Exercise_CPU_and_the_DevCloud

June 24, 2020

1 Exercise: CPU and the Devcloud

Now that we've walked through the process of requesting a CPU on Intel's DevCloud and loading a model, you will have the opportunity to do this yourself with the addition of running inference on an image.

In this exercise, you will do the following: 1. Write a Python script to load a model and run inference 10 times on a CPU on Intel's DevCloud. * Calculate the time it takes to load the model. * Calculate the time it takes to run inference 10 times. 2. Write a shell script to submit a job to Intel's DevCloud. 3. Submit a job using `qsub` on the **IEI Tank-870** edge node with an **Intel Xeon E3 1268L v5**. 4. Run `liveQStat` to view the status of your submitted job. 5. Retrieve the results from your job. 6. View the results.

Click the **Exercise Overview** button below for a demonstration.

Exercise Overview

IMPORTANT: Set up paths so we can run Dev Cloud utilities You *must* run this every time you enter a Workspace session.

```
In [ ]: %env PATH=/opt/conda/bin:/opt/spark-2.4.3-bin-hadoop2.7/bin:/opt/conda/bin:/usr/local/sb
        import os
        import sys
        sys.path.insert(0, os.path.abspath('/opt/intel_devcloud_support'))
        sys.path.insert(0, os.path.abspath('/opt/intel'))
```

1.1 The Model

We will be using the `vehicle-license-plate-detection-barrier-0106` model for this exercise. Remember that to run a model on the CPU, we need to use `FP32` as the model precision.

The model has already been downloaded for you in the `/data/models/intel` directory on Intel's DevCloud.

We will be running inference on an image of a car. The path to the image is `/data/resources/car.png`

2 Step 1: Creating a Python Script

The first step is to create a Python script that you can use to load the model and perform inference. We'll use the `%%writefile` magic to create a Python file called `inference_cpu_model.py`. In the next cell, you will need to complete the `TODO` items for this Python script.

`TODO` items:

1. Load the model
2. Prepare the model for inference (create an input dictionary)
3. Run inference 10 times in a loop

If you get stuck, you can click on the **Show Solution** button below for a walkthrough with the solution code.

In [ ]:
```
%%writefile inference_cpu_model.py

import time
import numpy as np
import cv2
from openvino.inference_engine import IENetwork
from openvino.inference_engine import IECore
import argparse

def main(args):
    model=args.model_path
    model_weights=model+'.bin'
    model_structure=model+'.xml'

    start=time.time()

    # TODO: Load the model
    model=IENetwork(model_structure, model_weights)

    core = IECore()
    net = core.load_network(network=model, device_name='CPU', num_requests=1)

    print(f"Time taken to load model = {time.time()-start} seconds")

    # Get the name of the input node
    input_name=next(iter(model.inputs))

    # Reading and Preprocessing Image
    input_img=cv2.imread('/data/resources/car.png')
    input_img=cv2.resize(input_img, (300,300), interpolation = cv2.INTER_AREA)
    input_img=np.moveaxis(input_img, -1, 0)

    # TODO: Prepare the model for inference (create input dict etc.)
    input_dict={input_name:input_img}

    start=time.time()
    for _ in range(10):
        # TODO: Run Inference in a Loop
```

```
            net.infer(input_dict)

        print(f"Time Taken to run 10 inference on CPU is = {time.time()-start} seconds")

    if __name__=='__main__':
        parser=argparse.ArgumentParser()
        parser.add_argument('--model_path', required=True)

        args=parser.parse_args()
        main(args)
```

Show Solution

2.1 Step 2: Creating a Job Submission Script

To submit a job to the DevCloud, you'll need to create a shell script. Similar to the Python script above, we'll use the `%%writefile` magic command to create a shell script called `inference_cpu_model_job.sh`. In the next cell, you will need to complete the `TODO` items for this shell script.

`TODO` items: 1. Create a `MODELPATH` variable and assign it the value of the first argument that will be passed to the shell script 2. Call the Python script using the `MODELPATH` variable value as the command line argument

If you get stuck, you can click on the **Show Solution** button below for a walkthrough with the solution code.

In [ ]:
```
%%writefile inference_cpu_model_job.sh
#!/bin/bash

exec 1>/output/stdout.log 2>/output/stderr.log

mkdir -p /output

#TODO: Create MODELPATH variable
MODELPATH = $1
#TODO: Call the Python script
python3 inference_cpu_model.py --model_path ${MODELPATH}

cd /output

tar zcvf output.tgz stdout.log stderr.log
```

Show Solution

2.2 Step 3: Submitting a Job to Intel's DevCloud

In the next cell, you will write your `!qsub` command to submit your job to Intel's DevCloud to load your model on the **Intel Xeon E3 1268L v5** CPU and run inference.

Your `!qsub` command should take the following flags and arguments: 1. The first argument should be the shell script filename 2. `-d` flag - This argument should be `.` 3. `-l` flag - This argument

should request a **Tank-870** node using an **Intel Xeon E3 1268L v5** CPU. The default quantity is 1, so the **1** after `nodes` is optional. To get the queue label for this CPU, you can go to this link 4. `-F` flag - This argument should be the full path to the model. As a reminder, the model is located in `/data/models/intel`.

Note: There is an optional flag, `-N`, you may see in a few exercises. This is an argument that only works on Intel's DevCloud that allows you to name your job submission. This argument doesn't work in Udacity's workspace integration with Intel's DevCloud.

If you get stuck, you can click on the **Show Solution** button below for a walkthrough with the solution code.

```
In [5]: job_id_core = !qsub inference_cpu_model_job.sh -d . -l nodes=1:tank-870:e3-1268l-v5 -F "
        print(job_id_core[0])
```

```
tQhefL08Tmw1aa3w1RweLyeWrIbhuY1P
```

Show Solution

2.3 Step 4: Running liveQStat

Running the `liveQStat` function, we can see the live status of our job. Running the this function will lock the cell and poll the job status 10 times. The cell is locked until this finishes polling 10 times or you can interrupt the kernel to stop it by pressing the stop button at the top:

- `Q` status means our job is currently awaiting an available node
- `R` status means our job is currently running on the requested node

Note: In the demonstration, it is pointed out that `W` status means your job is done. This is no longer accurate. Once a job has finished running, it will no longer show in the list when running the `liveQStat` function.

Click the **Running liveQStat** button below for a demonstration.

Running liveQStat

```
In [6]: import liveQStat
        liveQStat.liveQStat()
```

2.4 Step 5: Retrieving Output Files

In this step, we'll be using the `getResults` function to retrieve our job's results. This function takes a few arguments.

1. `job id` - This value is stored in the `job_id_core` variable we created during **Step 3**. Remember that this value is an array with a single string, so we access the string value using `job_id_core[0]`.
2. `filename` - This value should match the filename of the compressed file we have in our `inference_cpu_model_job.sh` shell script.

3. `blocking` - This is an optional argument and is set to `False` by default. If this is set to `True`, the cell is locked while waiting for the results to come back. There is a status indicator showing the cell is waiting on results.

Note: The `getResults` function is unique to Udacity's workspace integration with Intel's DevCloud. When working on Intel's DevCloud environment, your job's results are automatically retrieved and placed in your working directory.

Click the **Retrieving Output Files** button below for a demonstration.

Retrieving Output Files

```
In [7]: import get_results

        get_results.getResults(job_id_core[0], filename="output.tgz", blocking=True)
```

```
getResults() is blocking until results of the job (id:tQhefLO8Tmw1aa3w1RweLyeWrIbhuY1P) are read
Please wait...Success!
output.tgz was downloaded in the same folder as this notebook.
```

2.5 Step 6: View the Outputs

In this step, we unpack the compressed file using `!tar zxf` and read the contents of the log files by using the `!cat` command.

`stdout.log` should contain the printout of the print statement in our Python script.

```
In [8]: !tar zxf output.tgz
```

```
In [9]: !cat stdout.log
```

```
In [10]: !cat stderr.log
```

```
./submission.sh: line 8: MODELPATH: command not found
usage: inference_cpu_model.py [-h] --model_path MODEL_PATH
inference_cpu_model.py: error: argument --model_path: expected one argument
tar: stdout.log: file changed as we read it
```

```
In [ ]:
```