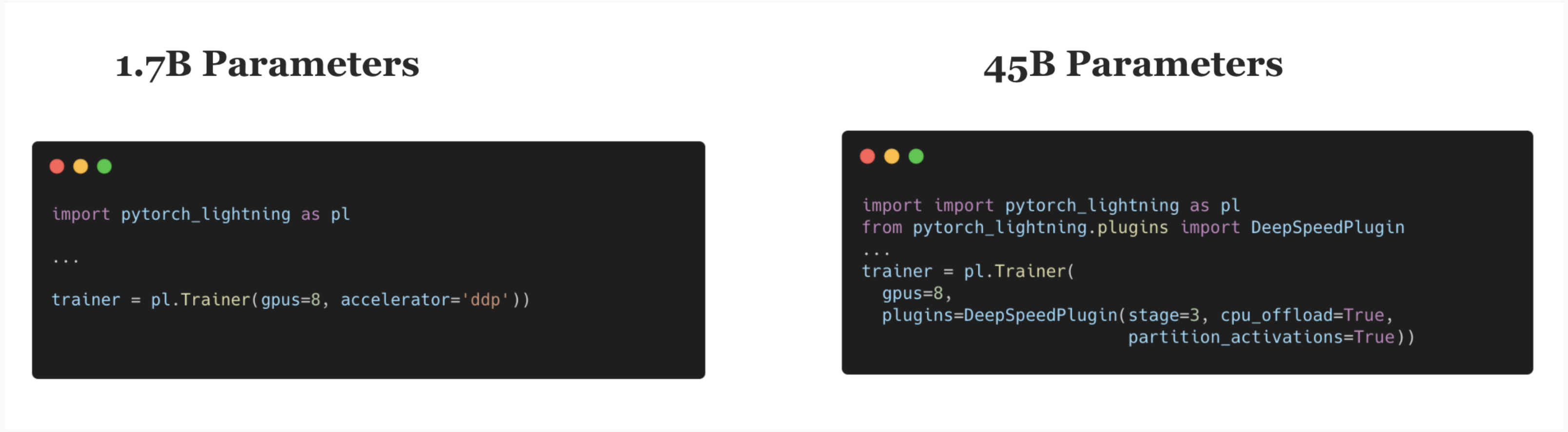
Accessible Multi-Billion Parameter Model Training with PyTorch Lightning + DeepSpeed

PyTorch Lightning team Mar 30 · 7 min read



```
import pytorch_lightning as pl

...

trainer = pl.Trainer(gpus=8, accelerator='ddp'))
```

```
import import pytorch_lightning as pl
from pytorch_lightning.plugins import DeepSpeedPlugin
...
trainer = pl.Trainer(
  gpus=8,
  plugins=DeepSpeedPlugin(stage=3, cpu_offload=True,
                          partition_activations=True))
```

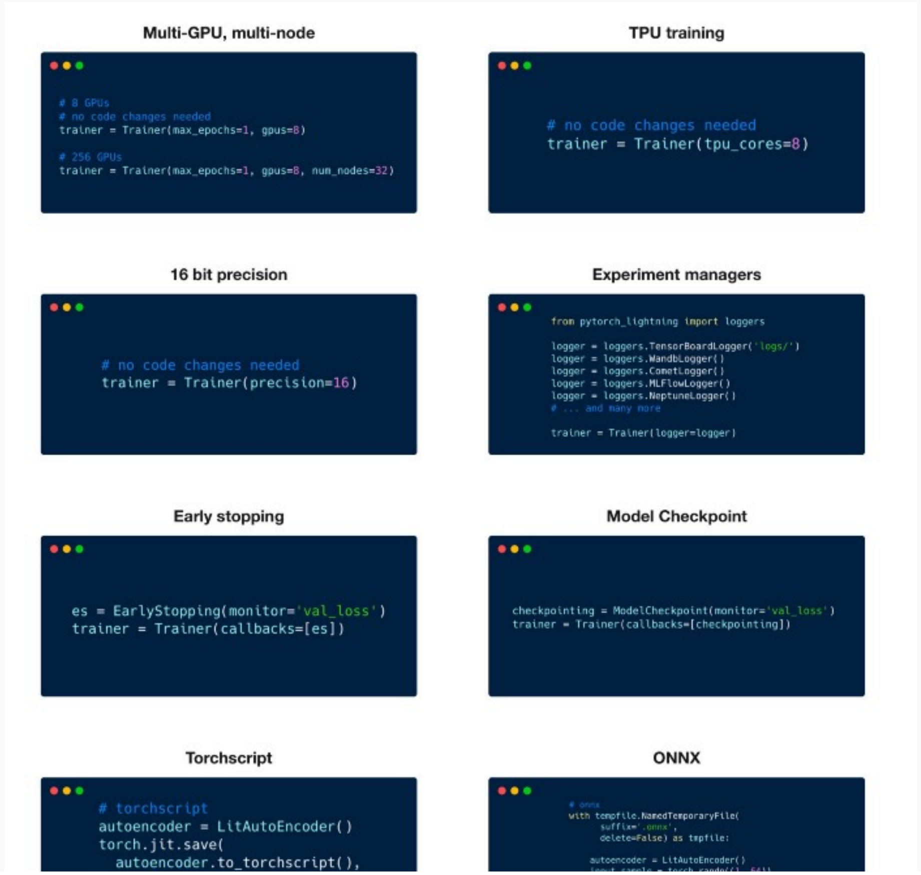
Max parameter size on using the same MinGPT model on the same lambda-labs A100 server with and without DeepSpeed with less than 3 lines of code difference

TLDR; This post introduces the PyTorch Lightning and DeepSpeed integration demonstrating how to scale models to billions of parameters with just a few lines of code.

What is PyTorch Lightning?

PDFmyURL

PyTorch Lighting is a lightweight PyTorch wrapper for high-performance AI research. PyTorch Lightning provides true flexibility by reducing the engineering boilerplate and resources required to implement state-of-the-art AI. Organizing PyTorch code with Lightning enables seamless training on multiple-GPUs, TPUs, CPUs and the use of difficult to implement best practices such as model sharding and even in 16-bit precision without changing your code.



Multi-GPU, multi-node

```python
# 8 GPUs
# no code changes needed
trainer = Trainer(max_epochs=1, gpus=8)

# 256 GPUs
trainer = Trainer(max_epochs=1, gpus=8, num_nodes=32)
```

TPU training

```python
# no code changes needed
trainer = Trainer(tpu_cores=8)
```

16 bit precision

```python
# no code changes needed
trainer = Trainer(precision=16)
```

Experiment managers

```python
from pytorch_lightning import loggers

logger = loggers.TensorBoardLogger('logs/')
logger = loggers.WandbLogger()
logger = loggers.CometLogger()
logger = loggers.MLFlowLogger()
logger = loggers.NeptuneLogger()
# ... and many more

trainer = Trainer(logger=logger)
```

Early stopping

```python
es = EarlyStopping(monitor='val_loss')
trainer = Trainer(callbacks=[es])
```

Model Checkpoint

```python
checkpointing = ModelCheckpoint(monitor='val_loss')
trainer = Trainer(callbacks=[checkpointing])
```

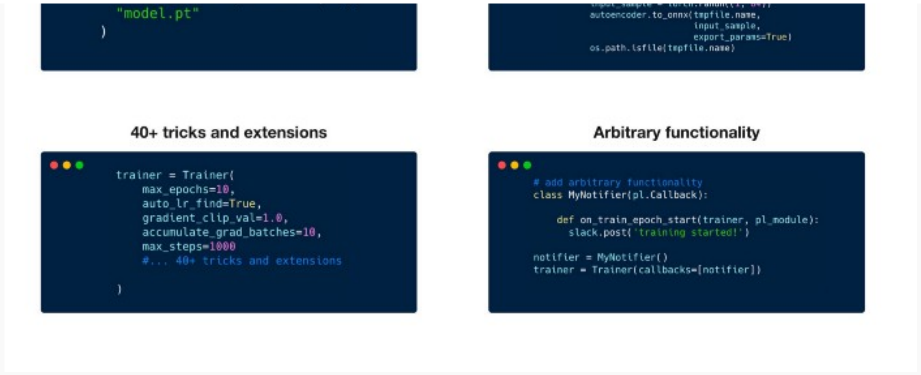
Torchscript

```python
# torchscript
autoencoder = LitAutoEncoder()
torch.jit.save(
  autoencoder.to_torchscript(),
```

ONNX

```python
# onnx
with tempfile.NamedTemporaryFile(
        suffix='.onnx',
        delete=False) as tmpfile:

    autoencoder = LitAutoEncoder()
    input_sample = torch.randn((1, 64))
```



```
        "model.pt"
    )
```

```
    autoencoder.to_onnx(tmpfile.name,
                        input_sample,
                        export_params=True)
    os.path.isfile(tmpfile.name)
```

40+ tricks and extensions

```
trainer = Trainer(
    max_epochs=10,
    auto_lr_find=True,
    gradient_clip_val=1.0,
    accumulate_grad_batches=10,
    max_steps=1000
    #... 40+ tricks and extensions

)
```

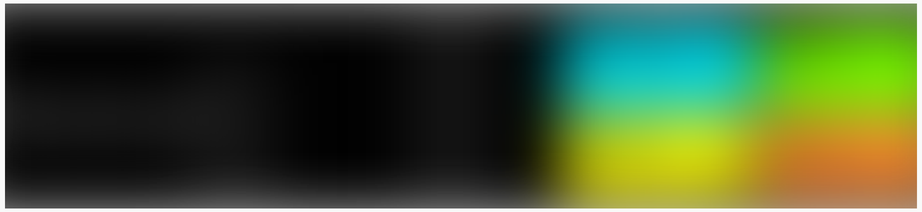
Arbitrary functionality

```
# add arbitrary functionality
class MyNotifier(pl.Callback):

    def on_train_epoch_start(trainer, pl_module):
      slack.post('training started!')

notifier = MyNotifier()
trainer = Trainer(callbacks=[notifier])
```

What is DeepSpeed?



DeepSpeed is a deep learning library on top of PyTorch that makes training models at extreme-scale efficient and easy for everyone. DeepSpeed offers powerful training features for data scientists training on massive supercomputers as well as those training on low-end clusters or even on a single GPU.

- Extreme model scale: DeepSpeed techniques like ZeRO and 3D parallelism can efficiently train multi-trillion parameter models on current GPU clusters with thousands of GPU devices.
- Extreme memory and communication efficiency: DeepSpeed democratizes multi-billion parameter training through efficient use of commodity memory and communication resources. A few examples include, (i) ZeRO-3 Offload which leverages CPU memory to efficiently train models with over 40B parameters on a single GPU, (ii) 1-bit Adam which reduces communication volume of the Adam

optimizer by up to 5X without sacrificing convergence efficiency, and (iii) 3D parallelism, which communicates more efficiently to train multi-billion parameter models up to 7x faster on clusters with limited network bandwidth.

- Extremely easy to use: Only few lines of code changes are required to integrate DeepSpeed into your model. Once integrated, any DeepSpeed optimization can be enabled by simply turning on the corresponding flag in your DeepSpeed Config file.

DeepSpeed is an important part of Microsoft’s new AI at Scale initiative to enable next-generation AI capabilities at scale, where you can find more information here.

How do I use DeepSpeed with PyTorch Lightning?

We first need to install DeepSpeed.

```
pip install deepspeed
```

After installing this dependency, PyTorch Lighting provides quick access to DeepSpeed through the Lightning Trainer. Below are a couple of code examples demonstrating how to take advantage of DeepSpeed in your Lightning applications without the boilerplate.

DeepSpeed ZeRO Stage 2 (Default)

DeepSpeed ZeRO Stage 1 is the first stage of parallelization optimization provided by DeepSpeed’s implementation of ZeRO. In addition to the standard parallelization of data it enables parallelization of the optimizer states reducing their memory overhead.

DeepSpeed Stage 2 is applied on top of Stage 1, partitions gradients in addition to data and optimizer states, further reducing memory footprint without impacting speed.

DeepSpeed with ZeRO Stage 3

DeepSpeed ZeRO Stage 3 applied on top of Stage 2, partitions model parameters in addition to gradients, data, and optimizer states, further reducing memory footprint. Since parameters are sharded across GPUs, DeepSpeed Stage 3 does cause additional communication overhead that impacts training speed. However, with the memory savings from Stage 3, the training speed can be more than recovered by increasing batch size. Most importantly, Stage 3 enables training of *multi-billion parameter* models.

DeepSpeed CPU Offload

CPU offload leverages the host CPU to further reduce overall memory consumption. Offloading to CPU means that there is more VRAM available on the GPU, enabling increased batch sizes and for even more throughput or larger models to be trained.

To use CPU Offloading, you need to substitute VRAM with normal RAM. This means you usually need a decent amount of space, i.e., for our benchmarks, we offloaded onto 1TB of RAM, and at the maximum size of 45 Billion parameters, we were using around 90% of our RAM.

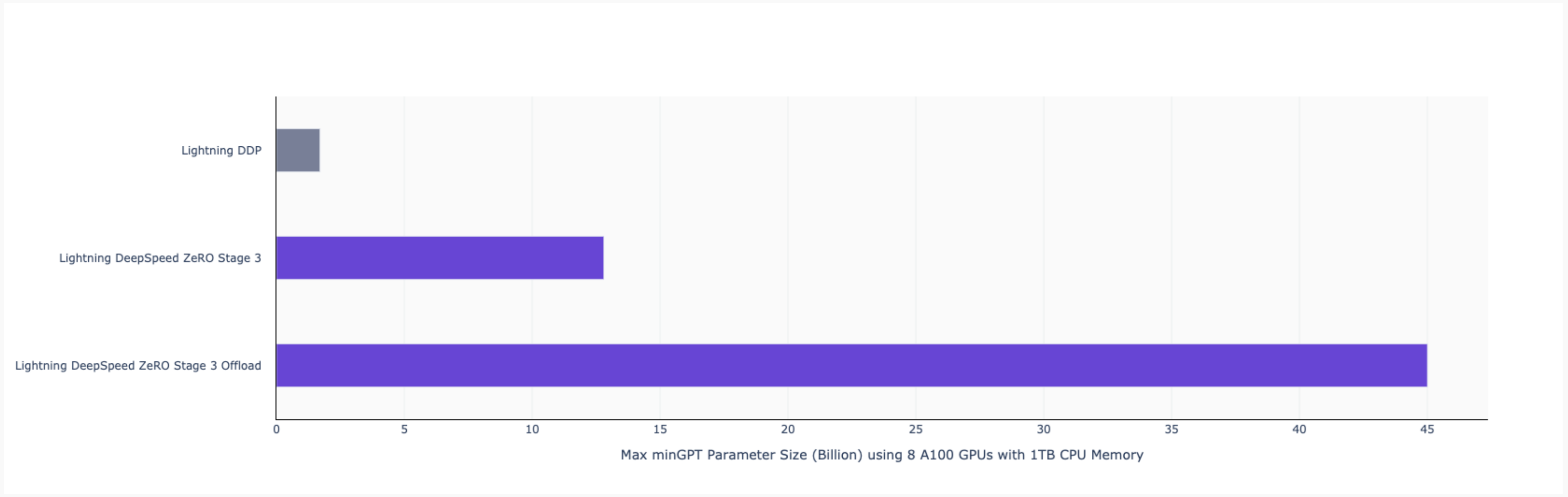
DeepSpeed with Activation Checkpointing

Activation checkpointing is a common technique used to reduce memory usage during training. With DeepSpeed Activation checkpointing, activations are not stored for the wrapped layer in the forward function and are re-calculated on the fly during the backwards pass.

```
x = deepspeed.checkpointing.checkpoint(block, x)
```

DeepSpeed also supports offloading activation checkpointsCPU for memory savings on the GPU, which can be used to increase the number of parameters or batch size. However, this requires more RAM. The Lightning DeepSpeed integration supports partitioning activations and checkpoints across GPUs while offloading activations to the CPU through the `partition_activations` and `cpu_checkpointing` flag respectively.

Benchmarks



Continue reading for more information.

To demonstrate the power of DeepSpeed as an accelerator, we use Lightning to benchmark a few large minGPT models based on Andrej Karpathy's work with different parameter sizes and DeepSpeed configurations.

Code for how we achieved these benchmarks can be found in the following.

SeanNaren/minGPT

Modified Andrej's and William's awesome code to create a simple github.com

Below we show the various and explain the various results of different configurations. The following benchmarks were run on a lambda-labs A100 server with 8 A100 GPUs with 1TB of RAM (which is used with CPU Offloading).

DDP

DDP is the traditional accelerator baseline for distributed PyTorch Lightning workloads; for these benchmarks, we use it as a control. The largest model that fits is **1.7B parameters.**

```
python benchmark.py --n_layer 15 --n_head 16 --n_embd 3072 --gpus 8 -
-precision 16 --limit_train_batches 128 --batch_size 1


# Average Epoch time: 43.69 seconds
# Average Peak memory 36148.55MiB
```

DeepSpeed ZeRO Stage 3

Without CPU Offloading, the largest model that fits is **12.8B parameters.**

```
python benchmark.py --n_layer 16 --n_head 16 --n_embd 8192 --gpus 8 -
-precision 16


# Average Epoch time: 45.05 seconds
# Average Peak memory 7796.98MiB
```

DeepSpeed ZeRO Stage 3 + CPU Offload

Using CPU Offloading and DeepSpeed activation checkpointing, the largest model that fits is **45B parameters.**

```
python benchmark.py --n_layer 56 --n_head 16 --n_embd 8192 --gpus 8 -
-precision 16  --cpu_offload

# Average Epoch time: 248.39 seconds
# Average Peak memory 4976.98MiB
```

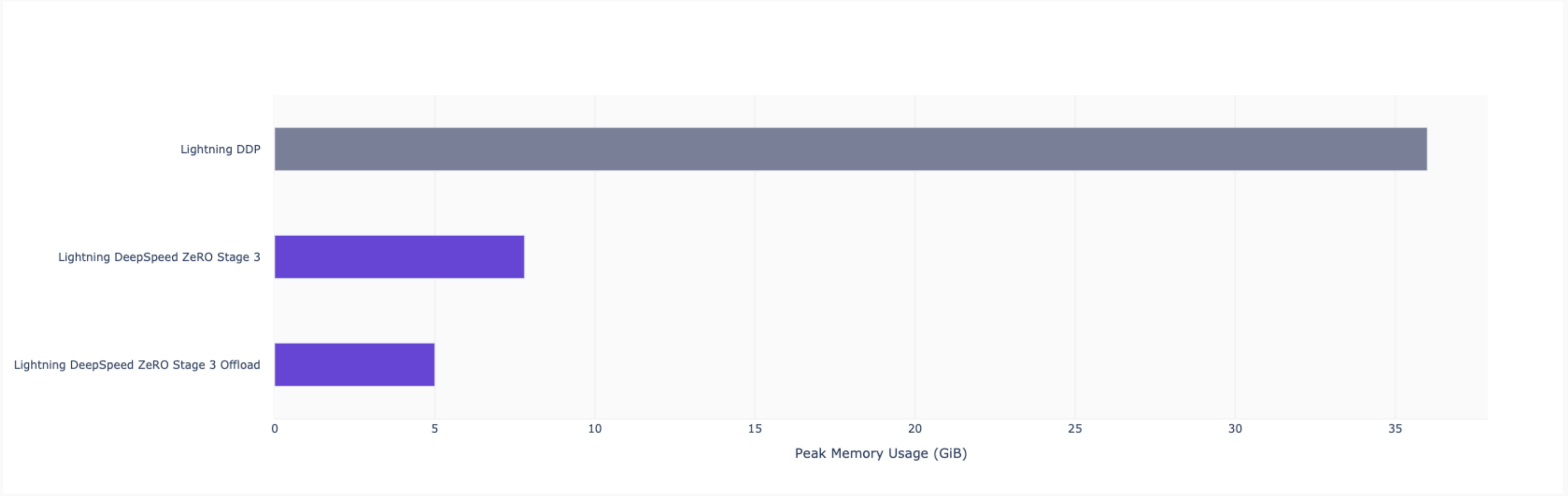
DeepSpeed ZeRO Stage 3 + CPU Offload + Activation Checkpointing

```
python benchmark.py --n_layer 56 --n_head 16 --n_embd 8192 --gpus 8 -
-precision 16  --cpu_offload

# Average Epoch time: 256.91 seconds
# Average Peak memory 2192.98MiB
```

Note: we wrapped our layers using deepspeed.checkpointing.checkpoint for this experiment.

Low GPU Memory Environments



Comparison of peak memory when using minGPT. CPU Offloading reduces the memory requirements substantially however trades off speed. With the same batch size of 1, we see a 6x speed reduction, however, given the reduction in memory usage, we can increase our batch size by a factor of x7, which will allow CPU offloading to remain at parity with DDP"

Below are benchmarks of minGPT with the largest DDP model that fits in memory. We assume a batch size of 1, as the user is trying to train a large model but may not have high memory GPUs such as A100s available to them. Consider using a G4 instance on AWS for example, or something smaller.

Command:

```
python benchmark.py --n_layer 15 --n_head 16 --n_embd 3072 --gpus 8 -
-precision 16 --limit_train_batches 128 --batch_size 1

# Average Epoch time: 256.91 seconds
# Average Peak memory 2192.98MiB
```

DeepSpeed PyTorch Lightning Learnings

- Make sure to use the DeepSpeed optimizers such as DeepSpeedCPUAdam, when using CPU Offloading, rather than the default torch optimizers to reduce CPU performance bottlenecks.
- Treat your GPU/CPU memory as one large pool. In some cases, you may not want to offload certain things (like activations) to provide even more space to offload model parameters
- While ZeRO Stage 3 CPU Offload is slower is due to the model being very small and the batch size being 1 since the peak memory is extremely small, enabling us to increase the batch size by massive factors and end up with significantly faster training times than we would without this trick.
- The massive GPU memory savings from ZeRO 3 Offload can support very large batch sizes even with gigantic models making it really fast despite the communication overhead.

Next Steps

Now that you have a sense of how to use DeepSpeed and Lightning, we look forward to seeing what models you can build.

Microsoft/DeepSpeed

DeepSpeed is a deep learning optimization library that makes distributed training easy, efficient, and effective. 10x...

github.com

PyTorchLightning/pytorch-lightning

Lightning disentangles PyTorch code to decouple science from engineering. Lightning structures PyTorch code...

github.com

If you have any questions or want to show off your own projects, feel free to comment below and be sure to keep an eye out on both the DeepSpeed and Lightning Repos for new features that are being actively developed all the time.

Thanks to イルカ Borovec.

166 3

Pytorch Pytorch Lightning Deep Learning Deepspeed AI

More from PyTorch Lightning Developer Blog

Follow

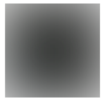
PyTorch Lightning is a lightweight machine learning framework that handles most of the engineering work, leaving you to focus on the science. Check it out: pytorchlightning.ai

Read more from PyTorch Lightning Developer Blog

More From Medium

TorchMetrics v0.3.0 — Information Retrieval metrics and more

PyTorch Lightning team in PyTorch Lightning Developer Blog

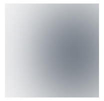


3 PyTorch Lightning Community Causal Inference Examples To Inspire Your Next Project!

Aaron (Ari) Bornstein in PyTorch Lightning Developer Blog

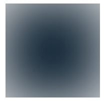
TorchMetrics — PyTorch Metrics built to scale

PyTorch Lightning team in PyTorch Lightning Developer Blog



Training Transformers at Scale With PyTorch Lightning

PyTorch Lightning team in PyTorch Lightning Developer Blog



How to choose a CI framework for deep learning

イルカ Borovec in PyTorch Lightning Developer Blog

3 PyTorch Lightning Winning Community Kernels to Inspire your Next Kaggle Victory

Aaron (Ari) Bornstein in PyTorch Lightning Developer Blog

4 PyTorch Lightning Community NLP Examples To Inspire Your Next Project!

Aaron (Ari) Bornstein in PyTorch Lightning Developer Blog

4 PyTorch Lightning Community Computer Vision Examples To Inspire Your Next Project!

Aaron (Ari) Bornstein in PyTorch Lightning Developer Blog

Medium

About Help Legal

PDFmyURL.com - convert URLs, web pages or even full websites to PDF online. Easy API for developers!