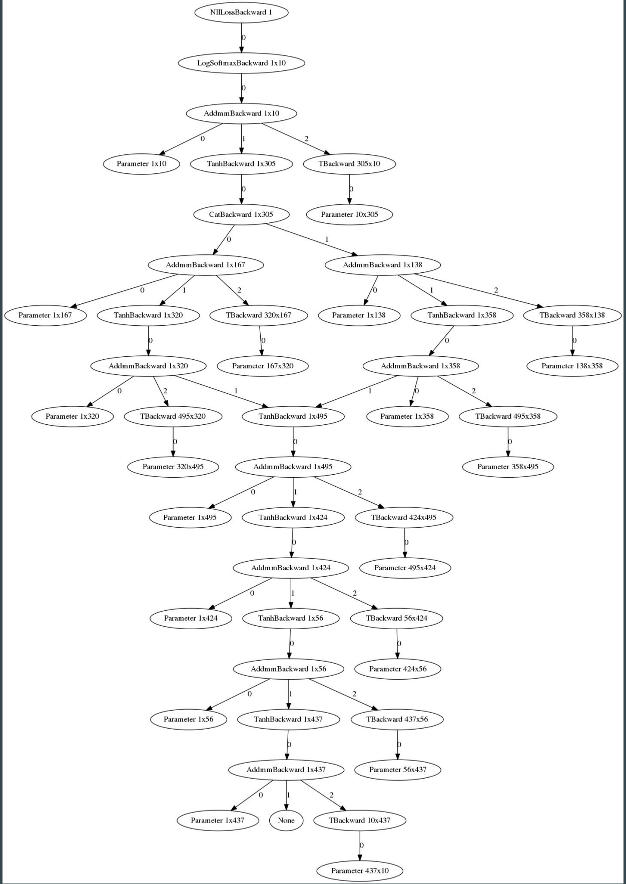


Time vs. Size of Random Graph

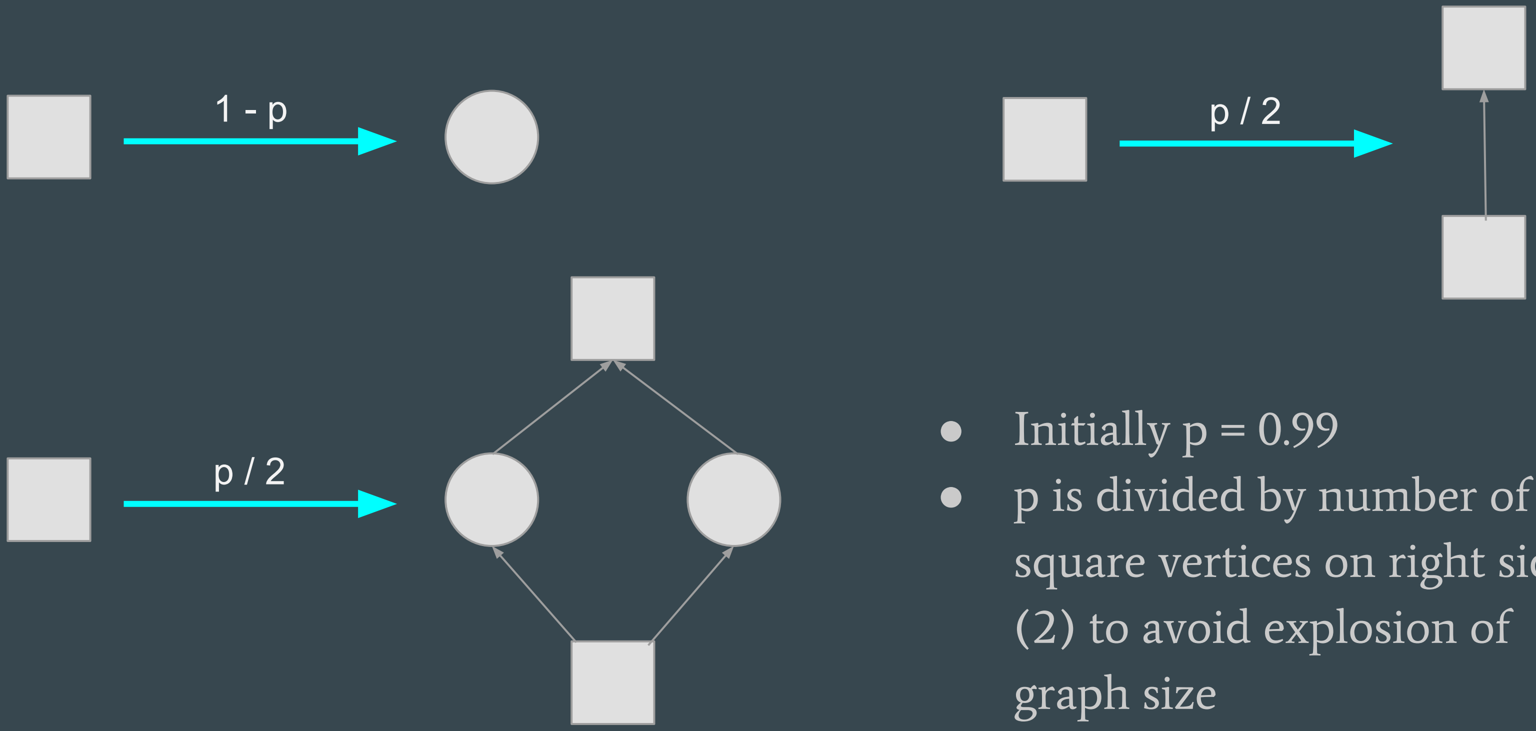
- Randomly generated feed-forward networks with diamond configurations
- Size of equivalent scalar graph is estimated from sizes and types of tensor operations

Random Graph Generation

- Recursively expand vertices using a “vertex replacement grammar”
- Consists entirely of weight matrices and tanh activation functions
- Sizes of tensor operations are sampled randomly from [10, 500]

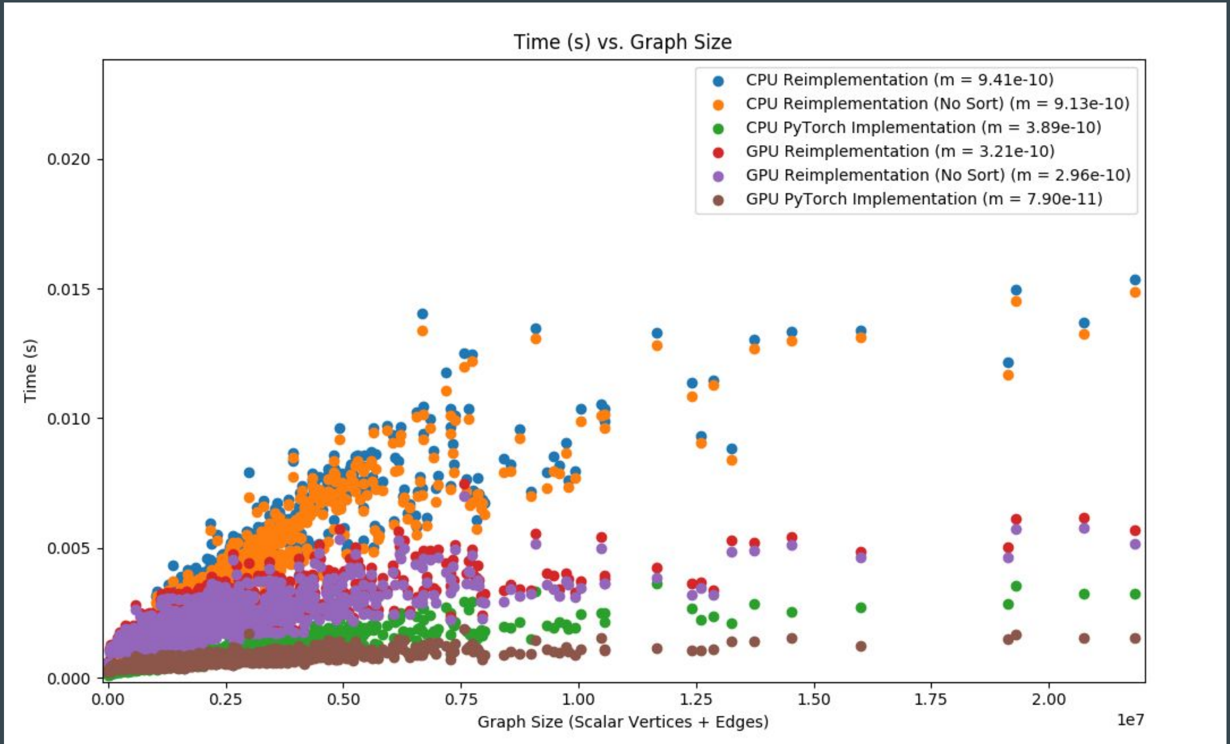


Random Graph Generation

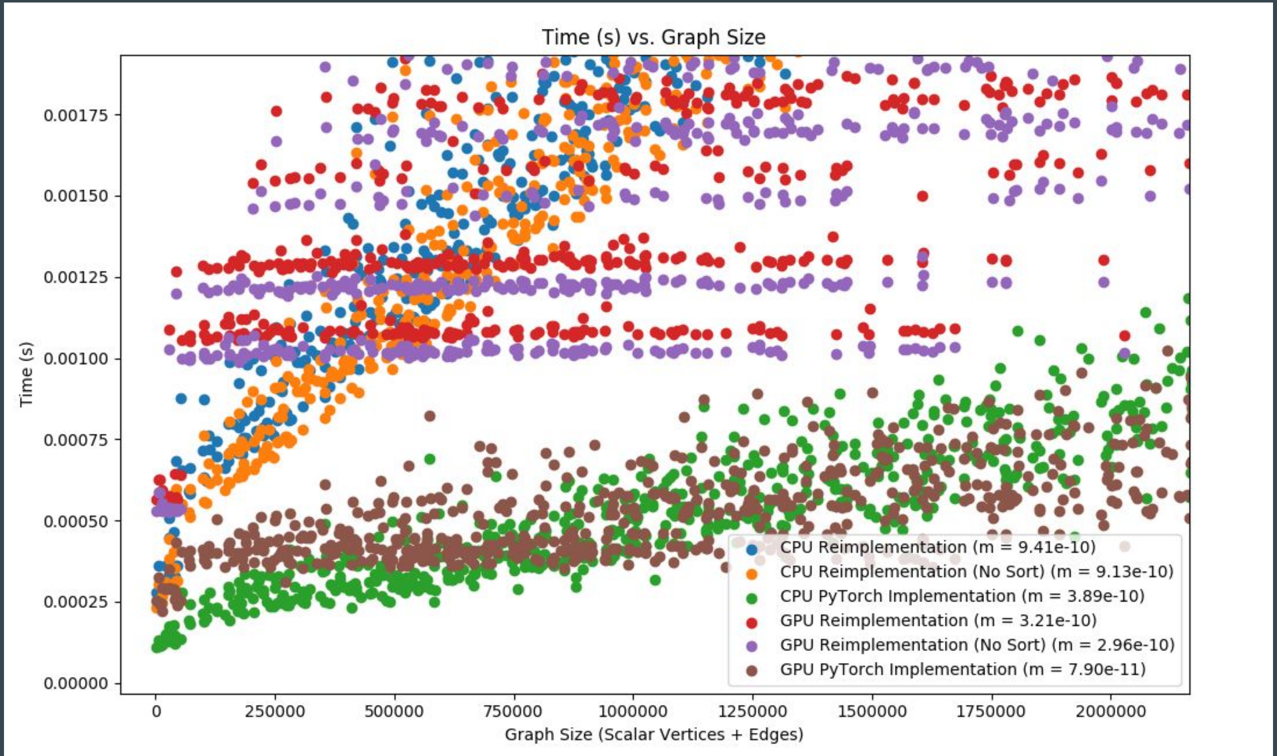


- Initially p = 0.99
- p is divided by number of square vertices on right side (2) to avoid explosion of graph size

Time vs. Size of Random Graph

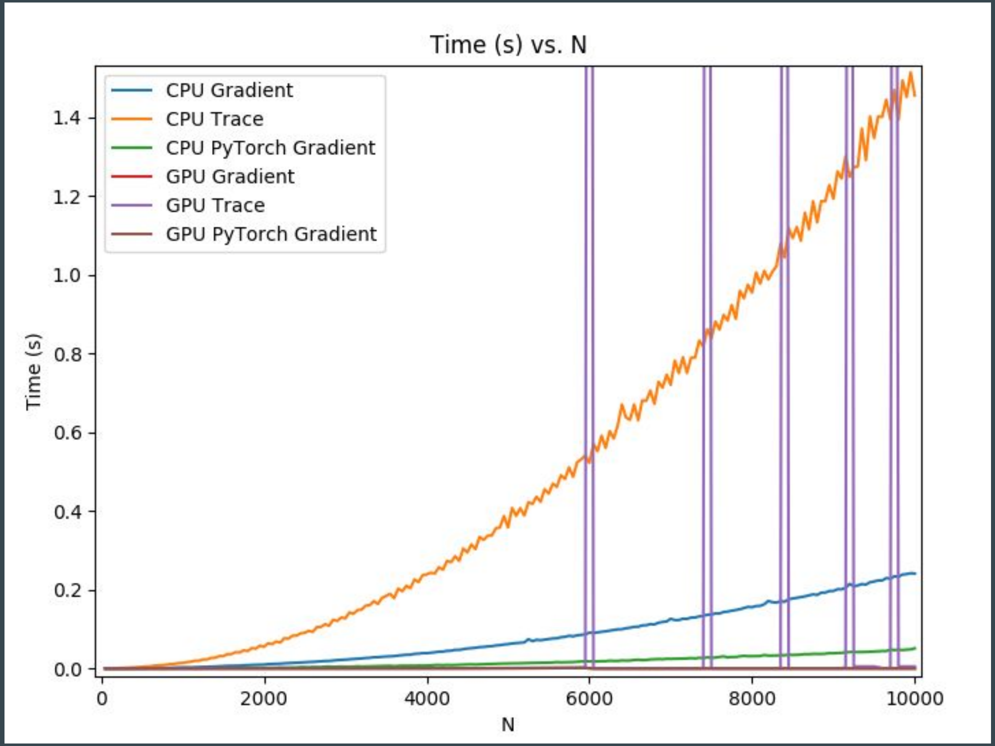


Time vs. Size of Random Graph

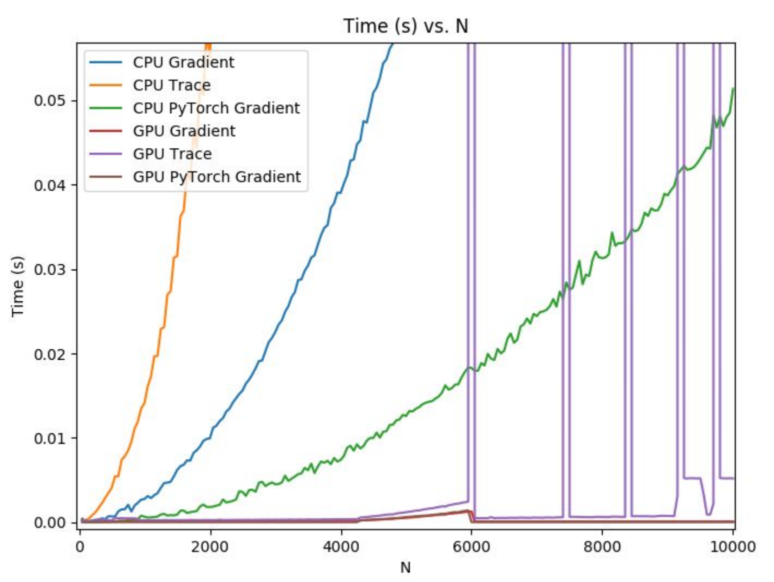


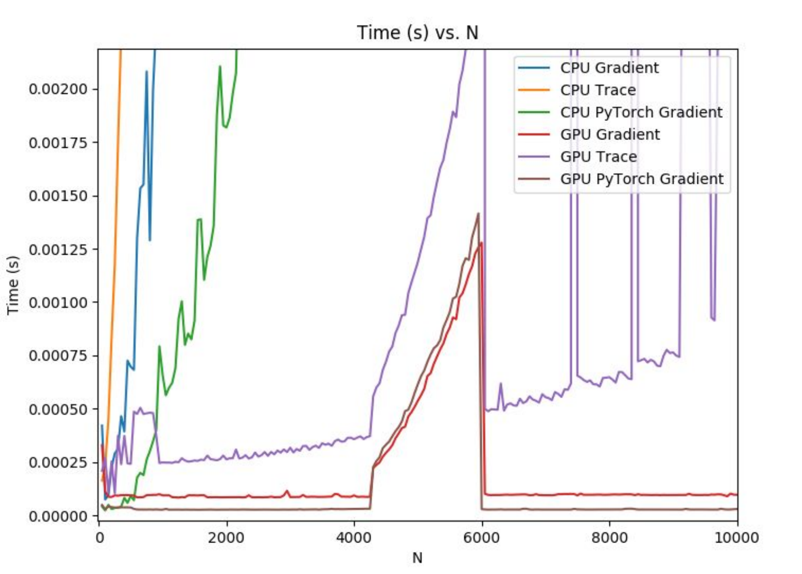
Matrix-vector multiplication gradient/trace

N is size of square matrix



Matrix-vector multiplication gradient/trace





What I Learned

- Inspecting the computation graph in PyTorch is... hard
- GPUs can still do pretty well on small graphs
- Gradient tracing is expensive compared to backpropagation
- In Python, write your graph traversals iteratively, without recursion