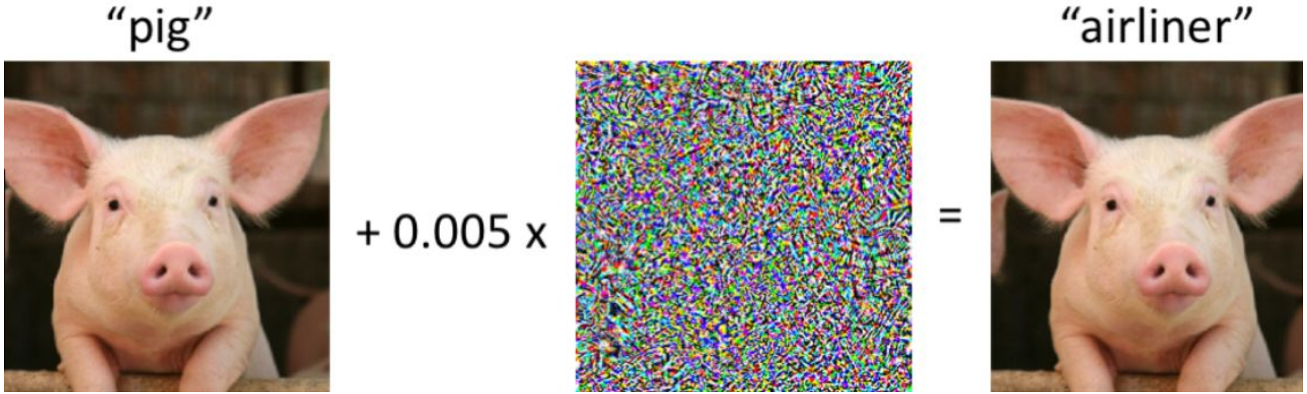
Parallel Training of Robust Neural Networks

Rajat Mittal, John Wang, Neehal Tumma, Sayak Maity

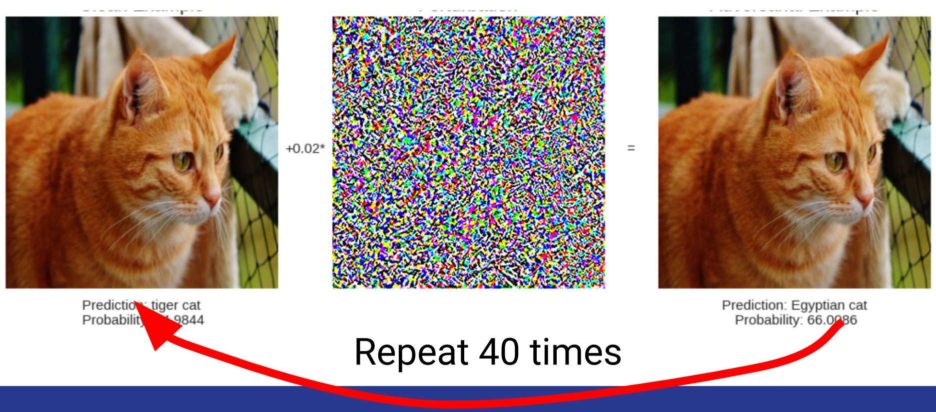
Background - CNNs



- susceptible to adversarial attacks
- exposing the model to adversaries during training can make it robust
- necessary for safety critical applications

How do we generate adversaries?

- We find adversaries using the gradient of the adversarial objective with respect to the input
- Iteratively apply gradient “ascent” to the pixels of an image to increase model’s error on the example



Fast Gradient Sign Method

$$x^{adv} = x - \varepsilon.sign(\nabla_x J(x, y_{target}))$$

$where$

x – Clean Input Image

x^{adv} – Adversarial Image

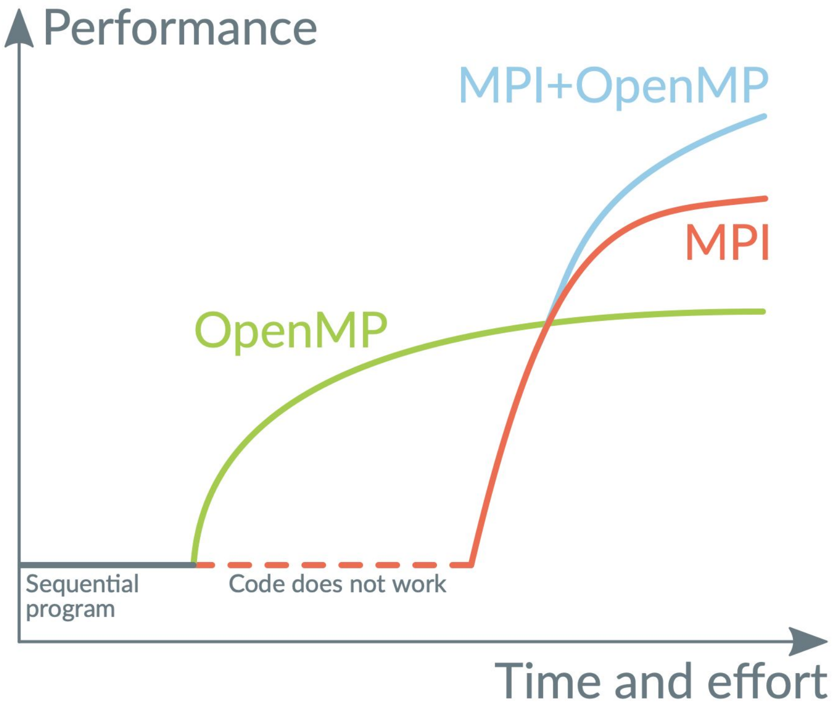
J – Loss Function

y_{true} – Target Label

ε – Tunable Parameter

Goals & Objectives

- Implement various parallelization strategies with OpenMP and MPI
- Demonstrate a proof-of-concept for parallel adversary generation
- Examine weak and strong scaling of our algorithm



Sequential Baseline

Adversarial Training

Loop through epochs

Loop through mini-batches

Generate adversaries (Bottleneck)

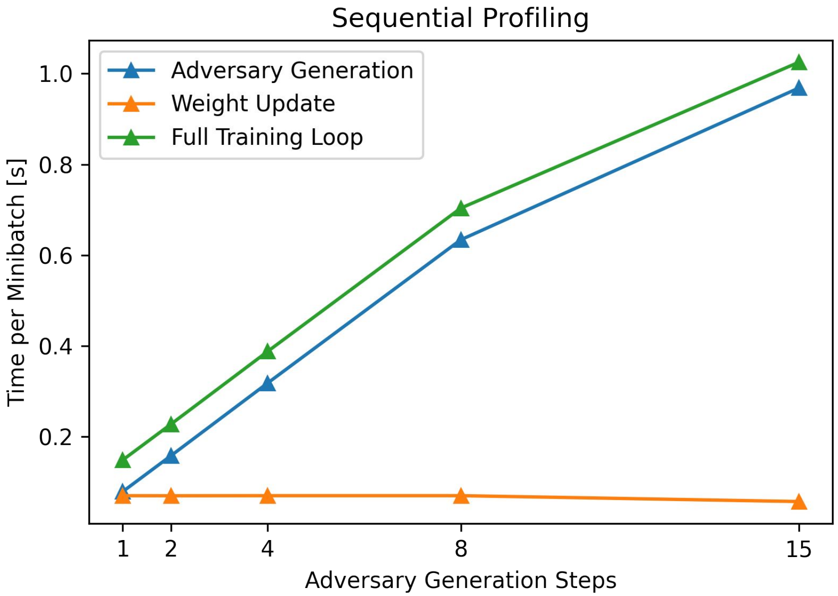
Calculate loss and backprop gradients on adversaries

Gradient Computation

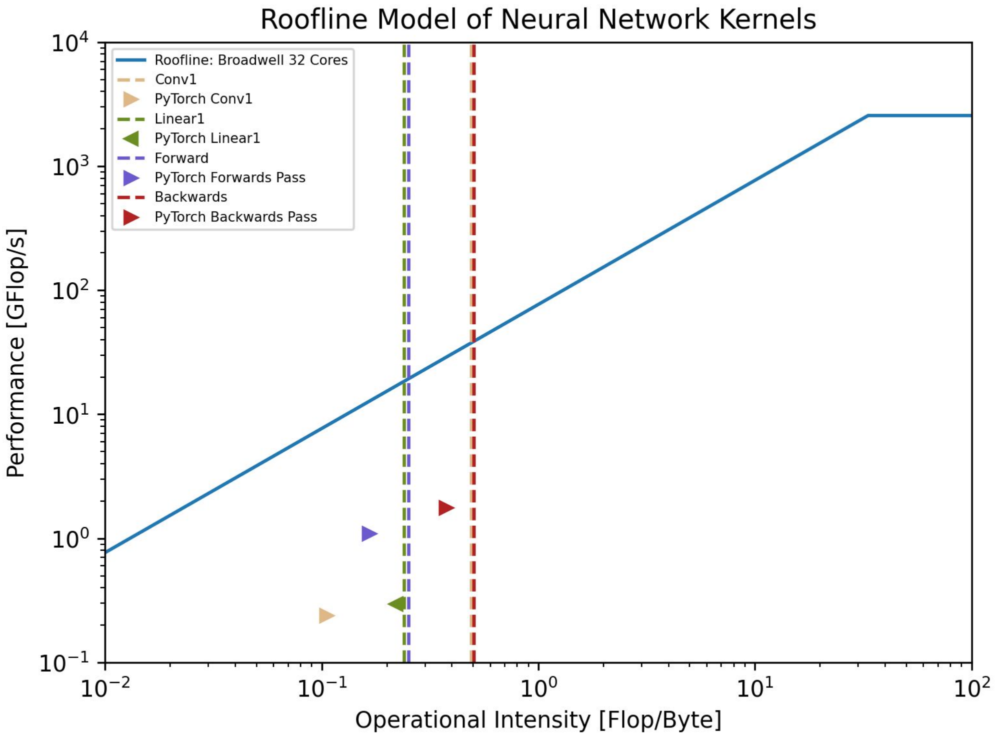
Repeat 15 times:

$$\frac{\partial L}{\partial I_v} = \frac{\partial L}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_{L-1}} \cdots \frac{\partial x_2}{\partial x_1} \cdot \frac{\partial x_1}{\partial I_v}$$

$$I_v^{t+1} = I_v^t + \alpha \cdot \frac{\partial L}{\partial I_v}$$

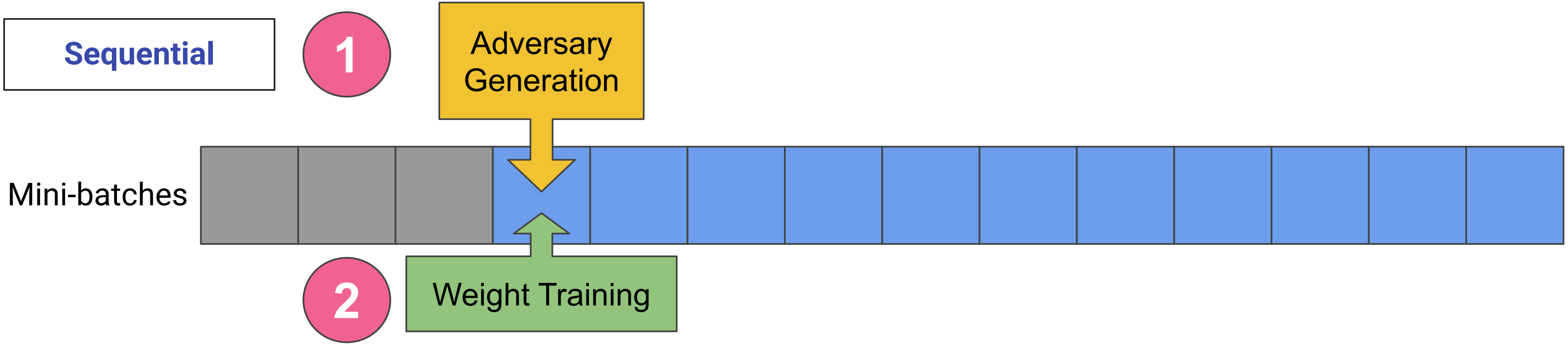


Roofline Analysis for Parallel Code



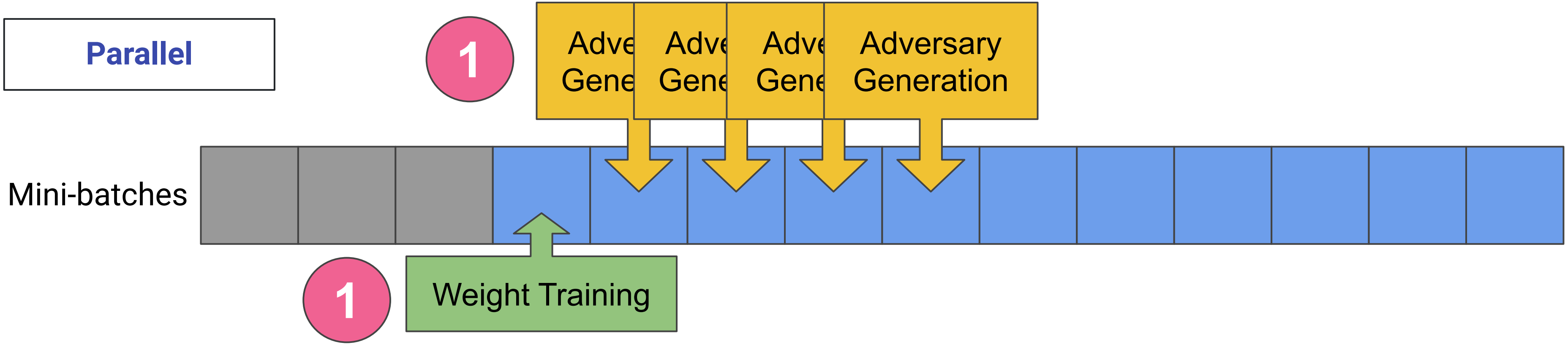
- We tested this on the Broadwell node with the Intel Xeon E5-2683v4 processor (32 cores)
- The compute kernels that dominate are the backwards pass and the convolutional layer
- For input size of neural networks of MNIST (28 x 28), have that both are approximately memory bound.

Parallelizing Weight Updates and Adversary Generation



- Simultaneously perform adversary generation and weight updates on MPI processes
- Every k epochs, weight training process sends new weights to adversary generating processes

Parallelizing Weight Updates and Adversary Generation



- Simultaneously perform adversary generation and weight updates on MPI processes
- Every k epochs, weight training process sends new weights to adversary generating processes

Splitting the Dataset

- Static minibatch assignment

Mini-batches	1	2	3	4	1	2	3	4	1	2	3	4	1	2

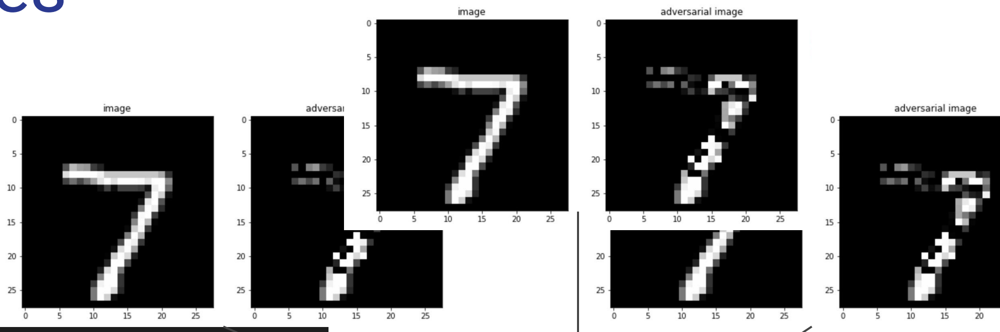
```
torch::data::samplers::DistributedRandomSampler sampler (train_dataset.size().value(),
    /*num_replicas=*/size-1, /*rank=*/rank-1);
auto train_loader =
    torch::data::make_data_loader(std::move(train_dataset),
    sampler,
    torch::data::DataLoaderOptions().batch_size(kTrainBatchSize)
    );

adversary(model, device, *train_loader);
```

One process is dedicated to model training, so the "world size" is size-1.

Communicating Adversaries

- Fairness imposed through loop

```
int batch_available = 0;
int count = 0;
while (count < kMaxProbes) {
    prev_source = (prev_source + 1) % (size - 1);
    int source = prev_source + 1;
    MPI_Iprobe(source, 101, MPI_COMM_WORLD, &batch_available, MPI_STATUS_IGNORE);
    if (batch_available) {
        MPI_Iprobe(source, 100, MPI_COMM_WORLD, &batch_available, MPI_STATUS_IGNORE);
        if (batch_available) {
            MPI_Recv(adversaries.data_ptr(), adversaries.numel(), MPI_FLO
                source, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv(targets.data_ptr(), targets.numel(), MPI_LONG,
                source, 101, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            return true;
        }
    }
    count++;
}
```



Weight Training Process

Cannot use MPI_Waitsome since 2 pieces of data are being sent!

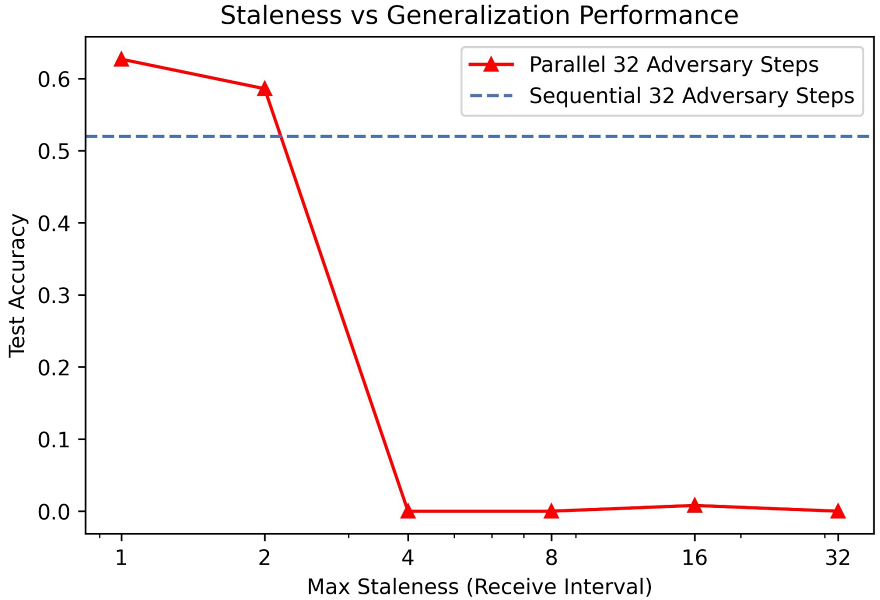
Sending Model Weights

Tradeoff: Do we want better adversaries or lower communication overhead?

```
torch::Tensor copied_input = input.clone();
for (int i = 0; i < steps; ++i) {
    auto grad = get_input_grad(model, copied_input, ...et);
    copied_input += step_sz * grad ;
    copied_input = torch::clamp(copied_input, vmin, vmax);
    if (requests && i % kCheckReceiveInterval == 0) {
        requests = check_receive_finished(model, requests);
    }
}
```

Sending Model Weights

Tradeoff: Do we want better adversaries or lower communication overhead?




```
torch::Tensor copied_input = input.clone();
for (int i = 0; i < steps; ++i) {
    auto grad = get_input_grad(model, copied_input, et);
    copied_input += step_sz * grad ;
    copied_input = torch::clamp(copied_input, vmin, vmax);
    if (requests && i % kCheckReceiveInterval == 0) {
        requests = check_receive_finished(model, requests);
    }
}
```

Ignoring Old Sends

```
MPI_Request* check_receive_finished(
    Net &model,
    MPI_Request* requests
    ) {
    int received;
    MPI_Testall(model.parameters().size(), requests, &received, MPI_STAT[illegible]NORE);
    if (received == 0) {
        return requests;
    }

    int message_available;
    do {
        MPI_Iprobe(0, model.parameters().size()-1, MPI_COMM_WORLD, &message_available, MPI_STATUS_IGNORE);
        if (message_available) {
            receive_model_weights(model, true);
        }
    } while (message_available);
    return receive_model_weights(model, false);
}
```

Tradeoff: We want the latest copies of the weights, so we have to "receive and ignore" all previous models sent.

Attempt to use OpenMP: Backpropagation for Adversary Generation is Inefficient

$$\frac{\partial L}{\partial W_L} = \frac{\partial L}{\partial x_L} \cdot \frac{\partial x_L}{\partial W_L}$$

$$\frac{\partial L}{\partial W_{L-1}} = \frac{\partial L}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_{L-1}} \cdot \frac{\partial x_{L-1}}{\partial W_{L-1}}$$

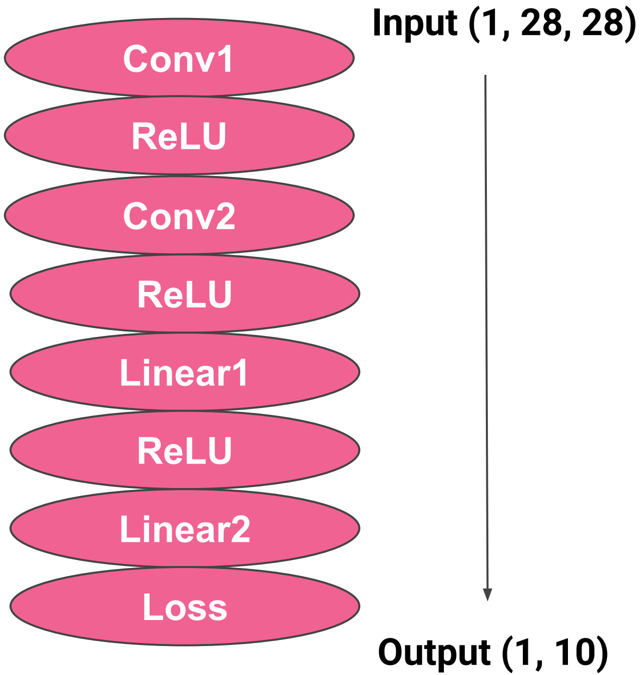
$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_{L-1}} \cdots \frac{\partial x_2}{\partial x_1} \cdot \frac{\partial x_1}{\partial W_1}$$

$$\boxed{\frac{\partial L}{\partial I_v} = \frac{\partial L}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_{L-1}} \cdots \frac{\partial x_2}{\partial x_1} \cdot \frac{\partial x_1}{\partial I_v}}$$

- Backpropagation avoids redundant computations of intermediate terms in the chain rule

- For adversary generation, we only require the product of a single chain of derivatives

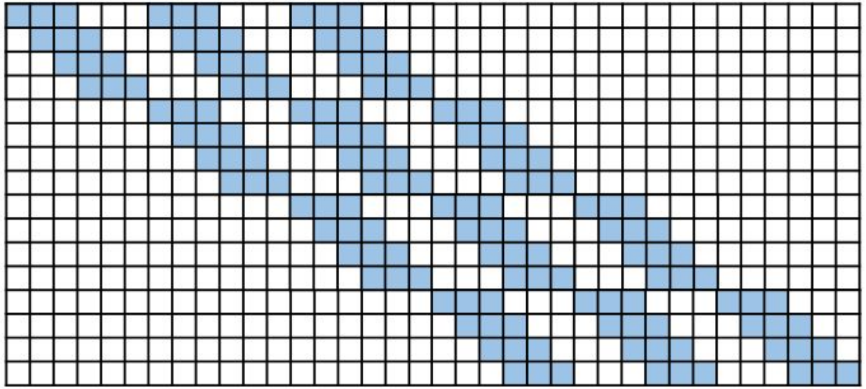
Manual Computation of Gradient



Gradient Computation

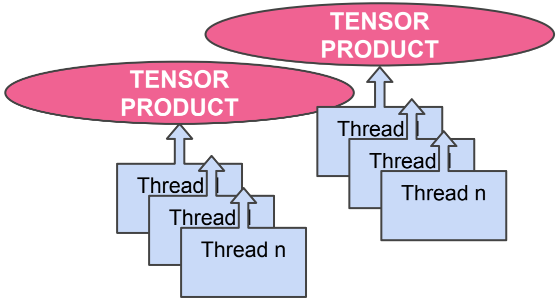
$$\text{loss} * \text{l2} * \text{l1\_relu} * \text{conv2\_relu} * \text{conv1\_relu}$$

- conv2_relu and conv1_relu operations are expensive since we need to compute a doubly blocked Toeplitz, likely not what PyTorch utilizes



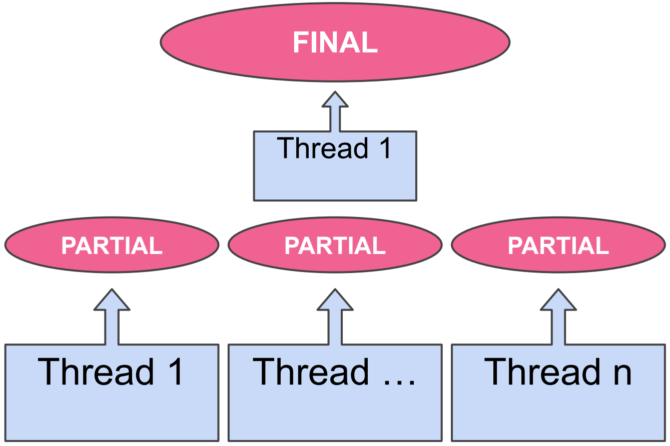
Parallel Algorithms for Tensor Multiplication

- One way to parallelize matrix multiplication is to use multithreading for each **individual tensor multiplication (intra-op parallelism).**



$$\frac{\partial L}{\partial I_v} = \underbrace{\frac{\partial L}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_{L-1}}}_{\text{Thread 1}} \cdot \underbrace{\frac{\partial x_{L-1}}{\partial x_{L-2}} \cdot \frac{\partial x_{L-2}}{\partial x_{L-3}}}_{\text{Thread 2}} \cdots \underbrace{\frac{\partial x_2}{\partial x_1} \cdot \frac{\partial x_1}{\partial I_v}}_{\text{Thread 3}}$$

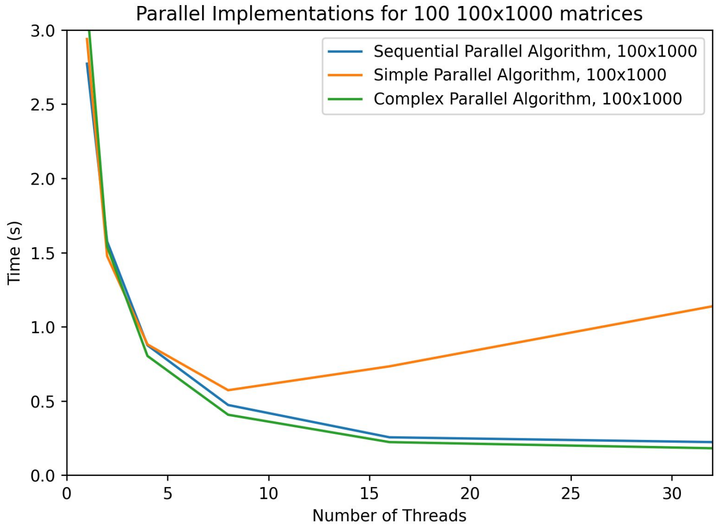
- We wish to explore instead using multithreading on **disjoint tensor multiplications (inter-op parallelism.**



$$\frac{\partial L}{\partial I_v} = \underbrace{\frac{\partial L}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_{L-1}}}_{\text{Thread 1}} \cdot \underbrace{\frac{\partial x_{L-1}}{\partial x_{L-2}} \cdot \frac{\partial x_{L-2}}{\partial x_{L-3}}}_{\text{Thread 2}} \cdots \underbrace{\frac{\partial x_2}{\partial x_1} \cdot \frac{\partial x_1}{\partial I_v}}_{\text{Thread 3}}$$

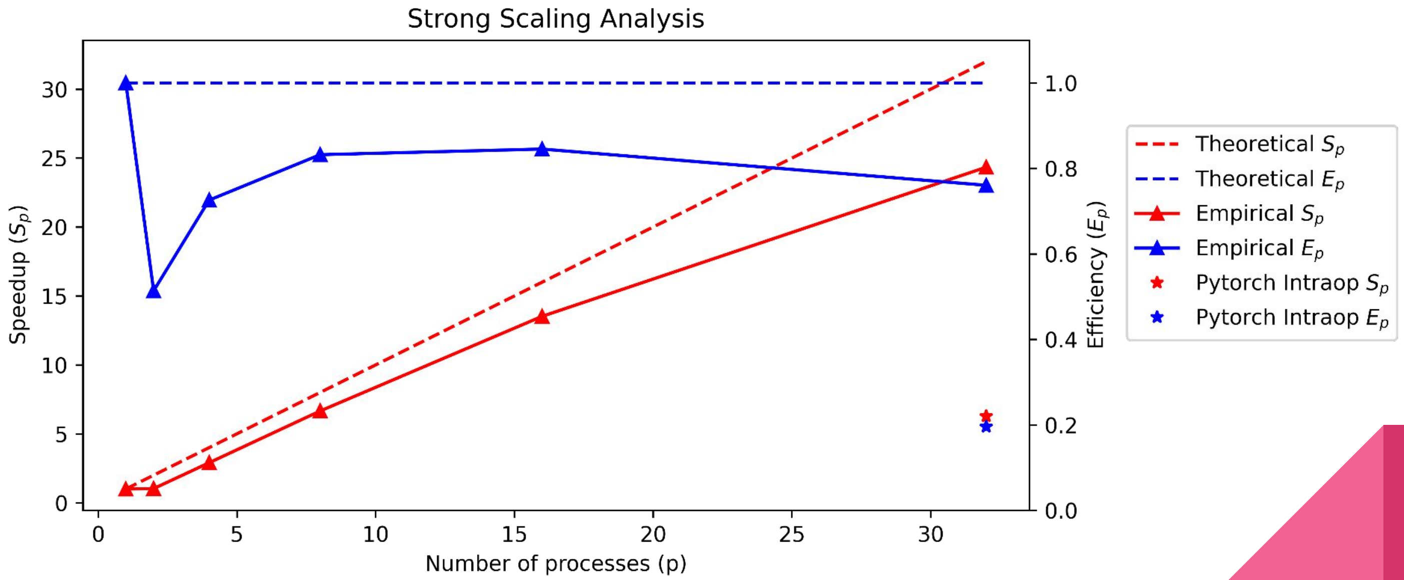
Why it didn't work

- Sequential for backprop is inherently better, due to compression when matrix multiplying
  - For instance, multiplying 1 x 50, 50 x 8000, 8000 x 1440, 1440 x 768.
  - Sequential compresses each matrix size to **1 x P matrices**
  - However, bottleneck occurs for parallel algorithm when multiplying 8000 x 1440, 1440 x 768
- We see results for **homogenous matrix sizes**



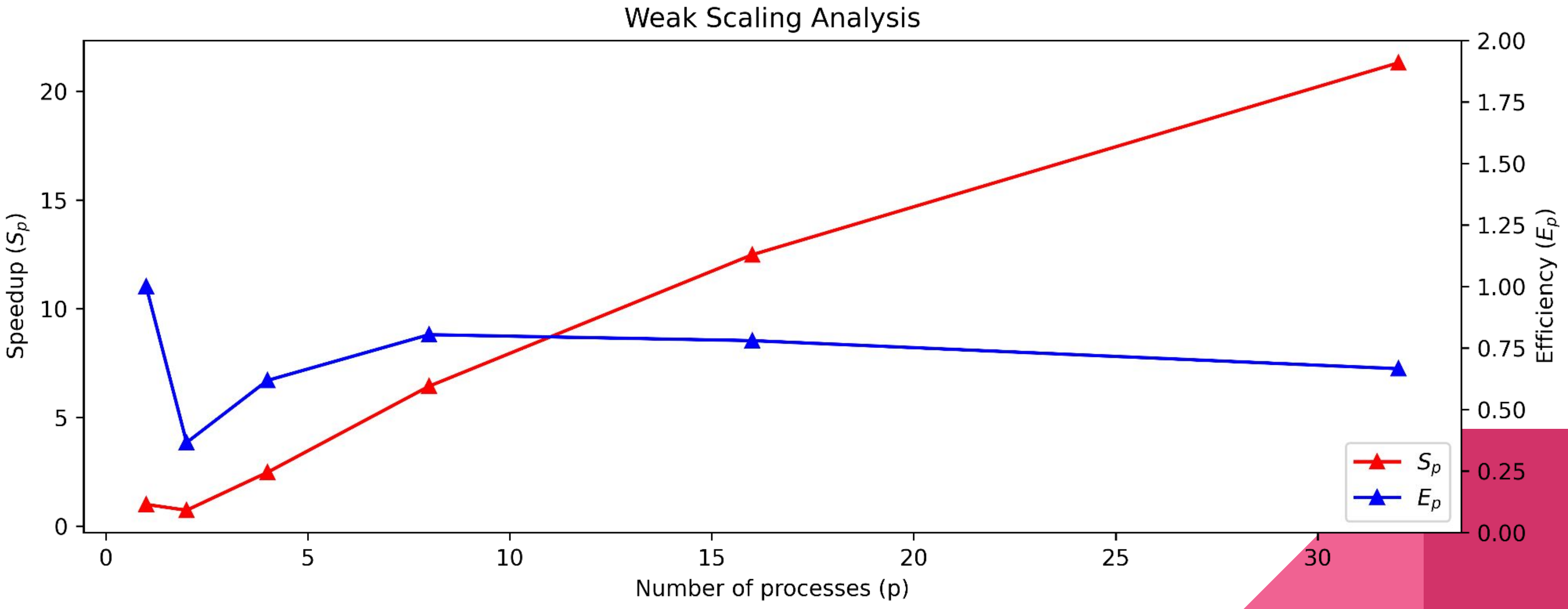
Sequential Baseline Strong Scaling Analysis

- We fix our problem size at 32 adversary steps so our serial fraction is 1/32



Sequential Baseline Weak Scaling Analysis

- We vary the problem size as a function of the number of adversary steps (which is also equal to the number of processes)



Takeaways & Future Work

- We were able to implement parallel adversary generation with little to no performance loss in the ML model
- We gained insight into the functionality of PyTorch under the hood and understood why it was so difficult to outperform it when using OpenMP
- Learning how the different methods of parallelization work helped us understand the performance results we got when developing the program
- Long-term goal is to create an open-source library that anyone can use for adversarial training