MVAPICH
MPI, PGAS and Hybrid MPI+PGAS Library

HiDL
High-Performance Deep Learning

Scaling Distributed DNN Training for Large Images

Elaboration Presentation

CSE 5914.01

Rayan Hamza, Nawras Alnaasan, Zhengqi Dong, and Arpan Jain

The Ohio State University

E-mail: hamza.23@osu.edu, alnaasan.1@osu.edu, dong.760@osu.edu, and jain.575@osu.edu

Outline

- Introduction
- Research Hypotheses
- Background Research
- Completed Work
  - Automatic splitting module of DNNs
    - Analytical Model
    - Network Representation and Splitting
  - CPU offloading of out-of-core models
- Future Work

Project Vision

- A user-friendly distributed DNN training framework for model- and hybrid- parallelism for vision models.

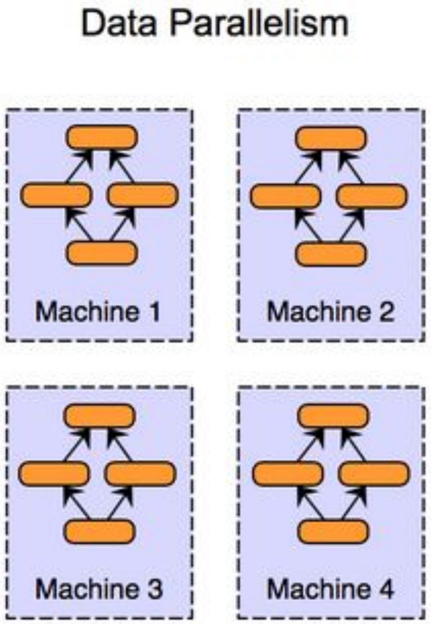
Problem Statement

- Design and implement a distributed DNN training framework in **PyTorch** to train **out-of-core DNNs** using an **automatic model splitting** module designed **to improve performance**.

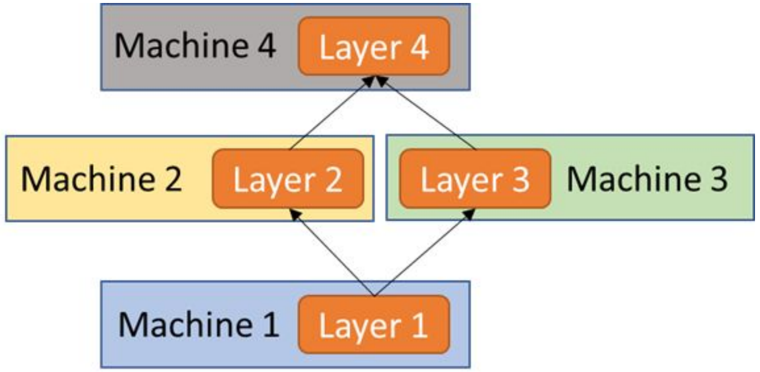
PyTorch

Motivation for Parallelism in Training

- **DNNs require a lot of memory, computation, and time**
  - Training large models serially could take over a week depending on the complexity
- **Solution – Data parallelism**
  - Use multiple GPUs to train data at the same time
  - These GPUs communicate using message passing interface (MPI)
  - Average out the gradient of each model on each GPU
- **New problem – some models can't fit on one GPU**
  - Models are becoming much more complicated
  - Pathology – 100k x 100k pixels in images



Src: https://xiandong79.github.io/Intro-Distributed-Deep-Learning



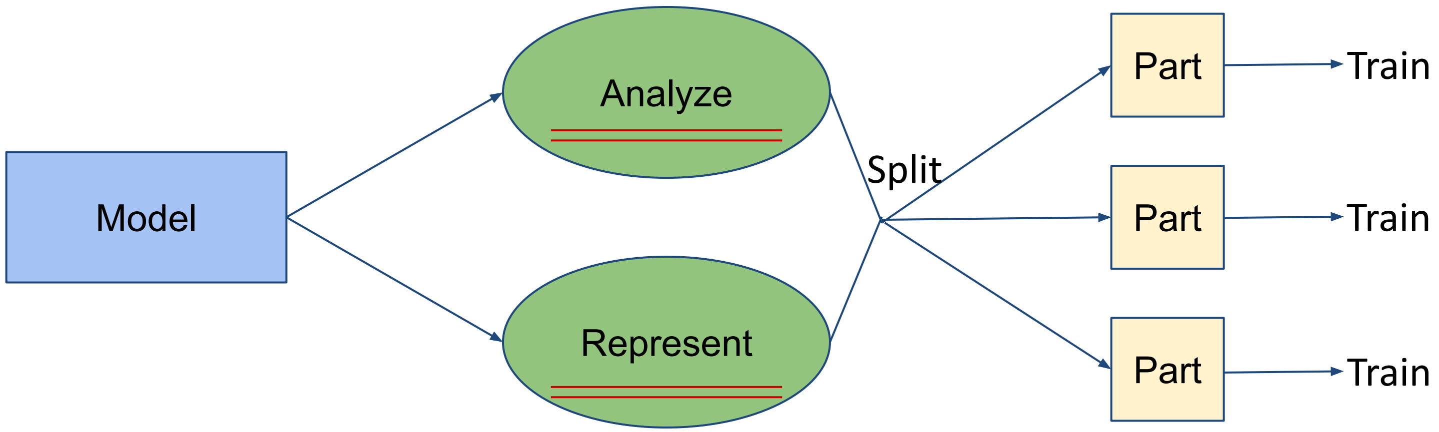
Model Parallelism (layer-level)

Research Hypotheses

- Use analytical models to estimate execution time for a model split to efficiently split DNNs across multiple GPUs
- Use PyTorch’s model and jit.trace API to understand data flow in DNNs written in PyTorch to implement user transparent model splitting
- Use CPU offloading mechanism to optimize GEMS-MASTER design

Automatic splitting module of DNNs - Approaches

- Analytical Model
  - Based on layer propagation time and parameters
  - Critical for deciding how to split the model
- Network representation and splitting
  - Represent model as a graph to find layer connections
  - Decide how to split the model
  - Segment the model and train over many machines



Automatic splitting module of DNNs - Background Research

- Analytical Model:

  Convolution has different time overheads depending on block size, conv. operation(e.g. adaptive tiling), and kernel info [1]

- Network Graph Representation:

  Existing tools for visualizing DNNs with PyTrorch: TensorBoard, Torchviz

  Inconvenient for the purposes of the project.

[1] van Werkhoven et. al. *Optimizing Convolution Operations on GPUs using Adaptive Tiling*, 16 September 2013.

Analytical Model - Target Layer Times

- Times for convolutional Layer
- nn.Conv2D(in_channels, out_channels, kernel_size, padding)
  - High maximums from first/last iterations of model - preparation
  - As expected, backprop is often twice as long as forward prop
  - Both layers have the same kernel size, but different in and out channels, which can affect the resulting times.

```
nn.Conv2d(1, 32, 3, 1)
```

```
Conv2d layer
-----------------------------------
### Forward ####
         min: 0.1886720061302185 ms
      median: 0.27238398790359497 ms
         max: 7.279615879058838 ms
        mean: 0.27238398790359497 ms
### Backward ####
         min: 0.8370879888534546 ms
      median: 1.9641599655151367 ms
         max: 12.137887954711914 ms
        mean: 2.029363780250288 ms
```

```
nn.Conv2d(32, 64, 3, 1)
```

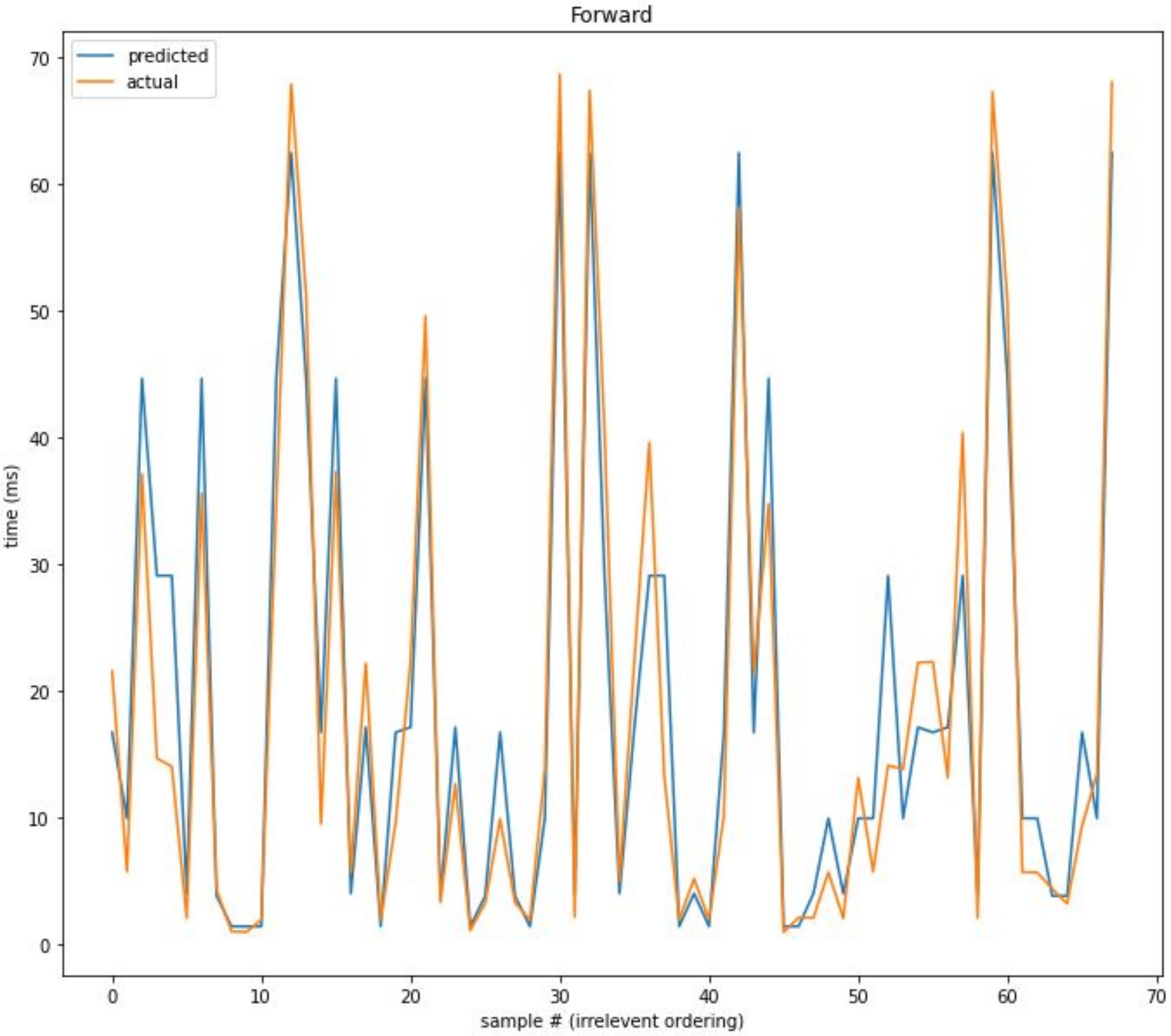
```
Conv2d layer
-----------------------------------
### Forward ####
         min: 0.3304640054702759 ms
      median: 0.3627519905567169 ms
         max: 8.35807991027832 ms
        mean: 0.3627519905567169 ms
### Backward ####
         min: 1.5809600353240967 ms
      median: 2.0004799365997314 ms
         max: 15.870783805847168 ms
        mean: 2.149602908826854 ms
```

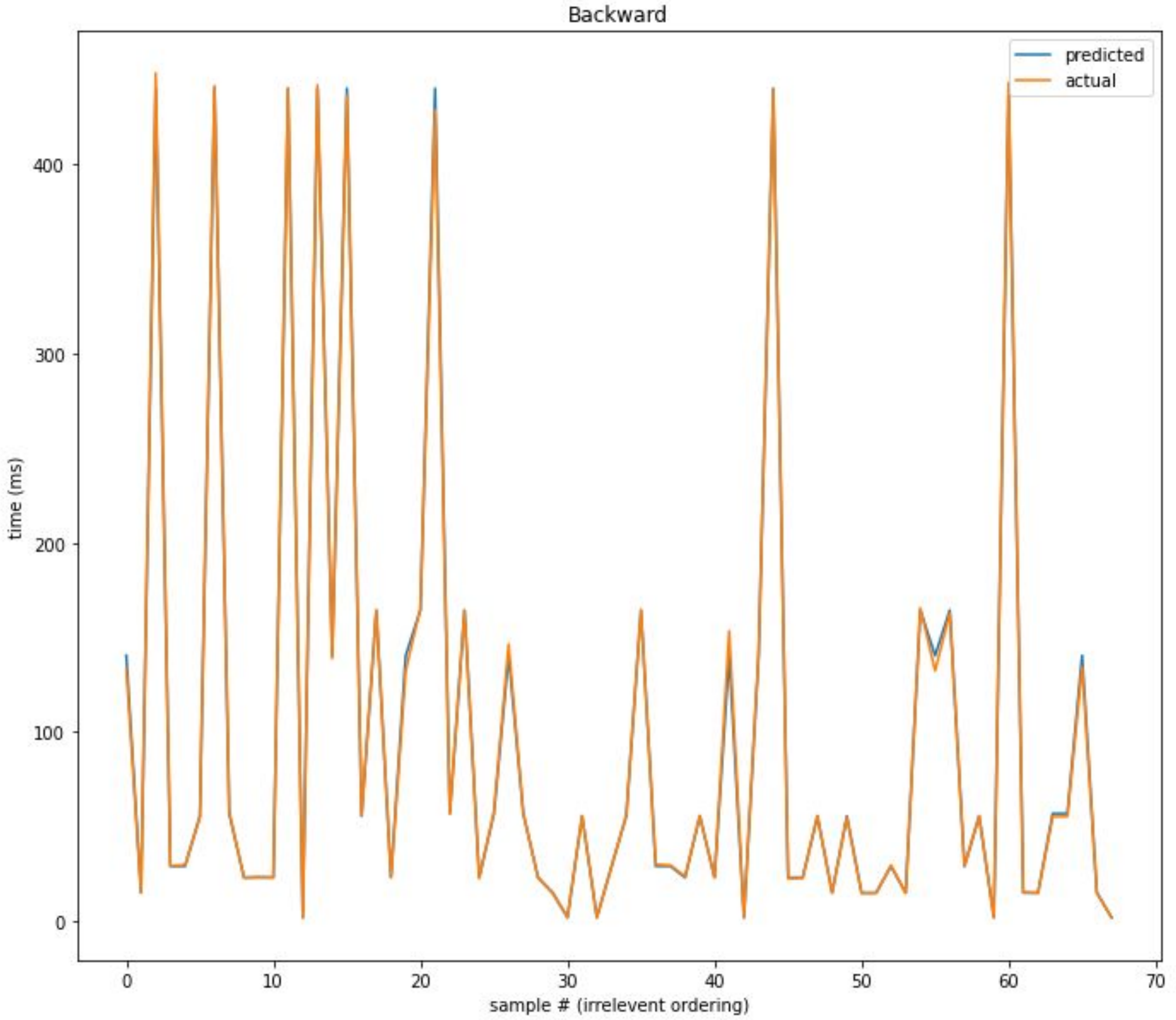
Analytical Model - Conv. Layer Time Prediction

- Prediction using polynomial curve-fitting from SKLearn framework.
  - from SKLearn import PolynomialFeatures
- Time based on following parameters:
  - In and out channels (32,64,128)
  - Image size (square) (256px)
  - kernel size (=3,5,7)
- Alternate model with different Conv. Layers

```
class Net(nn.Module):
    def __init__(self):
            super(Net, self).__init__()
            self.layers = nn.ModuleList([
                nn.Conv2d(in_channels = 3, out_channels=32, kernel_size=3, padding=1),
                nn.Conv2d(in_channels = 32, out_channels=32, kernel_size=3, padding=1),
                nn.Conv2d(in_channels = 32, out_channels=32, kernel_size=5, padding=2),
                nn.Conv2d(in_channels = 32, out_channels=32, kernel_size=7, padding=3),

                nn.Conv2d(in_channels = 32, out_channels=64, kernel_size=3, padding=1),

                nn.Conv2d(in_channels = 64, out_channels=64, kernel_size=3, padding=1),
                nn.Conv2d(in_channels = 64, out_channels=64, kernel_size=5, padding=2),
                nn.Conv2d(in_channels = 64, out_channels=64, kernel_size=7, padding=3),

                nn.Conv2d(in_channels = 64, out_channels=128, kernel_size=3, padding=1),

                nn.Conv2d(in_channels = 128, out_channels=128, kernel_size=3, padding=1),
                nn.Conv2d(in_channels = 128, out_channels=128, kernel_size=5, padding=2),
                nn.Conv2d(in_channels = 128, out_channels=128, kernel_size=7, padding=3),

                nn.AdaptiveAvgPool2d((1,1)),
                nn.Linear(128, 10),
                nn.LogSoftmax()
            ])
```

Analytical Model - Conv. Layer Time Prediction





Network Representation - Parsing The Forward Function

- No direct method to find layer/block connections in model in PyTorch

  - Using torch.jit.trace can get a string representation of the entire forward function

```
class TheModelClass(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
def forward(self,
    input: Tensor) -> Tensor:
  _0 = self.fc3
  _1 = self.fc2
  _2 = self.fc1
  _3 = self.pool2
  _4 = self.conv2
  _5 = self.pool1
  input0 = torch.relu((self.conv1).forward(input, )
)
  _6 = (_4).forward((_5).forward(input0, ), )
  input1 = torch.relu(_6)
  input2 = torch.view((_3).forward(input1, ), [-1,
400])
  input3 = torch.relu((_2).forward(input2, ))
  input4 = torch.relu((_1).forward(input3, ))
  return (_0).forward(input4, )
```

  - Parse string representation and create an adjacency list.

```
conv1:[]
pool1:['conv1']
pool2:['conv2']
conv2:['pool1']
fc1:['pool2']
fc2:['fc1']
fc3:['fc2']
```

- Visualize graph from adjacency list

Network Representation - Basic Results

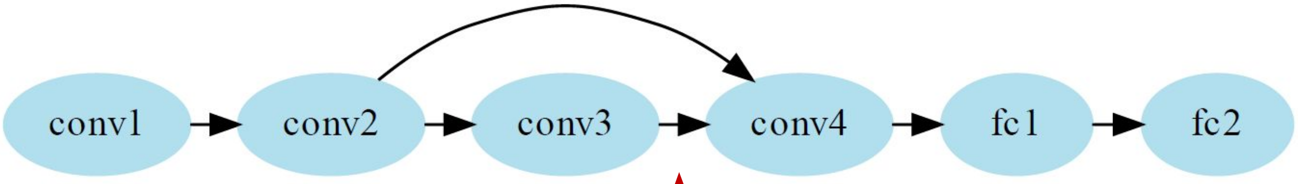
Basic CNN

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

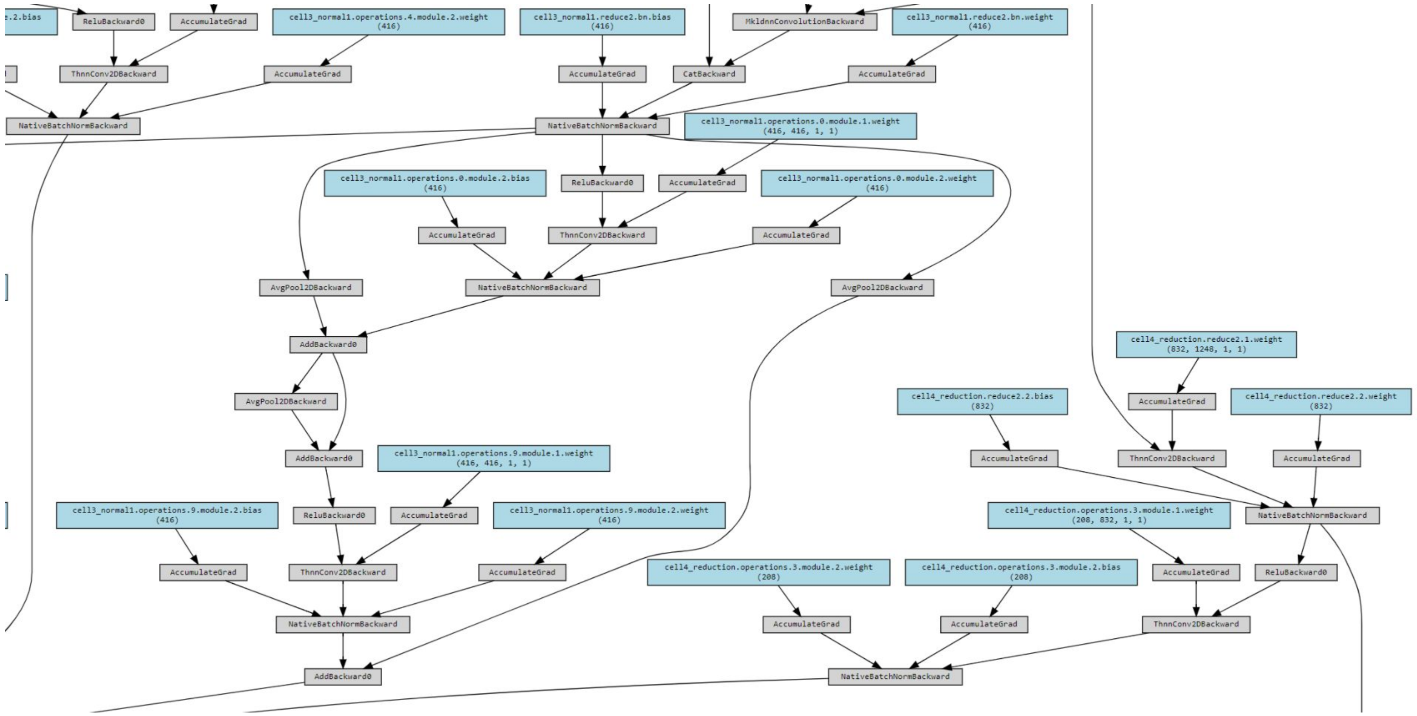
Network Representation - Interconnected Models

```python
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5, padding= 2)
        self.conv2 = nn.Conv2d(6, 16, 5, padding= 2)
        self.conv3 = nn.Conv2d(16, 6, 5, padding= 2)
        self.conv4 = nn.Conv2d(22, 6, 5, padding= 2)
        self.fc1 = nn.Linear(2322576, 120)
        self.fc2 = nn.Linear(120, 10)

    def forward(self, x):
        z = F.relu(self.conv1(x))
        z = F.relu(self.conv2(z))
        y = F.relu(self.conv3(z))
        x = self.conv4(torch.cat((z,y),1))
        x = x.view(-1, 2322576)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return x
```

Interconnected CNN



Network Representation - AmoebaNet



Output of TorchViz Library

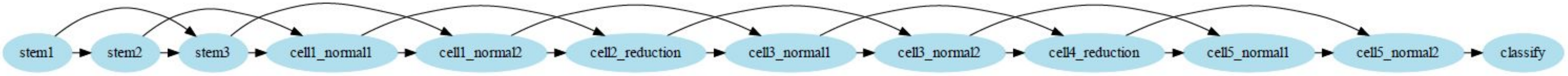
Issues

1. No one-to-one mapping with layers
2. Introduces nodes for weights and biases
3. Shows graph for backward propagation
4. No block-level abstraction

Network Representation - AmoebaNet

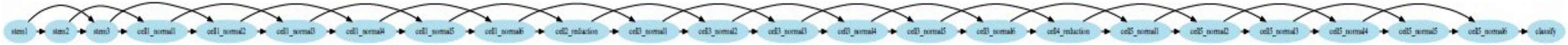
AmoebaNet 6-layers

(Block-level representation)



AmoebaNet 18-layers

(Block-level representation)

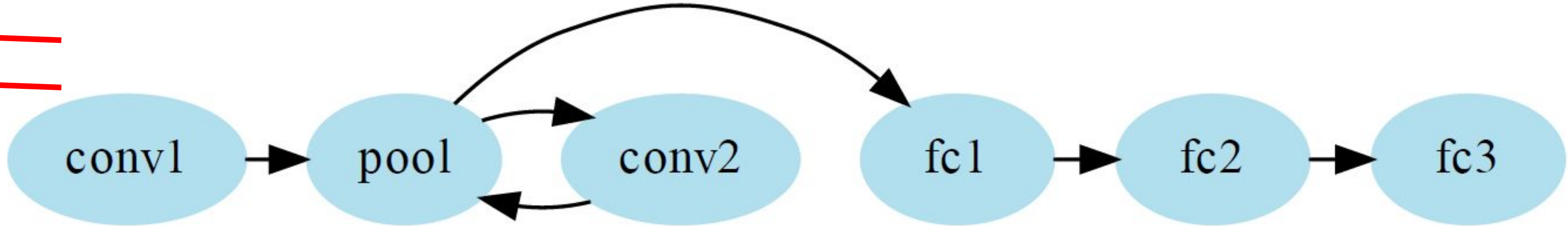


Note: each block can be recursively extended to a full graph

Network Representation - Areas of improvement

```python
class Net(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

- Same layer used twice as input and output of conv2
- Logically viable, but creates cycle in graph
- Difficult to split the model using this graph



Proposed solution: duplicate layer and rename it

CPU offloading of out-of-core models

- **Research Question:**
  - What is the optimal approach to apply CPU-offloading techniques for training out-of-core deep learning models?
- **Definition:**
  - **CPU-offloading:**
    - Moving some memory from GPU to CPU during training
  - **pin_momory:**
    - For data loading, passing pin_memory=True to a DataLoader will automatically put the fetched data Tensors in pinned memory, and thus enables faster data transfer to CUDA-enabled GPUs
  - **Nonblocking:**
    - Allow asynchronous GPU copie. In other word, we can bypass synchronization when it is unnecessary.

Reference:https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/

- **Approaches:**
  a. Baseline: No CPU-offloading, everything is trained on GPU
  b. CPU-offloading without any optimization
  c. CPU-offloading with pin-memory
  d. CPU-offloading with non-blocking mechanism
  e. CPU-offloading with pin-memory and non-blocking mechanism

- **Notes:**
  - 1 epoch = 8 step, each step use 32 sample as batchsize
  - Trained with 20 epochs, so 20 * 8 - 5= 155 iteration (Remove the first 5 outliers)
  - Vgg19 was tested on RI2 sky-k80, and the rest were tested on RI2 bdw-v100

CPU offloading

Approaches	VGG19 (sky-k80)	VGG19 (bdw-v100)	AlexNet (bdw-v100)	ResNet50 (bdw-v100)	InceptionV3 (bdw-v100)
Baseline on GPU	94.4539	30.0673	24.1323	26.1702	32.2195
Naïve CPU-offloading	179.9194	150.5116	42.2092	113.5793	124.2807
with pin-memory	194.5803	166.7242	47.4363	110.0957	120.2433
non-blocking	179.8007	153.9791	41.5072	113.6041	123.1403
pin-memory and non-blocking	190.4785	160.9983	47.5516	109.3674	119.2649

Table1: Measures the total of training time in sec for 20 epochs on Hymenoptera dataset, https://download.pytorch.org/tutorial/hymenoptera_data.zip; sky-k80 refers to skylake CPU with TESLA K80, and bdw-v100 refers to broadwell CPU with V100.

Model	Number of Parameter
VGG19	139,578,434
AlexNet	57,012,034
ResNet50	23,512,130
InceptionV3	24,348,900

Table2: The total number of parameters contained in models.

Key Takeaway

- CPU offloading provides the potential to train larger out-off-core models but also comes with the cost of I/O communication overhead.
- CPU offloading time is affected by underlying hardware architecture:
  - sky-k80 tripled the total training time of bdw-v100 on baseline model.
- CPU offloading time is influenced by the number of parameters in a tensor:
  - Fewer and denser tensors can accelerate the training.
- Optimizations help in models with more number of layers and parameters:
  - The CPU Offloading optimization methods have effect only on ResNet50 and InceptionV3.

Future Work

- Integrate Analytical model and Network representation code to split the model automatically
  - Use network representation to get model splits at different level (block-level, layer-level, and module-level)
  - Use analytical model to estimate the time for each layer/block/module and divide the model into splits (logically)
  - Use model generator to create different model splits
- Overlap CPU-offloading with computation to minimize cost and train larger models.
  - Improve the performance of model-parallelism by increasing the model size trainable on a single GPU.

Thank You for Your Time and Attention!

Hamza.23@osu.edu

Alnaasan.1@osu.edu

Dong.760@osu.edu

Jain.575@osu.edu

CPU offloading with vgg19 on sky-k80

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (median)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (median)
Baseline on GPU	94.4539	0.9739								
Naïve CPU-offloading	179.9194	0.9804	0.1069	0.1450	0.0646	0.1173	0.4836	0.5842	0.4358	0.4825
with pin-memory	194.5803	0.9739	0.1131	0.1414	0.0651	0.1184	0.4816	0.6069	0.4293	0.4805
non-blocking	179.8007	0.9739	0.0863	0.1329	0.0628	0.0682	0.4880	0.6082	0.4291	0.4891
pin-memory and non-blocking	190.4785	0.9804	0.0713	0.1310	0.0631	0.0657	0.4917	0.6061	0.4429	0.4931

Table: CPU-offloading evaluation of vgg19 on RI2 sky-k80 (139,578,434 params in total)

CPU offloading with vgg19 on bdw-v100

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (median)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (median)
Baseline on GPU	30.0673	0.9673								
Naïve CPU-offloading	150.5116	0.9739	0.0678	0.0935	0.0584	0.0644	0.2148	0.2672	0.1952	0.2082
with pin-memory	166.7242	0.9673	0.0640	0.0993	0.0591	0.0615	0.2346	0.2758	0.1966	0.2378
non-blocking	153.9791	0.9608	0.0665	0.0916	0.0575	0.0628	0.2159	0.2644	0.1968	0.2089
pin-memory and non-blocking	160.9983	0.9608	0.0648	0.0939	0.0584	0.0630	0.2100:	0.2634	0.1949	0.2068

Table: CPU-offloading evaluation of vgg19 on RI2 bdw-v100 (139,578,434 params in total)

CPU offloading with AlexNet on bdw-v100

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (median)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (median)
Baseline on GPU	24.1323	0.9085								
Naïve CPU-offloading	42.2092	0.9150	0.0285	0.0383	0.0243	0.0270	0.0762	0.0888	0.0680	0.0735
with pin-memory	47.4363	0.9346	0.0262	0.0384	0.0241	0.0251	0.0804	0.0924	0.0675	0.0811
non-blocking	41.5072	0.9346	0.0276	0.0572	0.0237	0.0261	0.0761	0.0898	0.0679	0.0729
pin-memory and non-blocking	47.5516	0.9216	0.0252	0.0360	0.0235	0.0242	0.0816	0.0933	0.0683	0.0812

Table: CPU-offloading evaluation of AlexNet on RI2 bdw-v100(57,012,034 params in total)

CPU offloading with ResNet50 on bdw-v100

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (medium)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (median)
Baseline on GPU	26.1702	0.9477								
Naïve CPU-offloading	113.5793	0.9608	0.0250	0.0539	0.0216	0.0238	0.0461	0.0714	0.0292	0.0423
with pin-memory	110.0957	0.9608	0.0252	0.0430	0.0221	0.0229	0.0463	0.0721	0.0311	0.0445
non-blocking	113.6041	0.9608	0.0205	0.0323	0.0170	0.0197	0.0474	0.0771	0.0307	0.0446
pin-memory and non-blocking	109.3674	0.9608	0.0208	0.0400	0.0172	0.0183	0.0457	0.0746	0.0310	0.0425

Table: CPU-offloading evaluation of ResNet50 on RI2 bdw-v100 (23,512,130 params in total)

CPU offloading with InceptionV3 on bdw-v100

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (median)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (median)
Baseline on GPU	32.2195	0.9281								
Naïve CPU-offloading	124.2807	0.9477	0.0352	0.0471	0.0317	0.0339	0.0555	0.0890	0.0390	0.0523
with pin-memory	120.2433	0.9346	0.0358	0.0575	0.0317	0.0328	0.0562	0.0990	0.0393	0.0534
non-blocking	123.1403	0.9477	0.0280	0.0394	0.0241	0.0271	0.0605	0.0973	0.0407	0.0602
pin-memory and non-blocking	119.2649	0.9281	0.0283	0.0620	0.0241	0.0254	0.0574	0.1029	0.0402	0.0532

Table: CPU-offloading evaluation of InceptionV3 on RI2 bdw-v100 (24,348,900 params in total)