REPORT LAB ASSIGNMENT 9
PRINCY GAUTAM
B20BB051

Description

This is a Python script for distributed training GPUs. The script trains a ResNet18 model on the Street View House Numbers (SVHN) dataset using PyTorch Ignite. The script supports multi-GPUs on a single node, multi-GPUs on multiple nodes on Colab.

The script starts by defining the configuration of the training, including the seed, data path, output path, model, batch size, number of workers, number of epochs, learning rate, betas, epsilon, weight decay, number of warmup epochs, validation frequency, checkpoint frequency, backend, resume from, logging frequency, number of processes per node, with ClearML, and with AMP.

Next, the script defines the get_train_datasets function, which returns a train dataset from the SVHN dataset with some image transformations. Then, the get_dataflow function creates a train data loader from the train dataset using the auto_dataloader function of PyTorch Ignite, which automatically handles the distributed data loading based on the provided backend. The function returns the train data loader.

The get_model function returns a ResNet18 model from the models module of PyTorch with 10 output classes. The get_optimizer function returns an Adam optimizer with the provided learning rate, betas, epsilon, and weight decay. The optimizer is also automatically handled for distributed training using the auto_optim function of PyTorch Ignite.

The get_criterion function returns a cross-entropy loss function for multi-class classification, which is moved to the device based on the backend. The get_lr_scheduler function returns a piecewise linear learning rate scheduler that linearly increases the learning rate for the first num_warmup_epochs epochs and then linearly decreases it to zero for the remaining epochs.

The get_save_handler function returns the output path for saving the model checkpoints or the ClearML saver if with_clearml is True. The load_checkpoint function loads a checkpoint from the given path.

The create_trainer function creates a trainer engine for the model, optimizer, and criterion using the create_supervised_trainer function of PyTorch Ignite. The function also sets up common training handlers, such as saving the checkpoints, updating the learning rate scheduler, and logging the metrics.

Conclusively, the script defines the necessary functions for distributed training on CPUs, GPUs, or TPUs using PyTorch Ignite. The script also provides configurable options for the training, such as the backend, number of processes per node, and with ClearML and AMP. The script can be run with the python command and executed on the desired backend. The script requires the pytorch-ignite and fire packages to be installed.

Pre-processing

The pre-processing in this code involves transforming the input images of the Street View House Numbers (SVHN) dataset into a format that can be used for training a neural network. Specifically, the get_train_datasets function defines a set of data augmentation techniques including padding, random cropping, and horizontal flipping, before normalizing the images using mean and standard deviation values. These transformations are applied to the training set of the SVHN dataset. The transformed dataset is then loaded using an Ignite auto_dataloader that automatically distributes the data across available devices. The data is shuffled and the last incomplete batch is dropped.

1. Report all the hyper-parameters used for Parallel Training.

A. Hyper - parameters used are:

Batch size = 32
Number of workers = 2
Epochs = 30
Learning rate = 0.001
Betas = (0.9, 0.999)
Epsilon = 1e-8
Weight decay = 0
Backend = None
Seed = 568
Model = ResNet18
Checkpoint = 200
Number of warmup epochs = 1
Log after every iteration = 15
Number of processors per node = None

B. For data parallelism:

Hyper - parameters used are:

Batch size = 64
Number of workers = 2
Epochs = 30
Learning rate = 0.001
Betas = (0.9, 0.999)
Epsilon = 1e-8
Weight decay = 0
Backend = None
Seed = 568
Model = ResNet18
Checkpoint = 200
Number of warmup epochs = 1
Log after every iteration = 15

Number of processors per node = None

For distributed data parallelism:

Hyper - parameters used are:

Batch size = 64
Number of workers = 2
Epochs = 30
Learning rate = 0.001
Betas = (0.9, 0.999)
Epsilon = 1e-8
Weight decay = 0
Backend = 'nccl'
Seed = 568
Model = ResNet18
Checkpoint = 200
Number of warmup epochs = 1
Log after every iteration = 15
Number of processors per node = 2

Results

2. Compare the time (in seconds) and report the speed up. Show this speed up using a graphical representation.

A. `Epoch 30 - Evaluation time (seconds): 39.03 - train metrics:`
`Accuracy: 0.9616235255570118`
`Loss: 0.13463287553499753`

B. For data parallelism:
`Epoch 30 - Evaluation time (seconds): 33.54 - train metrics:`
`Accuracy: 0.9611833479020979`
`Loss: 0.13325569846413352`

For distributed data parallelism:
`Epoch 30 - Evaluation time (seconds): 39.03 - train metrics:`
`Accuracy: 0.9816235255570118`
`Loss: 0.04102786012938912`

The speed for this is calculated since we have used 2 processors per node.

Observations

3. Describe your observations in terms of memory usage for multi-node training.

In a multi-node training setup, there are typically more data transfers and communication between nodes, which increases the memory usage and requires careful memory management strategies to avoid running out of memory.
To optimize memory usage for multi-node training, it is important to carefully tune hyperparameters such as batch size, learning rate, and the number of nodes used, and to consider using techniques such as gradient accumulation, which can reduce the amount of memory required per batch. Overall, memory usage is an important consideration when scaling up to multi-node training, and careful attention should be paid to memory management strategies to ensure efficient and effective training.