How to scale training on multiple GPUs

How to train a PyTorch model in multiple GPUs

Giuliano Giacaglia · Follow

Published in Towards Data Science · 9 min read · Dec 21, 2019

One of the biggest problems with Deep Learning models is that they are becoming too big to train in a single GPU. If the current models were trained in a single GPU, they would take too long. In order to train models in a timely fashion, it is necessary to train them with multiple GPUs.

We need to scale training methods to use 100s of GPUs or even 1000s of GPUs. For example, a famous researcher was able to reduce the ImageNet training time from 2 weeks to 18 minutes, or train the largest and the state of the art Transformer-XL in 2 weeks instead of 4 years. He used 100s of GPUs to do that.

We care deeply about our training iteration speeds. So in order to increase our iteration speeds, we've had to scale up our training to multiple GPUs. In this blog post, I will go over how to scale up training with PyTorch. We've had some models in TensorFlow (<2.0) and scaled our training, using Horovod, a tool created by Uber Engineering team. If you go down that path, we recommend installing it using their Docker Image.

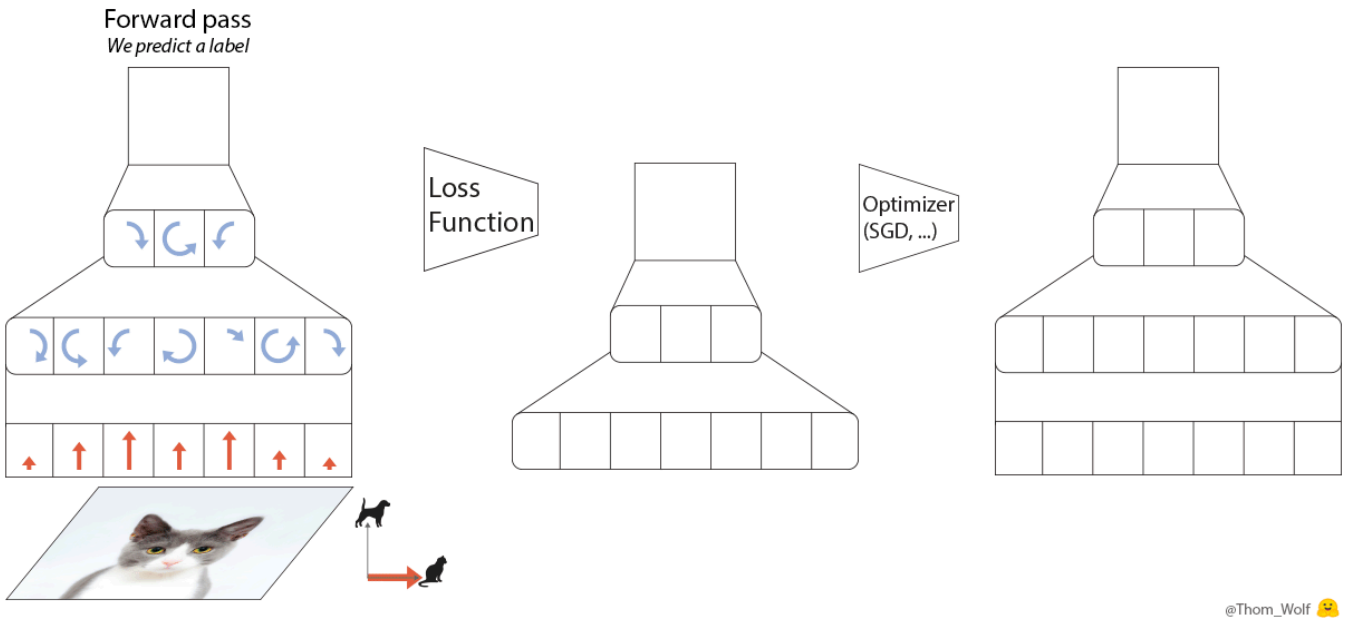
We find that PyTorch has the best balance between ease of use and control, without giving up performance. PyTorch built two ways to implement distribute training in multiple GPUs: `nn.DataParalllel` and `nn.DistributedParalllel`. They are simple ways of wrapping and changing your code and adding the capability of training the network in multiple GPUs.

`nn.DataParallel` is easier to use, but it requires its usage in only one machine. `nn.DataParalllel` only uses one process to compute model weights and distribute them to each GPU during each batch.

In this blog post, I will go into detail how `nn.DataParallel` and `nn.DistributedDataParalllel` work. I will cover the main differences between the two, and how training in multiple GPUs works. I will first explain how the training a neural network works.

Training loop

First, let's go over how training a neural network usually works. For this we will use some images created by HuggingFace:



There are four main steps for each loop that happens when training a neural network:

1. The forward pass, where the input is processed by the neural network
2. The loss function is calculated, comparing the predicted label with the ground-truth label
3. The backward pass is done, calculating the gradients for each parameter based on the loss (using back-propagation)

4. The parameters are updated using the gradients

For batch sizes greater than one, we might want to batch normalize the training. For an in-depth explanation of batch normalization I recommend going over this blog post:

Understanding the backward pass through Batch Normalization Layer

At the moment there is a wonderful course running at Standford University, called CS231n - Convolutional Neural...

kratzert.github.io

DataParallel

DataParallel helps distribute training into multiple GPUs in a single machine. Let's go into detail how DataParallel works. There are a few steps that happen whenever training a neural network using DataParallel:

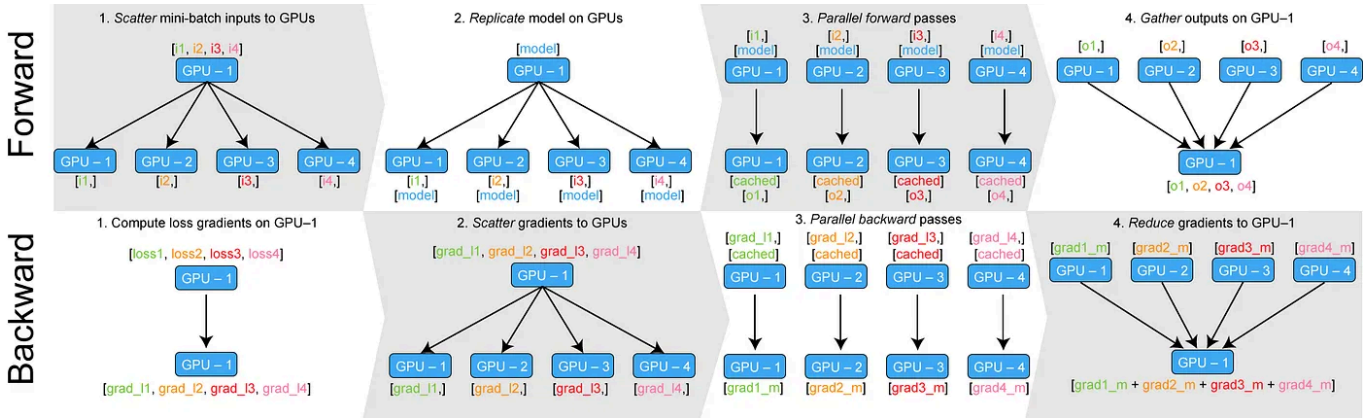


Image created by HuggingFace

1. The mini-batch is split on GPU:0

2. Split and move min-batch to all different GPUs

3. Copy model out to GPUs
4. Forward pass occurs in all different GPUs
5. Compute loss with regards to the network outputs on GPU:0, and return losses to the different GPUs. Calculate gradients on each GPU
6. Sum up gradients on GPU:0 and use the optimizer to update model on GPU:0

A Simple example

Let's code this up. First, let's import everything we need

```
import os
from datetime import datetime
import argparse
import torch.multiprocessing as mp
import torchvision
import torchvision.transforms as transforms
import torch
import torch.nn as nn
import torch.distributed as dist
from apex.parallel import DistributedDataParallel as DDP
from apex import amp
```

imports_train.py hosted with ❤️ by GitHub view raw

We define a very simple convolutional model for predicting MNIST

```python
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc = nn.Linear(7*7*32, num_classes)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        return out
```

mnist.py hosted with ❤️ by GitHub view raw

Line 4–14: We are defining the layers in this neural network.

Line 16–21: We define the forward pass

The main() function will take in some arguments and run the training function:

```
def train(gpu, args):
        torch.manual_seed(0)
    model = ConvNet()
    model = nn.DataParallel(model)
    torch.cuda.set_device(gpu)
    model.cuda(gpu)
    batch_size = 100
    # define loss function (criterion) and optimizer
    criterion = nn.CrossEntropyLoss().cuda(gpu)
    optimizer = torch.optim.SGD(model.parameters(), 1e-4)
    # Data loading code
    train_dataset = torchvision.datasets.MNIST(root='./data',
                                               train=True,
                                               transform=transforms.ToTensor(),
                                               download=True)
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                               batch_size=batch_size,
                                               shuffle=True,
                                               num_workers=0,
                                               pin_memory=True)

    start = datetime.now()
    total_step = len(train_loader)
    for epoch in range(args.epochs):
        for i, (images, labels) in enumerate(train_loader):
            images = images.cuda(non_blocking=True)
            labels = labels.cuda(non_blocking=True)
            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            if (i + 1) % 100 == 0 and gpu == 0:
                print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(
                    epoch + 1,
                    args.epochs,
                    i + 1,
                    total_step,
                    loss.item())
                   )
    if gpu == 0:
        print("Training complete in: " + str(datetime.now() - start))
```

train_mnist.py hosted with ❤️ by GitHub view raw

Line 2–6: We instantiate the model and set it to run in the specified GPU, and run our operations in multiple GPUs in parallel by using `DataParallel`.

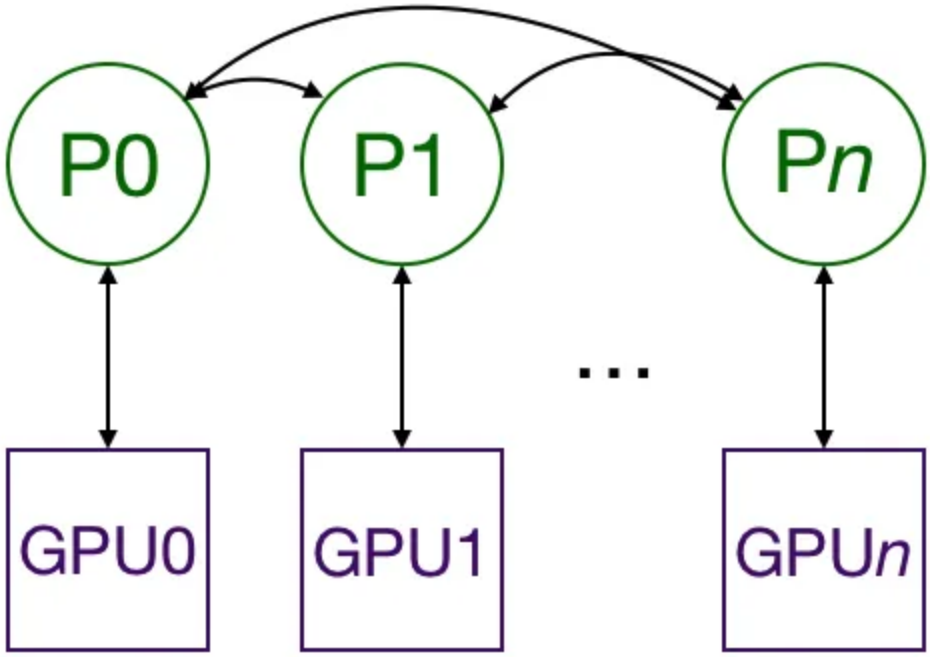
Line 9–23: We define the loss function (criterion), and the optimizer (in this case we are using SGD). We define the training data set (MNIST) and the loader of the data.

Line 24–45: That's where the loop for training the neural network happens. We load the inputs and the expected outputs. We run the forward pass and the backward pass and the optimizer.

There's definitely some extra stuff in here (the number of GPUs and nodes, for example) that we don't need yet, but it's helpful to put the whole skeleton in place.

DistributedDataParallel

For 'nn.DistributedDataParallel', the machine has one process per GPU, and each model is controlled by each process. The GPUs can all be on the same node or across multiple nodes. Only gradients are passed between the processes/GPUs.



During training, each process loads its own mini-batch from disk and passes it to its GPU. Each GPU does its forward pass, then the gradients are all-reduced across the GPUs. Gradients for each layer do not depend on previous layers, so the gradient all-reduce is calculated concurrently with the backwards pass to further alleviate the networking bottleneck. At the end of the backwards pass, every node has the averaged gradients, ensuring that the model weights stay synchronized.

Tutorial

To do this with multiprocessing, we need a script that will launch a process for every GPU. Each process needs to know which GPU to use, and where it ranks amongst all the processes that are running. We'll need to run the script on each node.

Let's take a look at the changes to each function. I've fenced off the new code to make it easy to find.

```
1  def main():
2      parser = argparse.ArgumentParser()
3      parser.add_argument('-n', '--nodes', default=1,
4                          type=int, metavar='N')
5      parser.add_argument('-g', '--gpus', default=1, type=int,
6                          help='number of gpus per node')
7      parser.add_argument('-nr', '--nr', default=0, type=int,
8                          help='ranking within the nodes')
9      parser.add_argument('--epochs', default=2, type=int,
10                         metavar='N',
11                         help='number of total epochs to run')
12     args = parser.parse_args()
13     ########################################################
14     args.world_size = args.gpus * args.nodes                #
15     os.environ['MASTER_ADDR'] = '10.57.23.164'              #
16     os.environ['MASTER_PORT'] = '8888'                      #
17     mp.spawn(train, nprocs=args.gpus, args=(args,))         #
18     ########################################################
```

multi_parallel.py hosted with ❤️ by GitHub view raw

Let's go over the arguments of the main function:

- `args.nodes` is the total number of nodes we are using (number of machines).
- `args.gpus` is the number of GPUs on each node (on each machine).
- `args.nr` is the rank of the current node (machine) within all the nodes (machines), and goes from 0 to `args.nodes - 1`.

Let's go through the new changes line by line:

Line 12: Based on the number of nodes and GPUs per node, we can calculate the `world_size`, or the total number of processes to run, which is equal to the total number of GPUs times the number of nodes.

Line 13: This tells the multiprocessing module what IP address to look at for process 0. It needs this so that all the processes can sync up initially. This needs to be the same across all nodes.

Line 14: Likewise, this is the port to use when looking for process 0.

Line 15: Now, instead of running the train function once, we will spawn `args.gpus` processes, each of which runs `train(i, args)`, where `i` goes from 0 to `args.gpus` - 1. Remember, we run the `main()` function on each node, so that in total there will be `args.nodes * args.gpus = args.world_size` processes.

Instead of lines 13 and 14, I could have run `export MASTER_ADDR=10.57.23.164` and `export MASTER_PORT=8888` in the terminal.

Next, let's look at the modifications to `train`. I'll fence the new lines again.

```
def train(gpu, args):
    ############################################################
    rank = args.nr * args.gpus + gpu
    dist.init_process_group(
        backend='nccl',
                init_method='env://',
        world_size=args.world_size,
        rank=rank
    )
    ############################################################

    torch.manual_seed(0)
    model = ConvNet()
    torch.cuda.set_device(gpu)
    model.cuda(gpu)
    batch_size = 100
    # define loss function (criterion) and optimizer
    criterion = nn.CrossEntropyLoss().cuda(gpu)
    optimizer = torch.optim.SGD(model.parameters(), 1e-4)

    ##################################################################
    # Wrap the model
    model = nn.parallel.DistributedDataParallel(model,
                                                device_ids=[gpu])
    ##################################################################

    # Data loading code
    train_dataset = torchvision.datasets.MNIST(
        root='./data',
        train=True,
        transform=transforms.ToTensor(),
        download=True
    )
    ###################################################################
    train_sampler = torch.utils.data.distributed.DistributedSampler(
        train_dataset,
        num_replicas=args.world_size,
        rank=rank
    )
    ###################################################################

    train_loader = torch.utils.data.DataLoader(
        dataset=train_dataset,
       batch_size=batch_size,
    ##############################
```

```
46          shuffle=False,            #
47      ##############################
48          num_workers=0,
49          pin_memory=True,
50      ##############################
51        sampler=train_sampler)    #
52      ##############################
53      ...
```

train_distributed.py hosted with ❤️ by GitHub view raw

I've removed some of thee code and replaced it with `...`, to make this tutorial easier to read, but if you want the full script, it is here.

Line 3: This is the global rank of the process within all of the processes. We'll use this for line 6.

Lines 4–6: Initialize the process and join up with the other processes. This is "blocking," meaning that no process will continue until all processes have joined. I'm using the `NCCL`, since its the fastest available.. The `init_method` tells the process group where to look for some settings. In this case, it's looking at environment variables for the `MASTER_ADDR` and `MASTER_PORT`, which we set within `main`. That's why we set it to `env://`. We could have set the `world_size` there as well as `WORLD_SIZE`.

Line 23: Wrap the model as a `DistributedDataParallel` model. This reproduces the model onto each GPU.

Lines 35–39: The `nn.utils.data.DistributedSampler` makes sure that each process gets a different slice of the training data, whenever loading the data. If you want to debug and verify that each GPU is loading the right data, you can calculate the SHAs of the tensors loaded into each GPU.

Lines 46 and 51: Use the `nn.utils.data.DistributedSampler` instead of shuffling the usual way. That's why we set shuffle to false.

To run this on, say, 4 nodes with 8 GPUs each, we need 4 terminals (one on each node). On node 0 (as set by line 13 in `main`):

```
1   python src/mnist-distributed.py -n 4 -g 8 -nr 0
```

bash_run.sh hosted with ❤️ by GitHub view raw

Then, on the other nodes:

```
1   python src/mnist-distributed.py -n 4 -g 8 -nr i
```

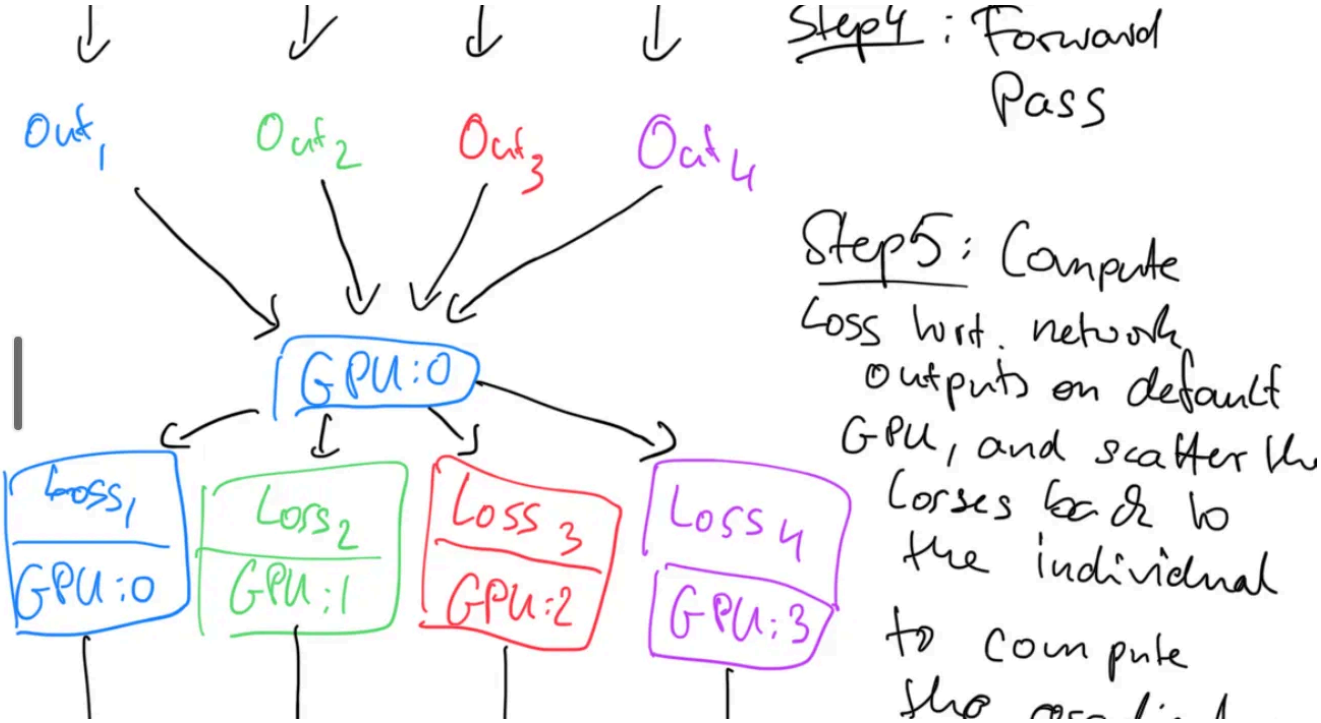
other_nodes_bash_run.sh hosted with ❤️ by GitHub view raw

for i∈1,2,3. In other words, we run this script on each node, telling it to launch `args.gpus` processes that sync with each other before training begins.

Note that the effective batch_size is now the per/GPU batch_size (the value in the script) * the total number of GPUs (the world size).

Problems

There are a few problems that might occur whenever running the same model in a few GPUs instead of one GPU. The biggest problem that can occur is that the main GPU may run out of memory. The reason for that is because the first GPU will save all the different outputs for the different GPUs to calculate the loss.



The following message will be displayed on the console whenever you are training the network: `ran out of memory trying to allocate 2.59GiB`

In order to solve this problem, and reduce the amount of memory usage, we use two techniques:

1. Reduce the batch_size
2. Use Apex for mixed precision

The first technique is pretty straightforward, and usually involves just changing one hyper-parameter.

The second technique means that we are going to decrease the precision of the weights that are used in the neural network, and therefore use less memory. Mixed-precision means you use 16-bit for certain things but keep things like weights at 32-bit. To learn more about mixed precision, I recommend reading this blog post:

What is the difference between FP16 and FP32 when doing deep learning?

Answer (1 of 3): This is a well-timed question, as we just added FP16 support to Horovod last Friday. So naturally, I'm...

www.quora.com

Apex for mixed precision

In order to solve the problem of running out of memory, we recommend using lower precision numbers. That allows us to use larger batch sizes and take advantage of NVIDIA Tensor Cores for faster computation.

In order to make APEX work, we need to change 2 parts of the code. The first is inside the `train` loop inside the codebase:

Training step

```
rank = args.nr * args.gpus + gpu
    dist.init_process_group(
        backend='nccl',
        init_method='env://',
        world_size=args.world_size,
        rank=rank)

        torch.manual_seed(0)
    model = ConvNet()
    torch.cuda.set_device(gpu)
    model.cuda(gpu)
    batch_size = 100
    # define loss function (criterion) and optimizer
    criterion = nn.CrossEntropyLoss().cuda(gpu)
    optimizer = torch.optim.SGD(model.parameters(), 1e-4)
    # Wrap the model
    ################################################################
    model, optimizer = amp.initialize(model, optimizer,
                                      opt_level='O1')
    model = DDP(model)
    ################################################################
    # Data loading code
        ...
    start = datetime.now()
    total_step = len(train_loader)
    for epoch in range(args.epochs):
        for i, (images, labels) in enumerate(train_loader):
            images = images.cuda(non_blocking=True)
            labels = labels.cuda(non_blocking=True)
            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
    ################################################################
            with amp.scale_loss(loss, optimizer) as scaled_loss:
                scaled_loss.backward()
    ################################################################
            optimizer.step()
     ...
```

training_apex.py hosted with ❤ by GitHub view raw

Line 18: `amp.initialize` wraps the model and optimizer for mixed precision training. Note that that the model must already be on the correct GPU before calling `amp.initialize`. The `opt_level` goes from `O0`, which uses all floats, through `O3`, which uses half-precision throughout. `O1` and `O2` are different degrees of mixed-precision, the details of which can be found in the Apex documentation.

Line 20: `apex.parallel.DistributedDataParallel` is a drop-in replacement for `nn.DistributedDataParallel`. We no longer have to specify the GPUs because Apex only allows one GPU per process. It also assumes that the script calls `torch.cuda.set_device(local_rank)` (line 10) before moving the model to GPU.

Lines 37–38: Mixed-precision training requires that the loss is scaled in order to prevent the gradients from underflowing. Apex does this automatically.

Make sure that whenever you initialize AMP, you set `opt_level=O1`, due to a bug with its implementation

Checkpoint

We need to change the way we save and load models whenever using Apex, See the following issue. And we need to change the way we save checkpoints and load them to our models:

```
# Save checkpoint
checkpoint = {
    'model': model.state_dict(),
    'optimizer': optimizer.state_dict(),
    'amp': amp.state_dict()
}
torch.save(checkpoint, 'amp_checkpoint.pt')
...


# Restore
model = ...
optimizer = ...
checkpoint = torch.load('amp_checkpoint.pt')

model, optimizer = amp.initialize(model, optimizer, opt_level=opt_level)
model.load_state_dict(checkpoint['model'])
optimizer.load_state_dict(checkpoint['optimizer'])
amp.load_state_dict(checkpoint['amp'])

# Continue training
...
```

checkpoint_apex.py hosted with ❤️ by GitHub view raw

Line 5: We add the `amp.state_dict` to the checkpoint

Line 19: We load the `state_dict` to amp here.

Conclusion

With all of that, you should be able to start training your model in multiple GPUs. We recommend start training a small model in one GPU before trying to scale training to multiple GPUs. But this tutorial may help if there is a need to scale training.

Links and references:

https://lambdalabs.com/blog/introduction-multi-gpu-multi-node-distributed-training-nccl-2-0/

https://medium.com/intel-student-ambassadors/distributed-training-of-deep-learning-models-with-pytorch-1123fa538848

https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

https://medium.com/huggingface/training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed-setups-ec88c3e51255

https://medium.com/south-park-commons/scaling-transformer-xl-to-128-gpus-85849508ec35

https://yangkky.github.io/2019/07/08/distributed-pytorch-tutorial.html

Machine Learning | Dataparallel | Multi Gpu | Scaling | Neural Networks

Written by Giuliano Giacaglia

1.7K Followers · Writer for Towards Data Science

https://holloway.com/mtt

Follow

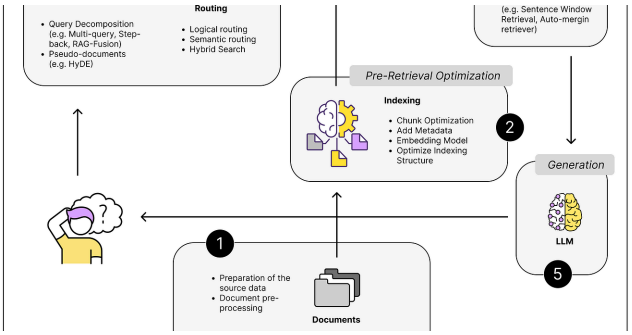
More from Giuliano Giacaglia and Towards Data Science

Giuliano Giacaglia in Towards Data Science

Transformers

Transformers are a type of neural network architecture that have been gaining...

Mar 10, 2019 10.5K 37



Dominik Polzer in Towards Data Science

17 (Advanced) RAG Techniques to Turn Your LLM App Prototype into...

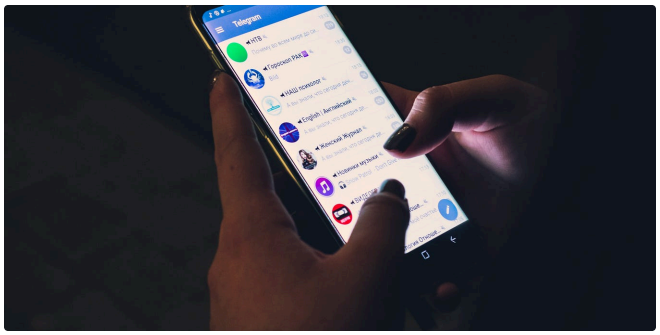
A collection of RAG techniques to help you develop your RAG app into something robus...

Jun 26 1.98K 20

Mauro Di Pietro in Towards Data Science

GenAI with Python: RAG with LLM (Complete Tutorial)

Build your own ChatGPT with multimodal data and run it on your laptop without GPU



Giuliano Giacaglia

How hackers are getting access to 1000s of Telegram accounts

A security hole is letting hackers access Telegram accounts

Jun 28 725 13 Jul 28, 2019 209 4

See all from Giuliano Giacaglia See all from Towards Data Science

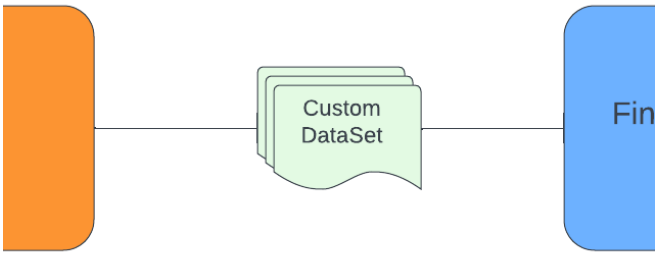
Recommended from Medium

RickyYang in DevOps.dev

Ubuntu 22.04 LTS with Nvidia GeForce RTX 4090 and CUDA 12.x

In order to run the Roop Face Swap project, I set up an environment on my Ubuntu 22.04...

Jan 27 78 1



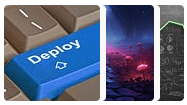
Suman Das

Fine Tune Large Language Model (LLM) on a Custom Dataset with...

The field of natural language processing has been revolutionized by large language...

Jan 24 1.4K 17

Lists



Predictive Modeling w/ Python

20 stories · 1394 saves

Practical Guides to Machine Learning

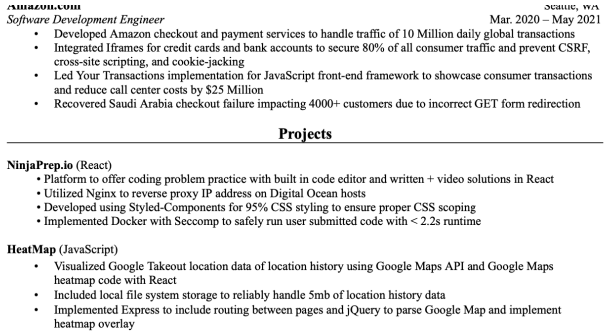
10 stories · 1687 saves

Natural Language Processing

1592 stories · 1143 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 423 saves



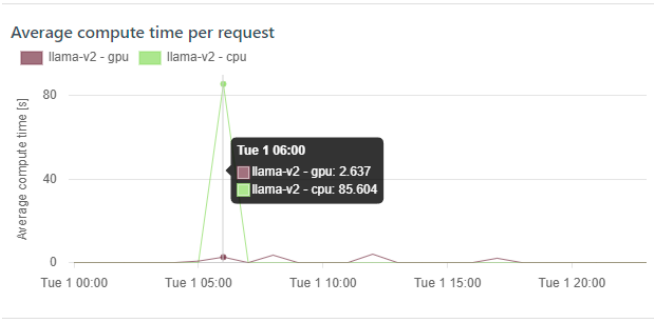
Software Development Engineer Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay



Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

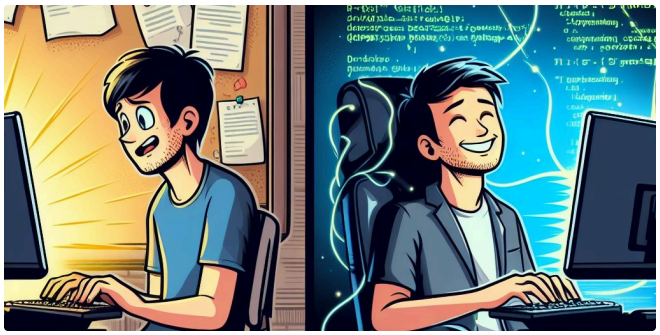
May 31 14.1K 211

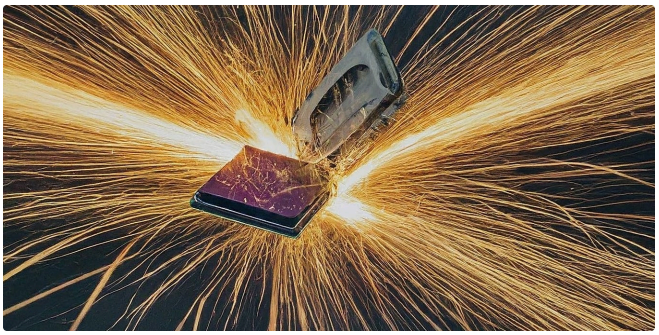
Yvan Fafchamps

How to benchmark and optimize LLM inference performance (for...

Including specific metrics and techniques to look out for

Apr 4 13





Abhay Parashar in The Pythoneers

17 Mindblowing Python Automation Scripts I Use Everyday

Xiwei Zhou

Apple M1 Max vs CoLab T4

Battle of the Local LLM Inference Performance

Scripts That Increased My Productivity and Performance

Jul 10 3.7K 33

Mar 11

See more recommendations