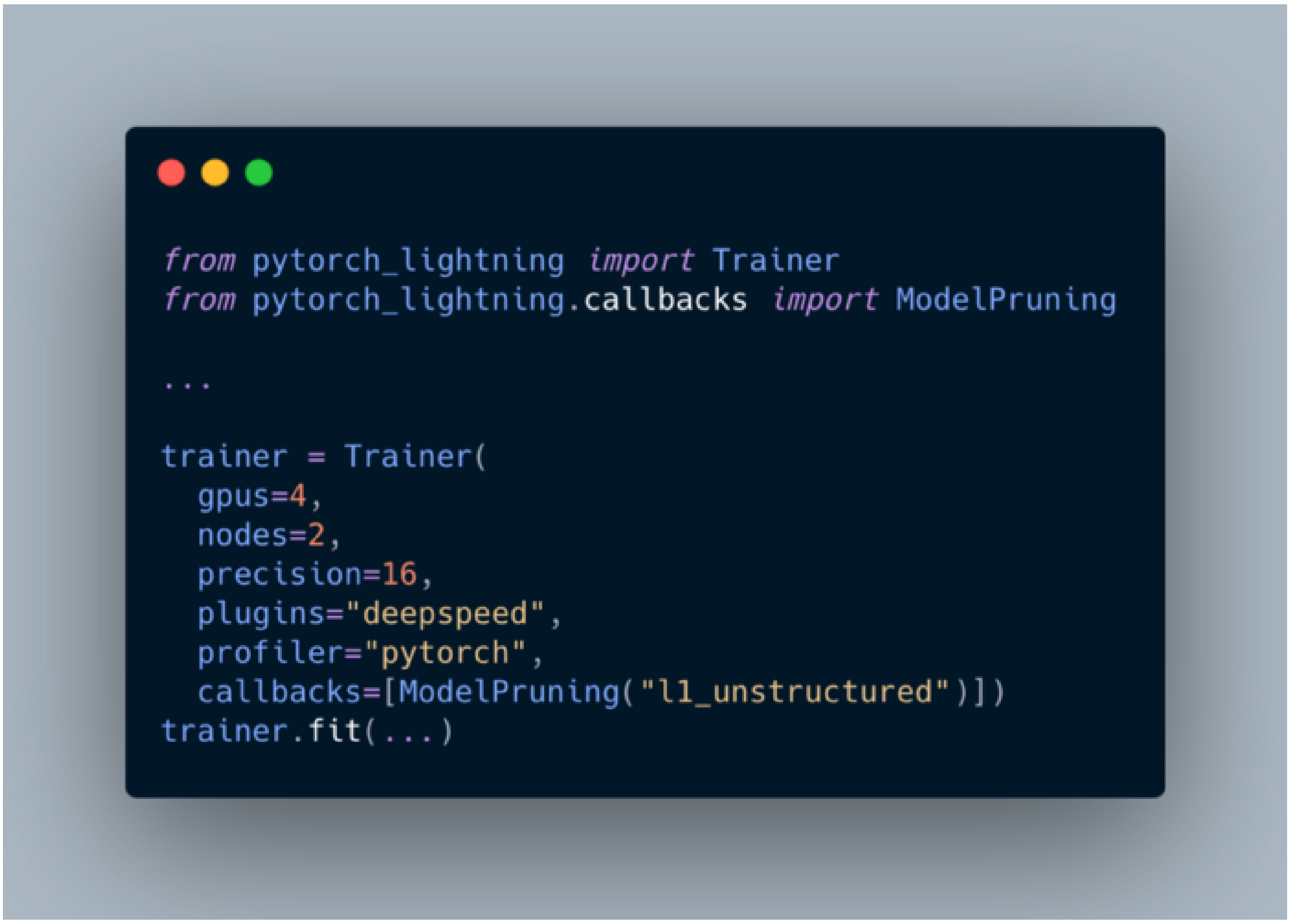
PyTorch Lightning V1.2.0- DeepSpeed, Pruning, Quantization, SWA

New release including many new PyTorch integrations, DeepSpeed model parallelism, and more.

PyTorch Lightning team
Feb 19 · 5 min read



We are happy to announce PyTorch Lightning V1.2.0 is now publicly available. It is packed with new integrations for anticipated features such as:

- PyTorch autograd profiler
- DeepSpeed model parallelism

- Pruning
- quantization
- Stochastic weights averaging
- + more stability improvements

Continue reading to learn more about what's available. As always, feel free to reach out on Slack or discussions for any questions you might have or issues you are facing.

PyTorch Profiler [BETA]

PyTorch `Autograd` provides a profiler that lets you inspect the cost of different operations

inside your model — both on the CPU and GPU (read more about the profiler in the PyTorch documentation). You can now enable the PyTorch profiler in Lightning out of the box:

```
trainer = Trainer(profiler="pytorch")
```

profiler.py hosted with ❤ by GitHub view raw

Or initialize the profiler for further customization:

```
from pytorch_lightning.profiler.profilers import PyTorchProfiler

profiler = PyTorchProfiler(output_filename="profiler.txt", ...)
trainer = Trainer(..., profiler=profiler)
```

profiler.py hosted with ❤ by GitHub view raw

Example report:

```
Profiler Report

    Profile stats for: training_step_and_backward
    ---------------------  ---------------  ---------------  -------
--------  ---------------  ---------------
    Name                   Self CPU total %  Self CPU total   CPU
total %      CPU total        CPU time avg
    ---------------------  ---------------  ---------------  -------
--------  ---------------  ---------------
    t                      62.10%           1.044ms          62.77%
```

```
1.055ms          1.055ms
    addmm                    32.32%           543.135us        32.69%
549.362us        549.362us
    mse_loss                 1.35%            22.657us         3.58%
60.105us         60.105us
    mean                     0.22%            3.694us          2.05%
34.523us         34.523us
    div_                     0.64%            10.756us         1.90%
32.001us         16.000us
    ones_like                0.21%            3.461us          0.81%
13.669us         13.669us
    sum_out                  0.45%            7.638us          0.74%
12.432us         12.432us
    transpose                0.23%            3.786us          0.68%
11.393us         11.393us
    as_strided               0.60%            10.060us         0.60%
10.060us         3.353us
    to                       0.18%            3.059us          0.44%
7.464us          7.464us
    empty_like               0.14%            2.387us          0.41%
6.859us          6.859us
    empty_strided            0.38%            6.351us          0.38%
6.351us          3.175us
    fill_                    0.28%            4.782us          0.33%
5.566us          2.783us
    expand                   0.20%            3.336us          0.28%
4.743us          4.743us
    empty                    0.27%            4.456us          0.27%
4.456us          2.228us
    copy_                    0.15%            2.526us          0.15%
2.526us          2.526us
    broadcast_tensors        0.15%            2.492us          0.15%
2.492us          2.492us
    size                     0.06%            0.967us          0.06%
0.967us          0.484us
    is_complex               0.06%            0.961us          0.06%
0.961us          0.481us
    stride                   0.03%            0.517us          0.03%
0.517us          0.517us
    --------------------  --------------  --------------  -------
--------  --------------  --------------
    Self CPU time total: 1.681ms
```

Learn about all the Lightning supported profilers here.

DeepSpeed Plugin [BETA]

DeepSpeed offers additional CUDA Deep Learning training optimizations, to train massive billion-parameter models. DeepSpeed offers lower-level training optimizations such as ZeRO-Offload, and useful memory/speed efficient optimizers such as 1-bit Adam. We've recorded 10+ Billion Parameter models using our default training configuration on multiple GPUs, with follow-up technical details coming soon.

To enable DeepSpeed in Lightning 1.2 simply pass in `plugins='deepspeed'` to your Lightning trainer (docs).

```
trainer = Trainer(gpus=4, plugins='deepspeed', precision=16)
```

deepspeed.py hosted with ❤ by GitHub view raw

Learn more about DeepSpeed implementation with technical publications here.

Pruning [BETA]

Pruning is a technique to optimize model memory, hardware, and energy requirements by eliminating some of the model weights. Pruning is able to achieve significant model efficiency improvements while minimizing the drop in task performance. The pruned model is smaller in size and faster to run.

To enable pruning during training in Lightning 1.2, simply pass in the ModelPruning callback to the Lighting Trainer (using torch pruning under the hood).

This callback supports multiple pruning functions (pass any torch.nn.utils.prune function as a string to select which weights to pruned), setting pruning percentage, performing iterative pruning, and applying the lottery ticket hypothesis, and more (docs).

```
from pytorch_lightning.callbacks import ModelPruning

trainer = Trainer(callbacks=[ModelPruning("l1_unstructured")])
```

pruning.py hosted with ❤ by GitHub view raw

Quantization [BETA]

Model quantization is another performance optimization technique that allows speeding up inference and decreasing memory requirements by performing computations and storing tensors at lower bitwidths (such as INT8 or FLOAT16) than floating-point precision. Quantization not only reduces the model size but also speeds up loading since operations on fixpoint are faster than on floating-point.

Quantization Aware Training (QAT) mimics the effects of quantization during training: all computations are carried out in floating points while training, simulating the effects of ints, and weights and activations are quantized into lower precision only once training is completed.

Lightning 1.2 includes Quantization Aware Training callback (using PyTorch native quantization, read more here), which allows creating fully quantized models (compatible with torchscript).

```python
from pytorch_lightning.callbacks import QuantizationAwareTraining

class RegressionModel(LightningModule):

    def __init__(self):
        super().__init__()
        self.layer_0 = nn.Linear(16, 64)
        self.layer_0a = torch.nn.ReLU()
        self.layer_1 = nn.Linear(64, 64)
        self.layer_1a = torch.nn.ReLU()
        self.layer_end = nn.Linear(64, 1)

                def forward(self, x):
        x = self.layer_0(x)
        x = self.layer_0a(x)
        x = self.layer_1(x)
        x = self.layer_1a(x)
        x = self.layer_end(x)
        return x

qcb = QuantizationAwareTraining(
                # specification of quant estimation quaity
                observer_type='histogram',
                # specify which layers shall be merged together to increase efficiency
                modules_to_fuse=[(f'layer_{i}', f'layer_{i}a') for i in range(2)]
                # make the model torchanble
                input_compatible=False,
)

trainer = Trainer(callbacks=[qcb])
qmodel = RegressionModel()
trainer.fit(qmodel, ...)

# take sample data batch, for example from you test dataloader
batch = iter(my_dataloader()).next()
# using fully quantized model, you need to apply quantization layer
qmodel(qmodel.quant(batch[0]))

# converting model to torchscript
tsmodel = qmodel.to_torchscript()
# even converted model preserve created quantisation layer which you can/should use
tsmodel(tsmodel.quant(batch[0]))
```

You can further customize the callback:

```
qcb = QuantizationAwareTraining(
    # specification of quant estimation quality
    observer_type='histogram',
    # specify which layers shall be merged together to increase efficiency
    modules_to_fuse=[(f'layer_{i}', f'layer_{i}a') for i in range(2)]
    # make your model compatible with all original input/outputs, in such case the mode
    input_compatible=True
)

batch = iter(my_dataloader()).next()
qmodel(batch[0])
```

quant2.py hosted with ❤ by GitHub view raw

Read the docs here.

Stochastic Weight Averaging [BETA]

Stochastic Weight Averaging (SWA) can make your models generalize better at virtually no additional cost. This can be used with both non-trained and trained models. The SWA procedure smooths the loss landscape thus making it harder to end up in a local minimum during optimization.

Lightning 1.2 supports SWA (using PyTorch native implementation), with a simple trainer flag (available with PyTorch version 1.6 and higher)

```
trainer = Trainer(stochastic_weight_avg=True)
```

swa.py hosted with ❤ by GitHub view raw

Or for further customization use the StochasticWeightAveraging callback:

```
from pytorch_lightning.callbacks import StochasticWeightAveraging

trainer = Trainer(callbacks=[StochasticWeightAveraging()])
```

swa-callback.py hosted with ❤ by GitHub view raw

Read the docs here.

Finetuning [BETA]

Finetuning (or transfer-learning) is the process of tweaking a model trained on a large dataset, to a particular (likely much smaller) dataset. For more details on finetuning, see this Flash notebook.

To make finetuning simpler with Lightning, we are introducing BackboneFinetuning callback you can customize for your own use case, or create your own callback, subclassing BaseFinetuning:

```
from pytorch_lightning.callbacks import BaseFinetuning

class MyBackboneFinetuning(BaseFinetuning):

    def __init__(self, unfreeze_backbone_at_epoch: int = 5, train_bn: bool = True, back
        self._unfreeze_backbone_at_epoch = unfreeze_backbone_at_epoch
        self._train_bn = train_bn
        self._backbone_lr = backbone_lr

    def freeze_before_training(self, pl_module: LightningModule):
        self.freeze(pl_module.backbone, train_bn=self._train_bn)

    def finetune_function(self, pl_module: LightningModule, epoch: int, optimizer: Opt:
        """Called on every epoch starts."""
        if epoch == self.unfreeze_backbone_at_epoch:
            self.unfreeze_and_add_param_group(
                pl_module.backbone,
                optimizer,
                lr=self._backbone_lr,
                train_bn=self.train_bn,
            )

trainer = Trainer(callbacks=[MyBackboneFinetuning()])
```

finetune.py hosted with ❤ by **GitHub** **view raw**

PyTorch Geometric integration

PyTorch Geometric (PyG) is a popular deep learning geometric extension library for PyTorch (see Fast Graph Representation Learning with PyTorch Geometric by Matthias Fey and Jan E. Lenssen). Currently, PyG provides over 60+ SOTA models and methods for Graph Convolution. You can now train PyG models with the Lightning Trainer! See examples here.

```
datamodule = Reddit('data/Reddit')
```

```
datamodule = Reddit('data/Reddit')
model = GraphSAGE(datamodule.num_features, datamodule.num_classes)

trainer = Trainer(gpus=2, accelerator='ddp', max_epochs=10)
trainer.fit(model, datamodule=datamodule)
```

geometric.py hosted with ❤ by **GitHub** **view raw**

New Accelerator/plugins API

Training Deep Learning Models at scale while retaining full flexibility requires a lot of orchestration between different responsibilities. The new API isolates responsibilities by introducing a new accelerator API as well as new types of Plugins: one for different training types (like a single device, DDP, …) and one to handle different floating-point precisions during training. Having dedicated interfaces reduces code duplication and enhances usability (for power users).

The `Trainer` interface did not change for most use-cases but was extended to allow further customization through plugins.

Simple accelerator use:

Or pass in a plugin for customization:

For help migrating custom plugins to the new API reach out to us on slack or via support@pytorchlightning.ai.

Special shout out to Justus Schock and Adrian Wälchli for all the hard work!

Other improvements

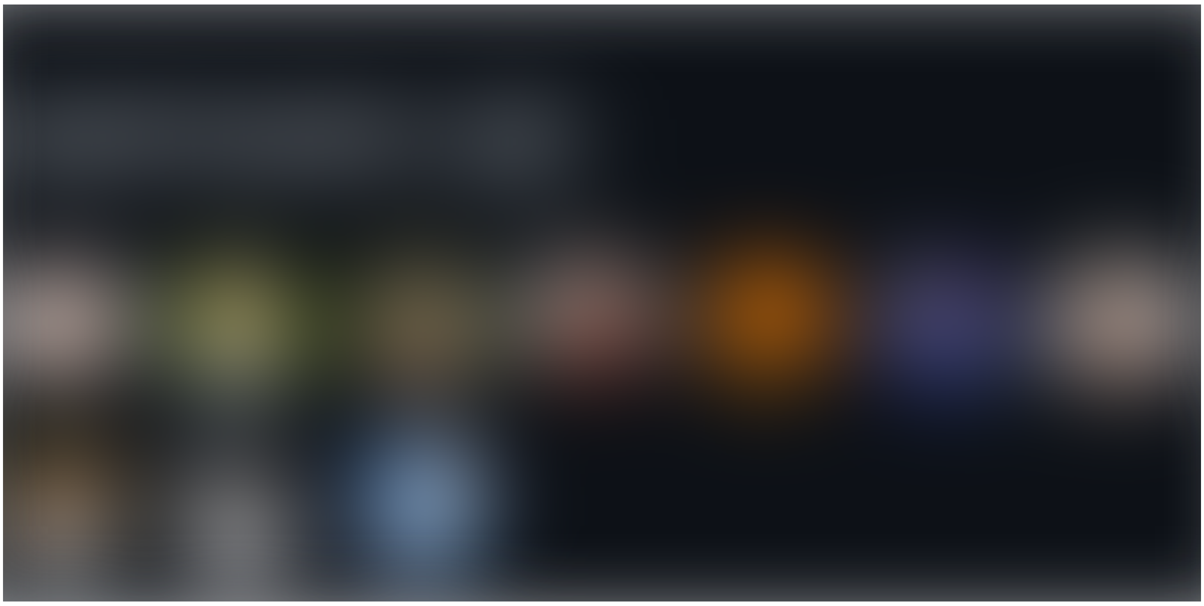
- Added support for multiple train loaders
- adding `trainer.predict` for simple inference with Lightning
- New metrics: `HammingDistance`, `StatScores`, `R2Score`
- Added `LightningModule.configure_callbacks` to enable the definition of model-specific callbacks
- Enabled `self.log` in callbacks

- Changed the seq of `on_train_batch_end`, `on_batch_end` & `on_train_epoch_end`, `on_epoch_end` hooks

See all changes in the release notes.

Thank you!

Big kudos to all the community members for their contributions and feedback. We now have over 400 Lightning contributors! Want to give open source a try and get free Lightning swag? We have a #new_contributors channel on slack. Check it out!



Pytorch Pytorch Lightning