Copyright Notice

©2013-2022 Tencent Cloud. All rights reserved.

Copyright in this document is exclusively owned by Tencent Cloud. You must not reproduce, modify, copy or distribute in any way, in whole or in part, the contents of this document without Tencent Cloud's the prior written consent.

Trademark Notice

All trademarks associated with Tencent Cloud and its services are owned by Tencent Cloud Computing (Beijing) Company Limited and its affiliated companies. Trademarks of third parties referred to in this document are owned by their respective proprietors.

Service Statement

This document is intended to provide users with general information about Tencent Cloud's products and services only and does not form part of Tencent Cloud's terms and conditions. Tencent Cloud's products or services are subject to change. Specific products and services and the standards applicable to them are exclusively provided for in Tencent Cloud's applicable terms and conditions.
Contents

Best Practices

  Using Docker to Install TensorFlow and Set GPU/CPU Support
  Using GPU Instance to Train ViT Model
Best Practices

Using Docker to Install TensorFlow and Set GPU/CPU Support

Overview

You can use Docker to run TensorFlow in a GPU instance quickly. In this way, you only need to install the NVIDIA® driver program in the instance and don't need to install NVIDIA® CUDA® Toolkit.

This document describes how to use Docker to install TensorFlow and configure GPU/CPU support in a GPU instance.

Notes

- This document uses a GPU instance on Ubuntu 20.04 as an example.
- The GPU driver has been installed in your GPU instance.

Directions

Installing Docker
1. Log in to the instance and run the following commands to install the required system tools:

```
sudo apt-get update
```

```
sudo apt-get install ca-certificates curl gnupg lsb-release
```

2. Run the following command to install the GPG certificate to write the software source information:

```
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

3. Run the following commands to update and install Docker-CE:

```
sudo apt-get update
```

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

### Installing TensorFlow

### Setting the NVIDIA container toolkit

1. Run the following command to set the package repository and GPG key as instructed in Setting up NVIDIA Container Toolkit:

```
distribution=$(./etc/os-release;echo $ID$VERSION_ID) \
&& curl -s -L https://nvidia.github.io/libnvidia-container/$distribution/libnvidia-container.list | \
```
sed 's#[de]b https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] https://#g' | \
    sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list

2. Run the following command to install the nvidia-docker2 package and its dependencies:

    sudo apt-get update

    sudo apt-get install -y nvidia-docker2

3. Run the following command to set the default runtime and restart the Docker daemon to complete installation:

    sudo systemctl restart docker

4. Then, you can run the following command to run the base CUDA container to test the job settings:

    sudo docker run --rm --gpus all nvidia/cuda:11.0.3-base-ubuntu20.04 nvidia-smi

The following information will appear:

+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.51.06 Driver Version: 450.51.06 CUDA Version: 11.0          |
+-----------------------------------------------------------------------------+
| GPU Name Persistence-M| Bus-Id Disp.A | Volatile Uncorr. ECC | Fan Temp Perf Pwr:Usage/Cap| Memory-Usage | GPU-Util Compute M. | MIG M. |
| 0 Tesla T4 On | 00000000:00:1E.0 Off | 0 | N/A 34C P8 9W / 70W | 0MiB / 15109MiB | 0% Default |
+ | N/A |
+ | Processes: |
| 0 | GPU GSI PID Type Process name GPU Memory |
| ID ID Usage |
+=============================================================================+
| No running processes found |
+-----------------------------------------------------------------------------+
Downloading a TensorFlow Docker image

The official TensorFlow Docker images are in the tensorflow/tensorflow code repository in Docker Hub. Image tags are defined in the following format as listed in Tags:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>latest</td>
<td>Latest (default) tag of the binary TensorFlow CPU image.</td>
</tr>
<tr>
<td>nightly</td>
<td>Nightly tag of the TensorFlow image, which is unstable.</td>
</tr>
<tr>
<td>version</td>
<td>Tag of the TensorFlow binary image, such as <code>2.1.0</code>.</td>
</tr>
<tr>
<td>devel</td>
<td>TensorFlow master Nightly tag of the development environment, which contains the TensorFlow source code.</td>
</tr>
<tr>
<td>custom-op</td>
<td>Special experimental image for custom TensorFlow operation development. For more information, see tensorflow/custom-op.</td>
</tr>
</tbody>
</table>

Each basic tag has variants with new or modified features:

<table>
<thead>
<tr>
<th>Tag Variant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag -gpu</td>
<td>Specified tag supporting GPU</td>
</tr>
<tr>
<td>tag -jupyter</td>
<td>Specified tag for Jupyter, which contains the TensorFlow tutorial laptop.</td>
</tr>
</tbody>
</table>

You can use multiple variants at a time. For example, the following command will download the TensorFlow image tags to your computer:

```
docker pull tensorflow/tensorflow # latest stable release
docker pull tensorflow/tensorflow:devel-gpu # nightly dev release w/ GPU support
docker pull tensorflow/tensorflow:latest-gpu-jupyter # latest release w/ GPU support and Jupyter
```

Starting the TensorFlow Docker container

Run the following command to start and configure the TensorFlow container. For more information, see Docker run reference.

```
docker run [-it] [--rm] [-p hostPort:containerPort] tensorflow/tensorflow[:tag] [command]
```
Examples

**Using an image supporting only CPU**

Use an image with the `latest` tag to verify the TensorFlow installation result. Docker will download the latest TensorFlow image when it runs for the first time.

```
docker run -it --rm tensorflow/tensorflow:latest  
python -c "import tensorflow as tf; print(tf.reduce_sum(tf.random.normal([1000, 1000])));
```

Below are the samples of other TensorFlow Docker solutions:

- **Start the bash shell session in the container where TensorFlow is configured:**
  
  ```
docker run -it tensorflow/tensorflow bash
  ```

- **To run the TensorFlow program developed on the host in the container, use the `-v hostDir:containerDir` `-w workDir` parameter to load the server directory and change the container working directory as follows:**
  
  ```
docker run -it --rm -v $PWD:/tmp -w /tmp tensorflow/tensorflow python ./script.py
  ```

**Note**

When you allow the host to access the files created in the container, permission problems may occur. Generally, we recommend you modify files on the host system.

- **Use TensorFlow with the `nightly` tag to start Jupyter laptop server:**

  ```
docker run -it -p 8888:8888 tensorflow/tensorflow:nightly-jupyter
  ```

  Use a browser to visit [http://127.0.0.1:8888/?token=...](http://127.0.0.1:8888/?token=...) as instructed at the [Jupyter website](https://jupyter.org).

**Using an image supporting GPU**

Run the following command to download and run the TensorFlow image supporting GPU:

```
docker run --gpus all -it --rm tensorflow/tensorflow:latest-gpu  
python -c "import tensorflow as tf; print(tf.reduce_sum(tf.random.normal([1000, 1000])));
```
It may take a while to set the image supporting GPU. To run the GPU-based script repeatedly, you can use `docker exec` to use the container repeatedly.

Run the following command to use the latest TensorFlow GPU image to start the `bash` shell session in the container:

```
docker run --gpus all -it tensorflow/tensorflow:latest-gpu bash
```
Using GPU Instance to Train ViT Model

Last updated: 2022-12-09 15:18:55

Note
This document is written by a Cloud GPU Service user and is for study and reference only.

Overview

This document describes how to use a GPU instance to train a ViT model offline to complete a simple image classification task.

ViT Model Overview

The Vision Transformer (ViT) model is proposed by Alexey Dosovitskiy to get the state-of-the-art (SOTA) result from multiple tasks.
For an input image, ViT splits it into multiple subimage patches. Each patch is spliced with position embedding and combined with class labels to be input to transformer encoder together. After the corresponding output layer results of the class label positions pass through a network, the ViT result will be output. In the pretraining status, the ground truth of the result can replaced by a patch of the mask.

**Instance Environment**

- **Instance type**: In this document, you can select a GN7 or GN8 model. Based on the GPU performance comparison provided in [Tesla P40 vs Tesla T4](example.com), the performance of T4 in Turing architecture is higher than that of P40 in Pascal architecture. Therefore, GN7.5XLARGE80 is selected in this document.
- **Region**: As large datasets may need to be uploaded, we recommend you select the region with the lowest latency. This document uses the [online ping](example.com) tool for testing. As the latency between the test region and Chongqing region where GN7 resides is the lowest, Chongqing region is selected in this example.
- **System disk**: 100 GB Premium Cloud Storage disk.
- **Operating system**: Ubuntu 18.04.
- **Bandwidth**: 5 Mbps.
- **Local operating system**: macOS

**Directions**

**Setting passwordless login for your instance (optional)**

1. (Optional) You can configure the server alias in `~/.ssh/config` on your local server. In this document, the alias `tcg` is used.

2. Run the `ssh-copy-id` command to copy the SSH public key of the local server to the GPU instance.

3. Run the following command in the GPU instance to disable password login to enhance security:

   ```bash
   echo 'PasswordAuthentication no' | sudo tee -a /etc/ssh/ssh_config
   ``

4. Run the following command to restart the SSH service.

   ```bash
   sudo systemctl restart sshd
   ```
Configuring the PyTorch-GPU development environment

To use pytorch-gpu for development, you need to further configure the environment as follows:

1. Install the NVIDIA graphics card driver.
   Run the following command to install the NVIDIA graphics card driver:
   
   ```
   sudo apt install nvidia-driver-418
   ```

   After the installation is completed, run the following command to check whether the installation is successful:
   
   ```
   nvidia-smi
   ```

   If the following result is returned, the installation is successful.

2. Configure the conda environment.
   Run the following commands to configure the conda environment:
   
   ```
   wget https://repo.anaconda.com/miniconda/Miniconda3-py39\_4.11.0-Linux-x86\_64.sh
   ```
3. Compile the `.condarc` file to add the following software source information and replace the conda software source with the Qinghua source.

```plaintext
channels:
- defaults
show_channel_urls: true
default_channels:
- https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
- https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/r
- https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/msys2
custom_channels:
conda-forge: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud
msys2: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud
bioconda: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud
menpo: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud
pytorch: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud
pytorch-lts: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud
simpleitk: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud
```

4. Run the following command to set the `pip` source to the Tencent Cloud image source.

```plaintext
pip config set global.index-url https://mirrors.cloud.tencent.com/pypi/simple
```

5. Install PyTorch.

Run the following command to install PyTorch:

```plaintext
conda install pytorch torchvision cudatoolkit=11.4 -c pytorch --yes
```

Run the following commands to view whether PyTorch is installed successfully:

```plaintext
python
```
import torch

print(torch.cuda.is_available())

If the following result is returned, PyTorch is installed successfully:

Preparing the experiment data

The test task in this training is an image classification task and uses the flower image classification dataset in the Tencent Cloud online document. The dataset contains five classes of flowers and is 218 MB in size. Below are the sampled dataset results (examples of images of flowers in each class):

<table>
<thead>
<tr>
<th>Daisy</th>
<th>Dandelion</th>
<th>Rose</th>
<th>Sunflower</th>
<th>Tulip</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Daisy Image" /></td>
<td><img src="image2.png" alt="Dandelion Image" /></td>
<td><img src="image3.png" alt="Rose Image" /></td>
<td><img src="image4.png" alt="Sunflower Image" /></td>
<td><img src="image5.png" alt="Tulip Image" /></td>
</tr>
</tbody>
</table>
The data of each class in the raw dataset is stored in the folder of the corresponding class. You need to convert it to the standard format of ImageNet and divide the training and verification datasets at the ratio of 4:1. Use the following code to convert the format:

```python
# split data into train set and validation set, train:val=scale
import shutil
import os
import math
scale = 4
data_path = './raw'
data_dst = '../train_val'

# create imagenet directory structure
os.mkdir(data_dst)
for item in os.listdir(data_path):
    item_path = os.path.join(data_path, item)
    if os.path.isdir(item_path):
        train_dst = os.path.join(data_dst, 'train', item)
        val_dst = os.path.join(data_dst, 'validation', item)
        os.mkdir(train_dst)
        os.mkdir(val_dst)
        files = os.listdir(item_path)
        print(f'Class {item}: Total sample count is {len(files)}')
        split_idx = math.floor(len(files) * scale / (1 + scale))
        print(f'Train sample count is {split_idx}')
        print(f'Validation sample count is {len(files) - split_idx}')
        for idx, file in enumerate(files):
            file_path = os.path.join(item_path, file)
            if idx <= split_idx:
                shutil.copy(file_path, train_dst)
            else:
                shutil.copy(file_path, val_dst)
        print(f'Split Complete. File path: {data_dst}')
```

Below is the dataset overview:

- **Class roses:**
  - Total sample count is 641
  - Train sample count is 512
  - Validation sample count is 129

- **Class sunflowers:**
  - Total sample count is 699
  - Train sample count is 559
  - Validation sample count is 140

- **Class tulips:**
Total sample count is 799
Train sample count is 639
Validation sample count is 160
Class daisy:
Total sample count is 633
Train sample count is 506
Validation sample count is 127
Class dandelion:
Total sample count is 898
Train sample count is 718
Validation sample count is 180

To accelerate the training process, you need to further convert the dataset to a GPU-friendly format such as NVIDIA Data Loading Library (DALI). The DALI library can use GPU to replace CPU to accelerate data preprocessing. When data in the ImageNet format already exists, you can simply run the following command to use DALI:

```bash
git clone https://github.com/ver217/imagenet-tools.git
cd imagenet-tools && python3 make_tfrecords.py 
--raw_data_dir="/./train\_val" 
--local_scratch_dir="/./train\_val\tfrecord" && 
python3 make_idx.py --tfrecord_root="/./train\_val\tfrecord"
```

**Model training result**

To facilitate subsequent training of large distributed models, this document describes how to train and develop a model based on the distributed training framework Colossal-AI. Colossal-AI provides a set of easy-to-use APIs, which enables you to easily perform data, model, pipeline, and mixed parallel training.

Based on the demo provided by Colossal-AI, this document uses ViT integrated in the pytorch-image-models repository for implementation. The minimum `vit_tiny_patch16_224` model at a resolution of 224*224 is used, where each sample is divided into 16 patches.

1. Run the following command to install Colossal-AI and pytorch-image-models as instructed in Start Locally:

```bash
pip install colossalai==0.1.5+torch1.11cu11.3 -f https://release.colossalai.org
```

2. Write the following model training code based on the demo provided by Colossal-AI:

```python
from pathlib import Path
from colossalai.logging import get_dist_logger
import colossalai
import torch
```
import os
from colossalai.core import global\_context as gpc
from colossalai.utils import get\_dataloader, MultiTimer
from colossalai.trainer import Trainer, hooks
from colossalai.nn.metric import Accuracy
from torchvision import transforms
from colossalai.nn.lr\_scheduler import CosineAnnealingLR
from tqdm import tqdm
from titans.utils import barrier\_context
from colossalai.nn.lr\_scheduler import LinearWarmupLR
from timm.models import vit\_tiny\_patch16\_224
from titans.dataloader.imagenet import build\_dali\_imagenet
from mixup import MixupAccuracy, MixupLoss
def main():
    parser = colossalai.get\_default\_parser()
    args = parser.parse\_args()
    colossalai.launch\_from\_torch(config='./config.py')
    logger = get\_dist\_logger()
    # build model
    model = vit\_tiny\_patch16\_224(num\_classes=5, drop\_rate=0.1)
    # build dataloader
    root = os.environ.get('DATA', '../train_val\_tfrecord')
    train\_dataloader, test\_dataloader = build\_dali\_imagenet(
        root, rand\_augment=True)
    # build criterion
    criterion = MixupLoss(loss\_fn\_cls=torch.nn.CrossEntropyLoss)
    # optimizer
    optimizer = torch.optim.SGD(
        model.parameters(), lr=0.1, momentum=0.9, weight\_decay=5e-4)
    # lr\_scheduler
    lr\_scheduler = CosineAnnealingLR(
        optimizer, total\_steps=gpc.config.NUM\_EPOCHS)
    engine, train\_dataloader, test\_dataloader, \_ = colossalai.initialize(
        model,
        optimizer,
        criterion,
        train\_dataloader,
        test\_dataloader,
    )
    # build a timer to measure time
    timer = MultiTimer()
    # create a trainer object
    trainer = Trainer(engine=engine, timer=timer, logger=logger)
    # define the hooks to attach to the trainer
    hook\_list = [
        hooks.LossHook(),
        hooks.LRSchedulerHook(lr\_scheduler=lr\_scheduler, by\_epoch=True),
hooks.AccuracyHook(accuracy_func=MixupAccuracy()),
hooks.LogMetricByEpochHook(logger),
hooks.LogMemoryByEpochHook(logger),
hooks.LogTimingByEpochHook(timer, logger),
hooks.TensorboardHook(log_dir='./tb_logs', ranks=[0]),
hooks.SaveCheckpointHook(checkpoint_dir='./ckpt')
]
# start training
trainer.fit(train_dataloader=train_dataloader,
epochs=gpc.config.NUM_EPOCHS,
test_dataloader=test_dataloader,
test_interval=1,
hooks=hook_list,
display_progress=True)
if __name__ == '__main__':
    main()

Below is the specific model configuration:

```python
from colossalai.amp import AMP_TYPE
BATCH_SIZE = 128
DROP_RATE = 0.1
NUM_EPOCHS = 200
CONFIG = dict(fp16=dict(mode=AMP_TYPE.TORCH))
gradient_accumulation = 16
clip_grad_norm = 1.0
dali = dict(
gpu_aug=True,
mixup_alpha=0.2)
```
Below is the model execution process. Each epoch time is within 20s:

The result shows that the highest accuracy of the model with the verification dataset is 66.62%. You can also increase the number of model parameters; for example, you can change the model to `v
Summary

The biggest problem encountered in this example was that cloning from GitHub was very slow. To solve this, a tunnel and ProxyChains were used for acceleration. However, such operations violated the CVM use rules and caused a period of unavailability. Eventually, this problem was solved by deleting the proxy and submitting a ticket. Using a public network proxy doesn't comply with the CVM use regulations. To guarantee the stable operations of your business, do not violate the regulations.

References

[2] NVIDIA/DALI