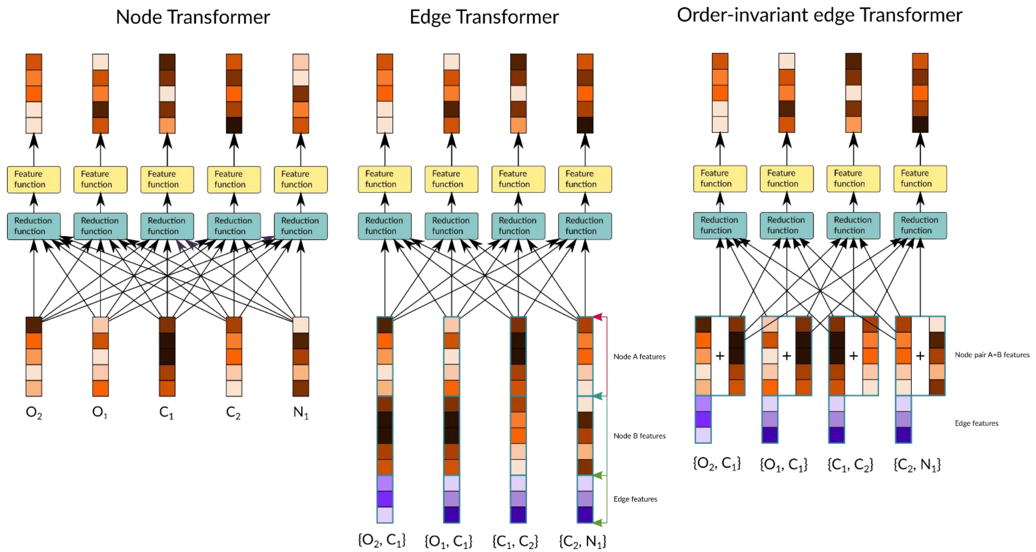
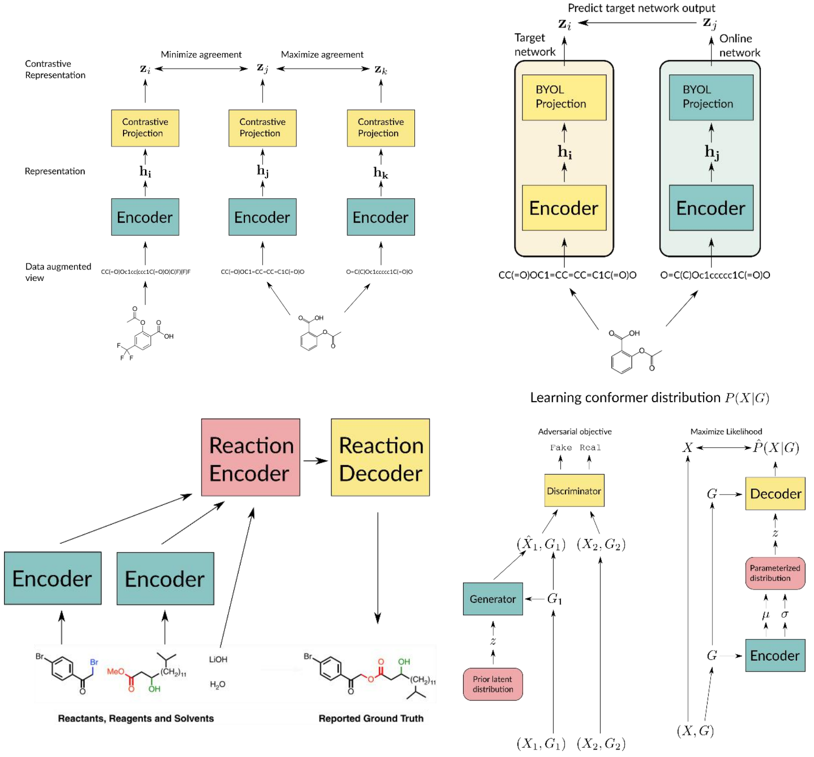
Optimizing deep learning

Erik Ylipää (erik.ylipaa@ri.se)

My research



Graph Transformers



Pretraining of
Molecule encoders

The EuroCC National EuroHPC Competence Center Sweden (ENCCS)

The EuroCC National EuroHPC Competence Center Sweden (ENCCS) is a joint initiative between the ten main Swedish research universities and RISE Research Institutes of Sweden. The center is hosted by Uppsala University (UU) on behalf of the consortium and will include the relevant competences at the other nodes. The initiative is funded by the EuroHPC JU, Swedish Research Council (Vetenskapsrådet) and the Swedish Innovation Agency (Vinnova). It is designed to prioritize support based on

- Needs of academic users with large scale allocations such as PRACE allocations in EU and SNIC allocations in Sweden
- The current industrial usage of HPC and their future HPC and Artificial Intelligence (AI) needs
- Needs for training and support to enable a wide range of Swedish users to use the new hardware deployed in pre-exascale systems, in particular Euro HPC JU (pre)exa-scale system LUMI

Profilers

- nvprof - deprecated profiler for CUDA
  - Nsight Systems should be used instead
- py-spy - sampling profiler for python code
- line_profiler - profile each line of a function

Logging GPU usage

- nvidia-smi --query-gpu=timestamp,pci.bus_id,utilization.gpu,utilization.memory --format=csv -l 1
- Small script for online logging: https://github.com/eryl/gpulog

CUDA semantics in PyTorch

- Cuda calls from our frameworks are generally asynchronous
- This might make profiling information at the python-level misleading, the time spent at one location in python code might be due to waiting for a previous call to finish

NVprof (deprecated)

- Used to profile CUDA API calls
- As a deep learning researcher, this is typically below the level of abstraction we work at
- I’ve mainly used it to understand if memory copy from host to device is an issue

Installing nvprof

- If using ubuntu package manager to install NVIDIA software, available as nvidia-profiler
  - apt install nvidia-profiler
- By default requires privileged access
- On the AIDA DGX-2, it seems to work out of the box

profiling with nvprof

- For the use case of analyzing memory copies, default invocation works fine
- `nvprof --log-file nvprof_log.txt python script.py`
- This will dump profiling output to the text file nvprof_log.txt

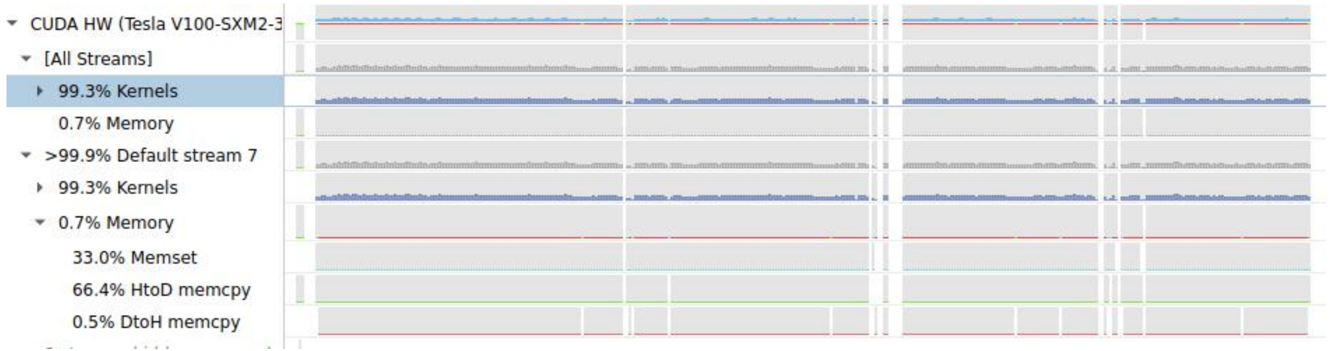
nvprof output

- The output shows what cuda kernels have been run on the GPU(s) how much % of time they have used
- Here I typically look to [CUDA memcpy HtoD], how much time was spent copying data from the host to the device
- For deep learning programs, this is typically quite low (0.12% in this case)

```
==40242== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   11.36%  19.0811s   1180113  16.168us     736ns  9.2059ms  void at::native::ve
                    9.08%  15.2482s     57720  264.18us  221.13us  6.3191ms  void cudnn::cnn::wg
                    6.57%  11.0269s     39000  282.74us  168.87us  7.4494ms  void cudnn::detail:
                    6.28%  10.5512s     61390  171.87us  151.21us  7.4208ms  maxwell_scudnn_wino
                    5.66%  9.50446s    265156  35.844us  1.6640us  9.3531ms  _ZN2at6native29vect
                    4.95%  8.31429s     36136  230.08us  110.79us  8.1986ms  void precomputed_co
                    4.52%  7.59856s    111540  68.124us  29.346us  5.8234ms  void cudnn::bn_bw_1
                    4.51%  7.57257s   1092780  6.9290us     736ns  6.4271ms  void at::native::ve
                    4.08%  6.86087s     41884  163.81us  130.63us  5.5894ms  maxwell_scudnn_128x
                    3.88%  6.51599s    111540  58.418us  24.257us  9.1865ms  void cudnn::bn_fw_t
                    3.01%  5.06350s     36660  138.12us  82.243us  9.0224ms  maxwell_sgemm_128x6
                    2.93%  4.92540s    364260  13.521us     960ns  2.3991ms  _ZN2at6native29vect
                    2.52%  4.23946s     13260  319.72us  153.10us  10.500ms  void cudnn::cnn::wg
                    2.48%  4.16063s     27300  152.40us  137.41us  4.3984ms  maxwell_scudnn_128x
                    2.39%  4.01173s     77264  51.922us  12.192us  4.3108ms  void cudnn::winogra
                    2.30%  3.85825s     36660  105.24us  9.3440us  4.7192ms  void cudnn::winogra
                    2.14%  3.59595s    364260  9.8710us     768ns  4.4021ms  _ZN2at6native29vect
                    1.46%  2.45122s      4680  523.77us  465.24us  619.26us  void cudnn::cnn::wg
                    1.36%  2.27590s    364260  6.2480us     896ns  2.8016ms  _ZN2at6native29vect
                    1.26%  2.11418s     12362  171.02us  84.004us  12.024ms  maxwell_scudnn_wino
                    1.25%  2.10078s      7866  267.07us  133.64us  4.8152ms  void precomputed_co
                    1.20%  2.02146s    364260  5.5490us     736ns  1.0458ms  void at::native::ve
                    1.09%  1.82651s      5460  334.53us  238.00us  621.82us  void cudnn::cnn::wg
                    0.96%  1.61851s      8580  188.64us  82.148us  5.0181ms  void cudnn::bn_bw_1
                    0.95%  1.59084s      6240  254.94us  162.41us  10.394ms  maxwell_scudnn_128x
                    0.87%  1.46213s      8792  166.30us  84.163us  7.2406ms  maxwell_scudnn_128x
                    0.82%  1.38390s    366287  3.7780us     608ns  2.1559ms  void at::native::ve
                    0.73%  1.21785s      3896  312.59us  222.22us  4.7840ms  void precomputed_co
                    0.67%  1.12980s      8580  131.68us  58.371us  2.2757ms  void cudnn::bn_fw_t
                    0.67%  1.11785s     36660  30.492us  13.249us  2.0194ms  void cudnn::winogra
                    0.63%  1.06592s     30380  35.086us  4.4480us  5.5235ms  void cudnn::bn_fw_i
                    0.63%  1.05865s     36660  28.877us  11.457us  2.3371ms  void cudnn::winogra
                    0.61%  1.03286s      5268  196.06us  99.076us  5.1863ms  maxwell_scudnn_wino
                    0.55%  924.02ms      5460  169.23us  157.22us  4.2721ms  maxwell_scudnn_128x
                    0.49%  819.03ms      1560  525.02us  434.16us  724.67us  void cudnn::detail:
                    0.42%  699.09ms      2340  298.76us  250.95us  4.2314ms  void cudnn::detail:
                    0.37%  623.92ms       978  637.95us  242.00us  11.460ms  void explicit_convo
                    0.37%  620.00ms      1560  397.44us  355.54us  462.61us  maxwell_scudnn_128x
                    0.32%  539.48ms       780  691.64us  674.24us  731.81us  maxwell_scudnn_128x
                    0.30%  511.08ms     97092  5.2630us  1.9200us  8.3843ms  cask_cudnn::compute
                    0.30%  498.11ms      1952  255.18us  123.46us  4.8029ms  maxwell_scudnn_wino
                    0.29%  493.52ms      1948  253.35us  230.89us  1.5981ms  maxwell_scudnn_128x
                    0.27%  458.24ms       780  587.49us  564.44us  680.09us  void at::native::_G
                    0.24%  396.89ms     42900  9.2510us  2.6560us  631.07us  void cudnn::ops::sc
                    0.21%  353.43ms      1560  226.56us  218.09us  239.50us  maxwell_scudnn_wino
                    0.21%  351.23ms      1952  179.94us  37.218us  1.1780ms  maxwell_scudnn_128x
                    0.20%  331.22ms       780  424.64us  418.51us  451.28us  void cudnn::bn_bw_1
                    0.19%  315.51ms       976  323.26us  172.58us  748.32us  maxwell_scudnn_128x
                    0.18%  299.01ms       976  306.36us  161.90us  624.06us  maxwell_scudnn_128x
                    0.15%  252.80ms       780  324.11us  316.72us  387.54us  maxwell_scudnn_128x
                    0.15%  249.31ms    120900  2.0620us  1.3120us  131.91us  void at::native::ve
                    0.14%  236.76ms       780  303.54us  297.10us  313.52us  void cudnn::bn_fw_t
                    0.12%  204.61ms      2888  70.846us     672ns  869.67us  [CUDA memcpy HtoD]
                    0.12%  198.07ms       780  253.94us  249.04us  303.21us  maxwell_scudnn_128x
                    0.11%  190.44ms     47898  3.9750us  1.8240us  579.93us  void cudnn::cnn::ke
```

Nsight

- The recommended system for profiling with a lot more bells and whistles
  - Command line utility similar to nvprof: nsys
- nsys profile --trace=cuda,cudnn,cublas,osrt,nvtx python simply_resnet.py /raid/erik/datasets/imagenet_subset/ --device cuda:4
- Doesn’t require priviliged access. Produces reports by default which can be analyzed in GUI tool



py-spy

- Profile the python-part of code
- Install with pip: pip install py-spy
- Can produce multiple different outputs: flame-graphs, speedscope and raw outputs

py-spy

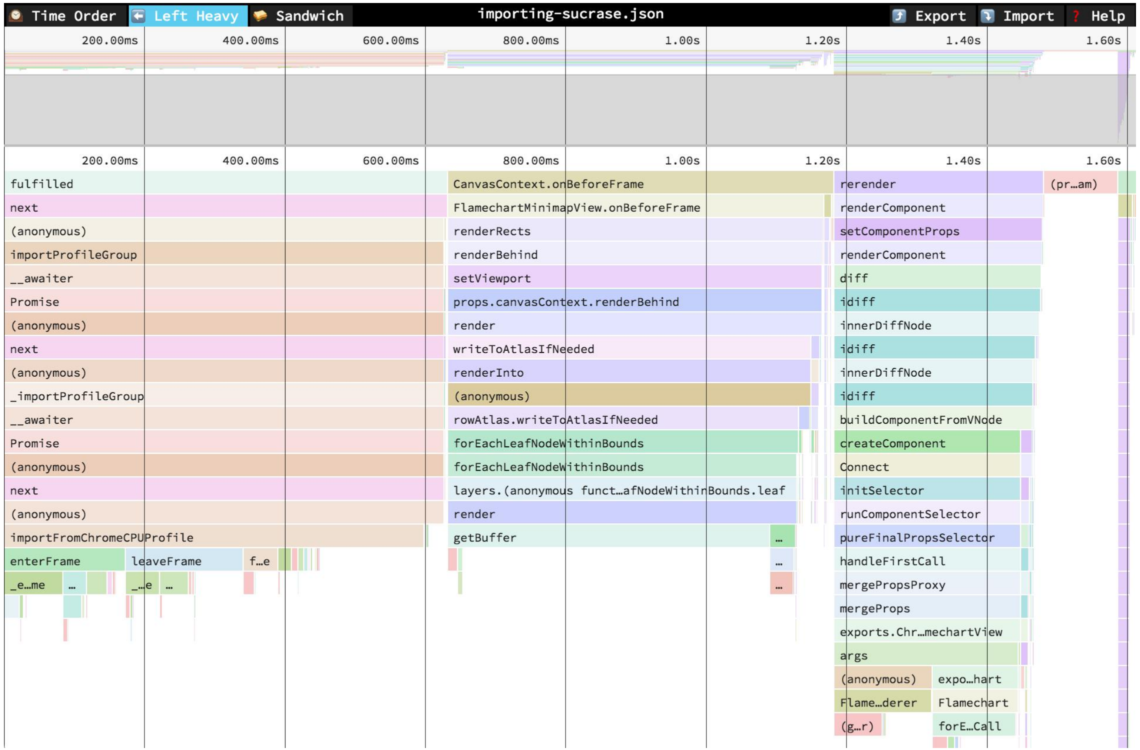
- py-spy record --output py-spy-profile.svg -- python simply_resnet.py /data/datasets/imagenet_subset/ --device cuda:0 --pin-memory --num-workers 12
- By default this creates a *flamegraph*, here written to py-spy-profile.svg
- Can also produce speedscope files and raw data dumps

Flamegraph

- Gives a quick overview of what function has been most active
- X-axis shows percentage of use at a certain stack depth (order is typically alphapetical, not time)
- Y-axis shows call stack depth, so a “flame” is essentially a branch of the call tree

Speedscope

- Alternative to flame graph, interactive with more ways to view profiles
- View profiles with web app: https://www.speedscope.app/
- Install locally from https://github.com/jlfwong/speedscope



Line_profiler

- Tool to do fine grained and targeted profiling
- Needs to modify the script to profile
- Install with pip:
  - pip install line_profiler
- The first hit on Google is the deprecated repo, this is the correct one: https://github.com/pyutils/line_profiler

Modifying script for line_profiler

- The line profiler program looks for functions decorated with the `@profile` decorator
- Add this to the functions you wish to line-profile
- To profile you script, call kernprof -l instead of python
  - `kernprof -l script.py`
- This dumps profiling information to a file as the script, with an .lprof extension

Inspect line_profiler dumps

- use the line_profiler module to read the .lprof files produced by kernprof -l
  - `python -m line_profiler script.py.lprof`

```
123         1      50763.0  50763.0      0.0          if not torch.cuda.is_available():
124                                                       raise RuntimeError(f"CUDA not available and device set to {args.device}")
125                                                   else:
126         1        252.0    252.0      0.0              device = torch.device(args.device)
127         1         12.0     12.0      0.0              torch.backends.cudnn.benchmark = True
128
129         1         36.0     36.0      0.0      print(f"Device is set to {device}")
130
131         1      40725.0  40725.0      0.0      train_set, dev_set, test_set = make_datasets(args.images, rng=rng)
132
133         1          1.0      1.0      0.0      batch_size = 4
134         1          1.0      1.0      0.0      max_epochs = 1
135
136         1          1.0      1.0      0.0      dataloader_kwargs = dict()
137         1          1.0      1.0      0.0      if args.pin_memory:
138                                                   dataloader_kwargs['pin_memory'] = True
139         1          1.0      1.0      0.0      if args.num_workers:
140                                                   dataloader_kwargs['num_workers'] = args.num_workers
141         1          1.0      1.0      0.0      if args.prefetch_factor:
142                                                   dataloader_kwargs['prefetch_factor'] = args.num_workers
143         1         68.0     68.0      0.0      training_loader = DataLoader(train_set, batch_size=batch_size, **dataloader_kwargs)
144         1         39.0     39.0      0.0      dev_loader = DataLoader(dev_set, batch_size=batch_size, **dataloader_kwargs)
145         1         37.0     37.0      0.0      test_loader = DataLoader(test_set, batch_size=batch_size, **dataloader_kwargs)
146
147         1     657171.0 657171.0      0.3      model = resnet152(pretrained=False, num_classes=train_set.num_classes)
148                                               #model = resnet18(pretrained=False, num_classes=train_set.num_classes)
149                                               #model = LogisticRegression(train_set[0][0].shape, train_set.num_classes)
150                                               #model = alexnet(pretrained=False, num_classes=train_set.num_classes)
151         1   19973237.0 19973237.0      9.3    model.to(device)
152
153         1         97.0     97.0      0.0      loss_fn = nn.CrossEntropyLoss()
154         1       9452.0   9452.0      0.0      optimizer = Adam(model.parameters(), lr=1e-3, weight_decay=3e-7)
155
156         2          7.0      3.5      0.0      for epoch in range(max_epochs):
157         1          2.0      2.0      0.0          training_losses = []
158       781   12159874.0  15569.6      5.7          for x, y in tqdm(training_loader, desc='training progress'):
159       780    9967371.0  12778.7      4.7              optimizer.zero_grad()
160       780   26537897.0  34022.9     12.4              prediction = model(x.to(device))
161       780   17518033.0  22459.0      8.2              loss = loss_fn(prediction, y.to(device))
162       780   32254004.0  41351.3     15.1              loss.backward()
163       780   71446672.0  91598.3     33.4              optimizer.step()
164       780    8935548.0  11455.8      4.2              training_losses.append(loss.item())
165         1      72525.0  72525.0      0.0          print(f'Training loss: {np.mean(training_losses)}')
166
167         1          3.0      3.0      0.0          val_losses = []
168         1       5515.0   5515.0      0.0          model.eval()
169         1         18.0     18.0      0.0          with torch.no_grad():
170        99    2383925.0  24080.1      1.1              for x,y in dev_loader:
171        98    2640635.0  26945.3      1.2                  prediction = model(x.to(device))
172        98    2339464.0  23872.1      1.1                  loss = loss_fn(prediction, y.to(device))
```

line_profiler tip

- The @profile decorator gets defined in the global scope of your script *when* you invoke it using kernprof. It will not be available otherwise
- To quickly go between supporting line_profiler and running normally, add this snippet to your script
  - It defines a dummy @profile decorator if there isn't one

```
try:
    @profile
    def foo():
        pass
    del foo
except
NameError:
    def
profile(f):
        return f
```

QnA on profiling

Efficient PyTorch with DeepSpeed

Data loaders

- Data loaders are essential to good performance
- PyTorch `torch.util.data.DataLoader` has two important settings
  - `num_workers`, setting this to a positive integer handles data loading in separate processes. Set this to as many CPU threads as you have. Each process will load and collate one batch, making sure that the data loading does not block the main training loop. This hides latency of processing even if it’s relatively costly.
  - `pin_memory`, set this to True to make the returned tensors be placed in main memory pinned by CUDA, which makes transfer to the GPU faster

DataParalell vs DistributedDataParallel vs Deepspeed

- The recommended way of doing data parallel workloads in pytorch is the Distributed Data Parallel framework

- There is the more easy to use DataParallel, but this has performance issues since a single process handles all synchronization across GPUs (see this excellent write-up: https://www.telesens.co/2019/04/04/distributed-data-parallel-training-using-pytorch-on-aws/)

- Distributed Data Parallel is relatively easy to use, but documentation is a bit scattered

- There's another framework which does the same thing, and a lot more: Deepspeed from Microsoft, https://github.com/microsoft/DeepSpeed, deepspeed.ai

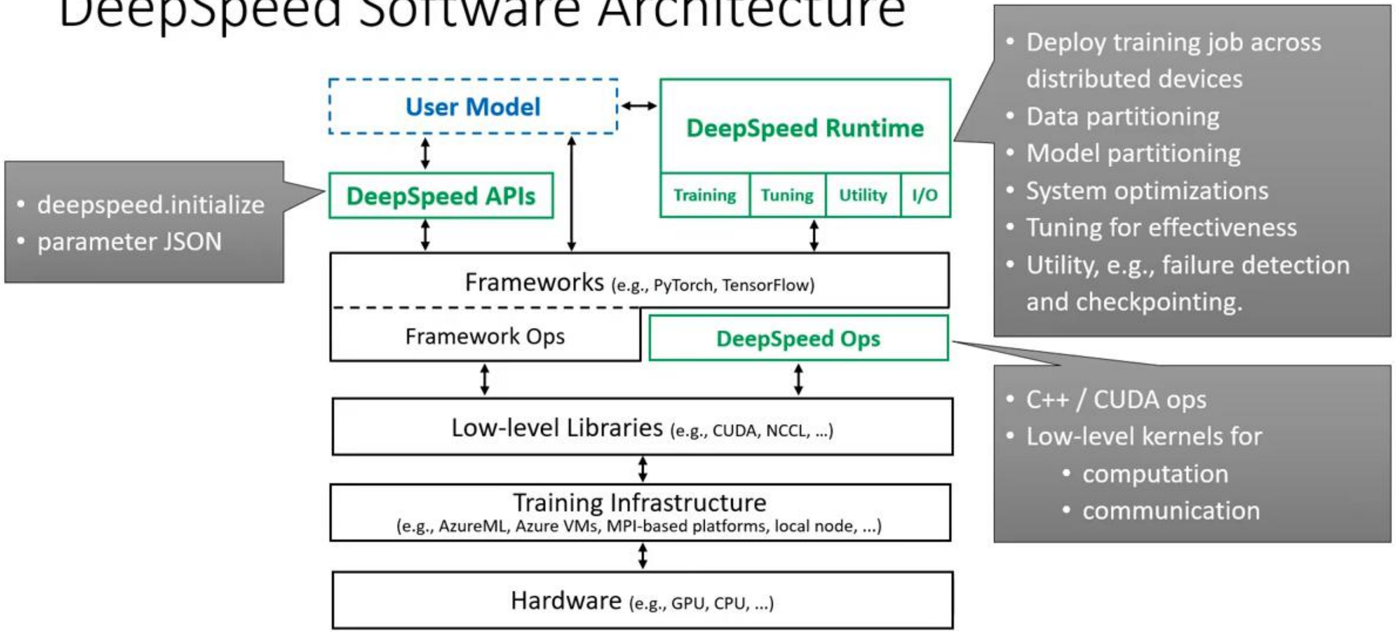
Deepspeed

- Relatively new framework (there are definitely rough edges)
- Designed to take away as much of the hustle of writing efficient deep learning programs as possible
- Distributed learning, same code works as well for single-node+single-gpu/single-node+multiple-gpu/multiple-nodes+multiple-gpus
- Efficient ways of synchronizing gradients with custom optimizers
- Built in support for fp16 and mixed precision training (AMP) using NVIDIA's Apex package as well as automatically handling of loss scaling and correct gradient clipping
- Ways of easily doing model parallel training with pipelining, and combining it with data parallel training

Deepspeed

- Optimization framework Zero, for sharding optimizer state/gradients/parameters over multiple GPUs.
  - Can drastically reduce memory usage with minor impact on performance
  - The first stage of this optimization shards the optimizer states (e.g. ADAMSs moving averages) so that each worker only has parts of the state
- One driver for deepspeed are huge Language Models, so framework also contains custom efficient kernels for Transformers
- Also has some support for one-cycle style LR policies, including learning rate range finders. There seems to be gathering evidence that this is a good idea for large batch training.

DeepSpeed Software Architecture



1. Optimizations on top of training framework vs closely coupled
2. Agnostic of training infrastructure

[Hands-on Tutorials] DeepSpeed 01, KDD2020
https://youtu.be/CaseqC45DNc

Deepspeed gotchas

- Includes many custom PyTorch extensions (e.g. optimizers) which will be JIT-compiled when running
  - Needs full build environment (so full CUDA-install and C++ compiler and standard libraries)
  - This shouldn't be a problem at AIDA where you have full control over your VM

Deepspeed install

- First install CUDA of desired version
- Install pytorch (recommended through anaconda) to match the installed cuda version
- Install deepspeed with pip:
  - pip install deepspeed

Modifying a script to use deepspeed

- The script will be launched through deepspeeds launcher script. Certain command line arguments are assumed to be available so run the following on your argparse.ArgumentParser object:
  - `parser = deepspeed.add_config_arguments(parser)`
- I had to add the following explicitly:
  - `parser.add_argument('--local_rank', type=int, default=-1, help='local rank passed from distributed launcher')`

Deepspeed configuration

- Deepspeed is configured using a JSON file with a particular set of values
- Most parameters for training is changed here, such as batch size, gradient accumulation steps, optimizer, scheduler, 16bit precision training, model parallelism and Zero configuration

```json
{
  "train_batch_size": 4,
  "steps_per_print": 2000,
  "optimizer": {
    "type": "Adam",
    "params": {
      "lr": 0.001,
      "betas": [
        0.8,
        0.999
      ],
      "eps": 1e-8,
      "weight_decay": 3e-7
    }
  },
  "scheduler": {
    "type": "WarmupLR",
    "params": {
      "warmup_min_lr": 0,
      "warmup_max_lr": 0.001,
      "warmup_num_steps": 1000
    }
  },
  "wall_clock_breakdown": false
}
```

Modifying a script to use deepspeed

- Deepspeed provides a wrapper around any torch.nn.Module object
- To use deepspeed, wrap your model in the deepspeed.initialize function:
  - ```
    model_engine, optimizer, _, __ =
    deepspeed.initialize(args=args, model=model,
    model_parameters=model.parameters())
    ```
- The args is your command line arguments which deepspeed will go through to find its configuration
- The model_engine object is now our wrapped model and will behave
    ```

Deepspeed dataloaders

- Deepspeed will work with PyTorchs dataloader, but to handle parallels sampling without extra steps use:
  - `training_loader = model_engine.deepspeed_io(train_set)`
- This wrapper also accept some of the same arguments as PyTorch dataloaders, in particular collate_fn and pin_memory
  - Other arguments like batch size or sampler can be left to the wrapper

Modifying script to use deepspeed

- Since we assume data parallel, we need to let deepsped determine where to place tensors
- When we move a tensor to device, use `model_engine.local_rank`
  - `prediction = model_engine(x.to(model_engine.local_rank)`
- This takes care of placing the tensor on the correct GPU based on the process running the code

Modifying a script to use deepspeed

- Deepspeed handles the optimizer steps, as well has the backpropagation
- We need to replace our typical calls to the optimizer with specific calls from the model engine
  - Don’t call the backward of the loss node (`loss.backward()`), instead use `model_engine.backward(loss)`
  - Don't call `optimizer.zero_grads()` or `optimizer.step()`, instead call `model_engine.step()`. It will take care of zeroing gradients when appropriate.

Running the new script

- We need to run the script using deepspeeds launcher:
  - `deepspeed simply_resnet_deepspeed.py /data/datasets/imagenet_subset/ --deepspeed_config ds_config.json`
- This takes care of setting up the runtime environment and inject the correct information in all processes

Training large batches

- Training with large batches has historically generated worse models than with small batches
- Normalizations (such as BatchNorm) seems to help with this
- Learning rate schedule also seems to be important (LAMB and One-Cycle policy seems to be gaining momentum)

ABSTRACT

Training large deep neural networks on massive datasets is computationally very challenging. There has been recent surge in interest in using *large batch* stochastic optimization methods to tackle this issue. The most prominent algorithm in this line of research is LARS, which by employing *layerwise adaptive* learning rates trains RESNET on ImageNet in a few minutes. However, LARS performs poorly for attention models like BERT, indicating that its performance gains are *not* consistent across tasks. In this paper, we first study a principled layerwise adaptation strategy to accelerate training of deep neural networks using large mini-batches. Using this strategy, we develop a new layerwise adaptive large batch optimization technique called LAMB; we then provide convergence analysis of LAMB as well as LARS, showing convergence to a stationary point in general nonconvex settings. Our empirical results demonstrate the superior performance of LAMB across various tasks such as BERT and RESNET-50 training with very little hyperparameter tuning. In particular, for BERT training, our optimizer enables use of very large batch sizes of 32868 without any degradation of performance. By increasing the batch size to the memory limit of a TPUv3 Pod, BERT training time can be reduced from 3 days to just 76 minutes (Table 1). The LAMB implementation is available online[1].

You, Yang, et al. "Large batch optimization for deep learning: Training bert in 76 minutes." ICLR 2020

QnA on deepspeed

Erik Ylipää

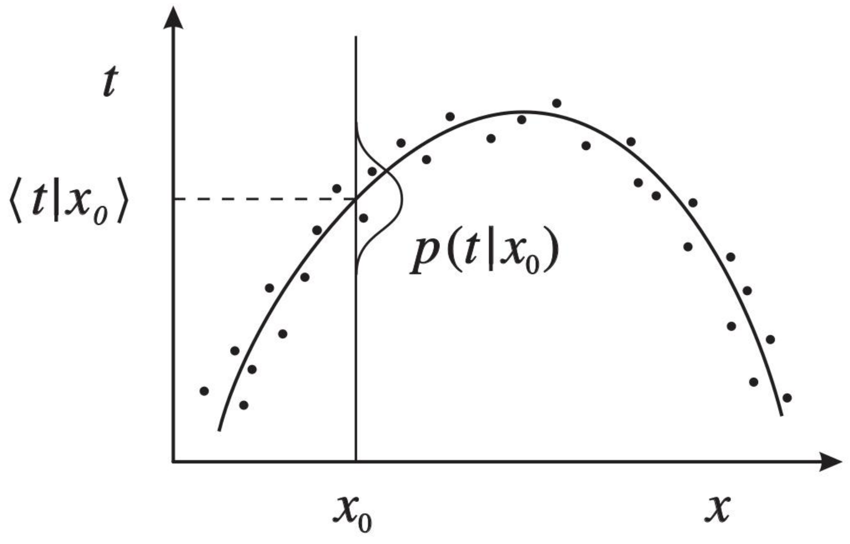
erik.ylipaa@ri.se

One-hot-encodings

- Never use one-hot-encodings as inputs to neural networks
  - Instead use an Embedding layer with integer encoded categorical values
- There are two issues, 1) inefficiency and 2) optimization:
  1. Multiplying a one-hot-vector with a matrix is mostly wasted computation, multiplying all rows of the matrix except one with 0
  2. If the vocabulary is large, or the frequency distribution is very skewed, optimizers which accumulates gradient statistics (i.e ADAM, RMSProp, SGD+momentum) have no chance of applying sparse updates to those statistics

Regression targets

- If we model regression with a linear output layer and Mean Squared Error loss the implicit assumption is that the conditional distribution is Gaussian
- In particular, we will predict the conditional mean

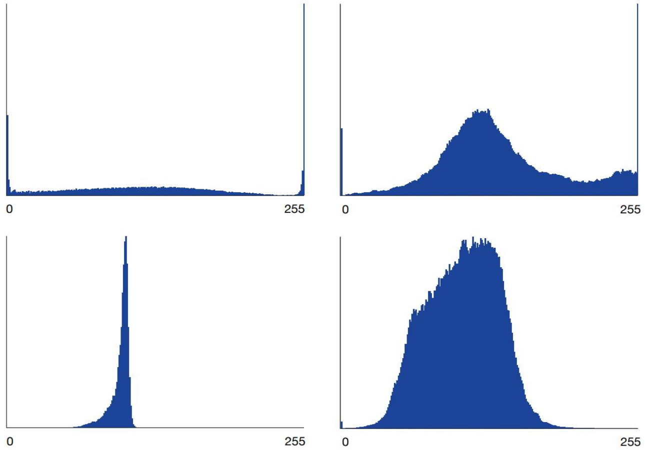


Bishop, Christopher M. 'Mixture density networks.' (1994).

RI.
SE

Non-Gaussian P(Y|X)

- Often the conditional distribution is not Gaussian
- If the real distribution is multimodal or skewed, the conditional mean will often be a poor prediction
- A mixture of Gaussian can theoretically model any continous distribution, but in practice can underperform



Four different conditional distributions on autoregressive tasks

Oord, Aaron van den, Nal Kalchbrenner, and Koray Kavukcuoglu. ‘Pixel recurrent neural networks.‘ arXiv preprint arXiv:1601.06759 (2016).

Discretizing trick

- Model continuous variables as dicrete
- Essentially predict histogram bins
- Determining *bins* (hinkar) becomes an issue
- One strategy is to look at the marginal emperical CDF of Y
- How many bins to use? Look at the data, what granularity makes sense?