Project Report: Inventory Monitoring at Distribution Centers

Hugo Albuquerque Cosme da Silva (hualcosa@gmail.com)

Table of contents:

1. Definition

1.1 Project Overview:

Inventory monitoring is crucial for businesses as it impacts production, warehouse costs, and order fulfillment. Effective inventory management helps contain costs, ensures businesses have the correct amount of stock, and cuts down on excess inventory. By keeping track of inventory, businesses can streamline production and fulfillment processes, lower costs, minimize storage needs, and forecast sales trends. It also provides critical data to help businesses respond to trends, avoid breakdowns in supply chain management, and maintain profitability. An accurate inventory monitoring system allows businesses to understand what products need to be bought and in what quantities, helping them reduce the holding costs of inventory that does not sell. Overall, inventory monitoring plays a vital role in keeping businesses organized, optimizing operations, and satisfying customers with timely deliveries.

Advances in Artificial Intelligence, especially in computer vision, allowed novel approaches for monitoring inventory in real time. This project is about one of them: Imagine warehouses where robots put and pick items to be delivered in bins. These robots have cameras attached to them and take photos of these bins. The photos are then uploaded to the cloud and are the inputs to a machine-learning pipeline that classifies the number of objects in the bins. This output and other metadata, like SKU names, can be sent to the company's inventory system to update the product stock. The described situation is approximately what happens at Amazon's distribution center, and this project will use an Image dataset they provided.

1.2 Problem statement:

The project aims to develop a machine-learning pipeline that leverages AWS services to build an Image Classifier that counts the number of objects present in the distribution center's bins. Besides that, profiling and debugging will be performed in order to evaluate how the pipeline could be improved.

1.3 Metrics

The success criteria of this project is: To successfully build a machine learning pipeline that deals with the data consumption, preprocessing, model training, profiling, and debugging using AWS Sagemaker services.

2. Analysis:

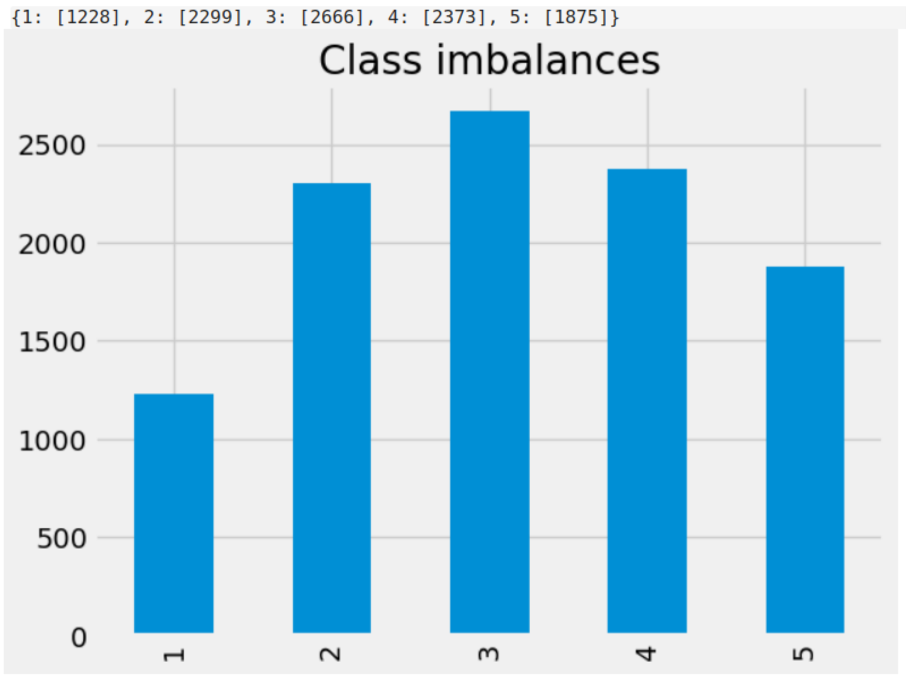
2.1 Data Exploration and Exploratory Visualization

The Amazon Bin Image Dataset contains over 500,000 images and metadata from bins of a pod in an operating Amazon Fulfillment Center. The bin images in this dataset are captured as robot units carry pods as part of normal Amazon Fulfillment Center operations. We are working with a subset of this data that contains:

- 1228 images with 1 object
- 2299 images with 2 objects
- 2666 images with 3 objects
- 2373 images with 4 objects
- 1875 images with 5 objects

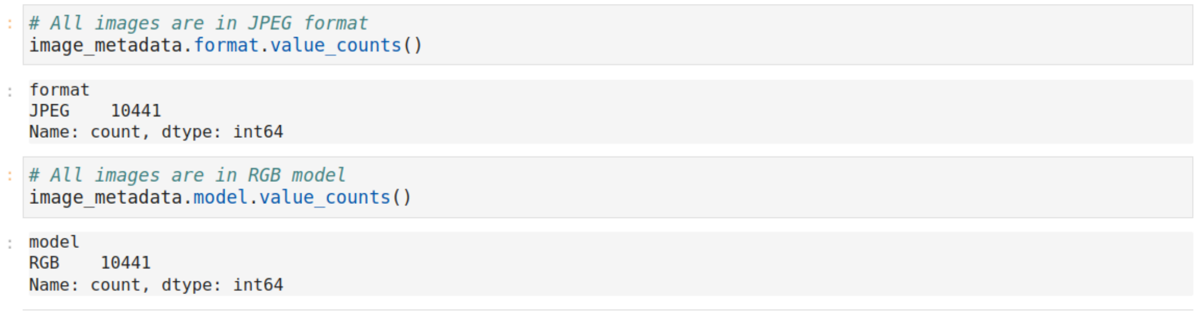
A quick exploratory data analysis was conducted to investigate image format, size, mode, and potential class imbalances. The related code and visualizations can be found in the "Dataset and Exploratory Data Analysis" section of the project's notebook(*sagemaker.ipynb*)

Figure 1 - Class Imbalances



```
{1: [1228], 2: [2299], 3: [2666], 4: [2373], 5: [1875]}
```

Figure 2 - Getting insights from the data



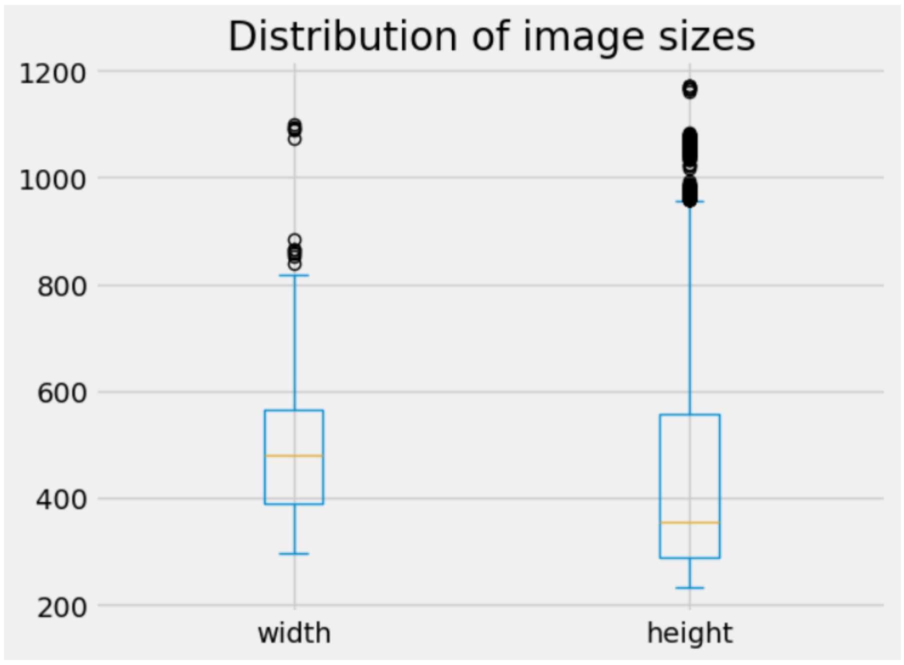
```
# All images are in JPEG format
image_metadata.format.value_counts()
```

```
format
JPEG    10441
Name: count, dtype: int64
```

```
# All images are in RGB model
image_metadata.model.value_counts()
```

```
model
RGB    10441
Name: count, dtype: int64
```

Figure 3 - Image size distribution



2.2 Development environment

The project was developed in a Sagemaker studio instance(ml.t3.medium) with a "Pytorch 2.0.0 Python 3.10 CPU Optimized" environment. Services used include AWS S3 to host the training data, Sagemaker for performing training jobs, and CloudWatch logs for monitoring instance metrics and training logs.

2.3 Algorithms and Benchmark

The benchmark model used was a pre-trained Pytorch Resnet50. The reasons for using it were:

- it showed promising results in the previous Image Classification projects
- the easiness of fine-tuning such a model

- the intellectual curiosity to evaluate how well it would perform in the project's task

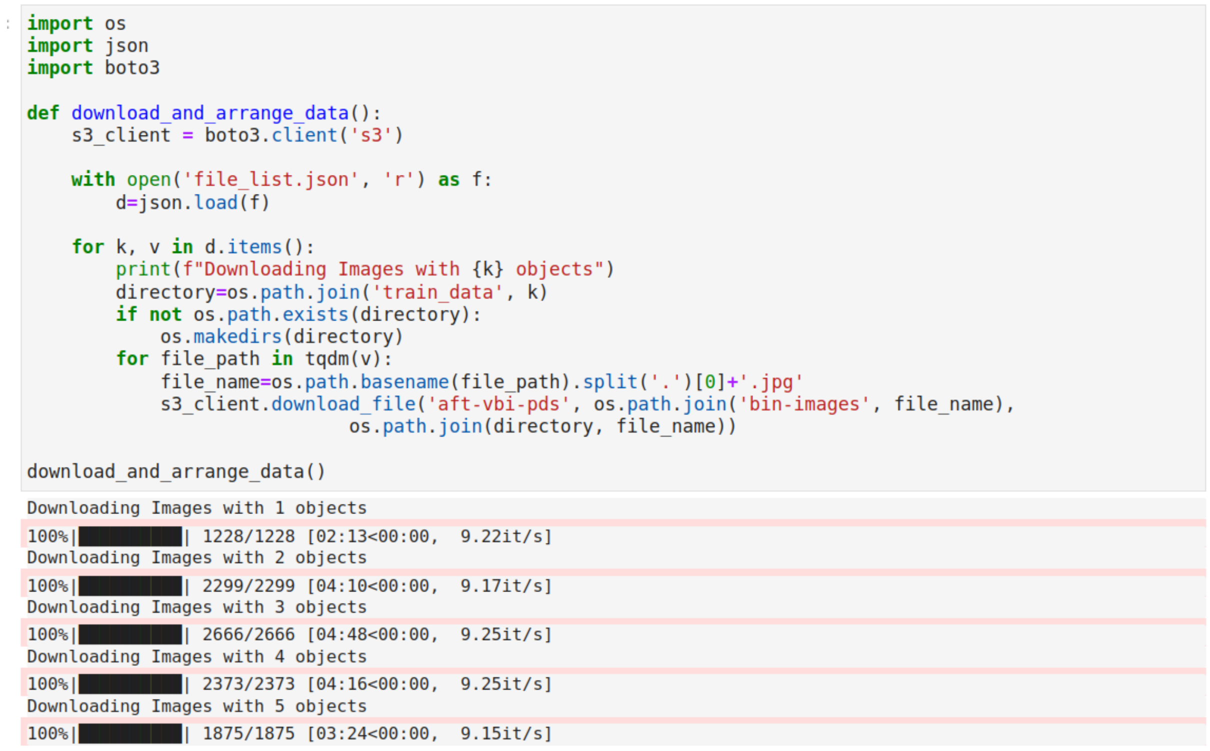
The model evaluation metric used was the average accuracy score across the different classes.

3. Methodology:

3.1 Data Acquisition:

The was downloaded from the public S3 Bucket maintained by Amazon:

Figure 4 - Downloading data from S3



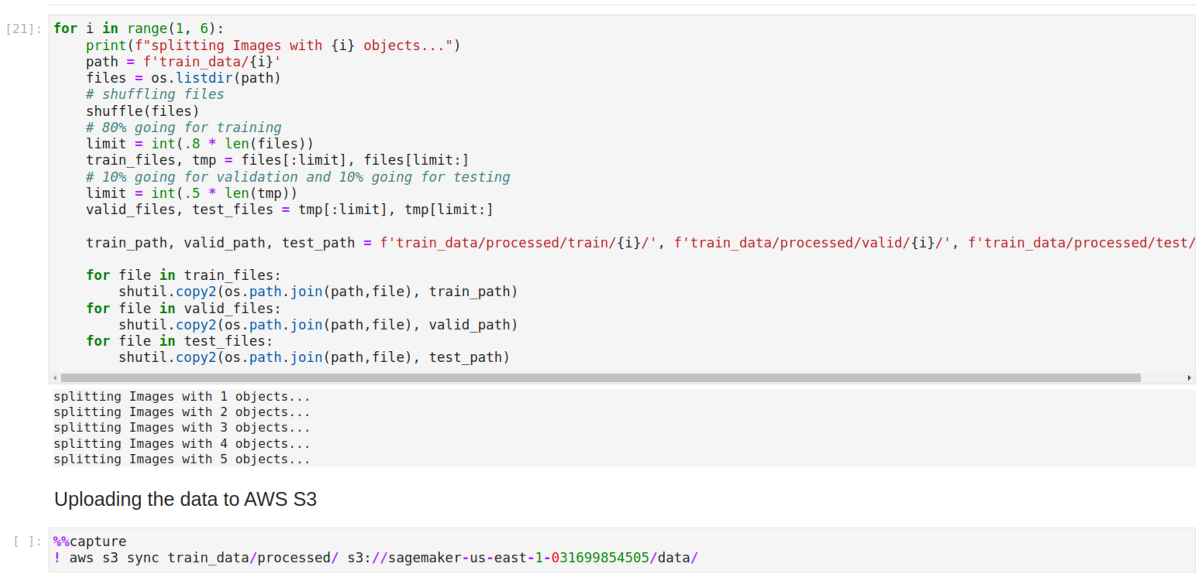
```python
import os
import json
import boto3

def download_and_arrange_data():
    s3_client = boto3.client('s3')

    with open('file_list.json', 'r') as f:
        d=json.load(f)

    for k, v in d.items():
        print(f"Downloading Images with {k} objects")
        directory=os.path.join('train_data', k)
        if not os.path.exists(directory):
            os.makedirs(directory)
        for file_path in tqdm(v):
            file_name=os.path.basename(file_path).split('.')[0]+'.jpg'
            s3_client.download_file('aft-vbi-pds', os.path.join('bin-images', file_name),
                             os.path.join(directory, file_name))

download_and_arrange_data()
```

```
Downloading Images with 1 objects
100%|██████████| 1228/1228 [02:13<00:00,  9.22it/s]
Downloading Images with 2 objects
100%|██████████| 2299/2299 [04:10<00:00,  9.17it/s]
Downloading Images with 3 objects
100%|██████████| 2666/2666 [04:48<00:00,  9.25it/s]
Downloading Images with 4 objects
100%|██████████| 2373/2373 [04:16<00:00,  9.25it/s]
Downloading Images with 5 objects
100%|██████████| 1875/1875 [03:24<00:00,  9.15it/s]
```

3.2 Data preprocessing

The data was split into training, validation, and test sets with proportions of 80%, 10%, and 10%, respectively. and then was uploaded to S3.

Figure 5 - Processing Data 1



```
[21]: for i in range(1, 6):
          print(f"splitting Images with {i} objects...")
          path = f'train_data/{i}'
          files = os.listdir(path)
          # shuffling files
          shuffle(files)
          # 80% going for training
          limit = int(.8 * len(files))
          train_files, tmp = files[:limit], files[limit:]
          # 10% going for validation and 10% going for testing
          limit = int(.5 * len(tmp))
          valid_files, test_files = tmp[:limit], tmp[limit:]

          train_path, valid_path, test_path = f'train_data/processed/train/{i}/', f'train_data/processed/valid/{i}/', f'train_data/processed/test/

          for file in train_files:
              shutil.copy2(os.path.join(path,file), train_path)
          for file in valid_files:
              shutil.copy2(os.path.join(path,file), valid_path)
          for file in test_files:
              shutil.copy2(os.path.join(path,file), test_path)
```

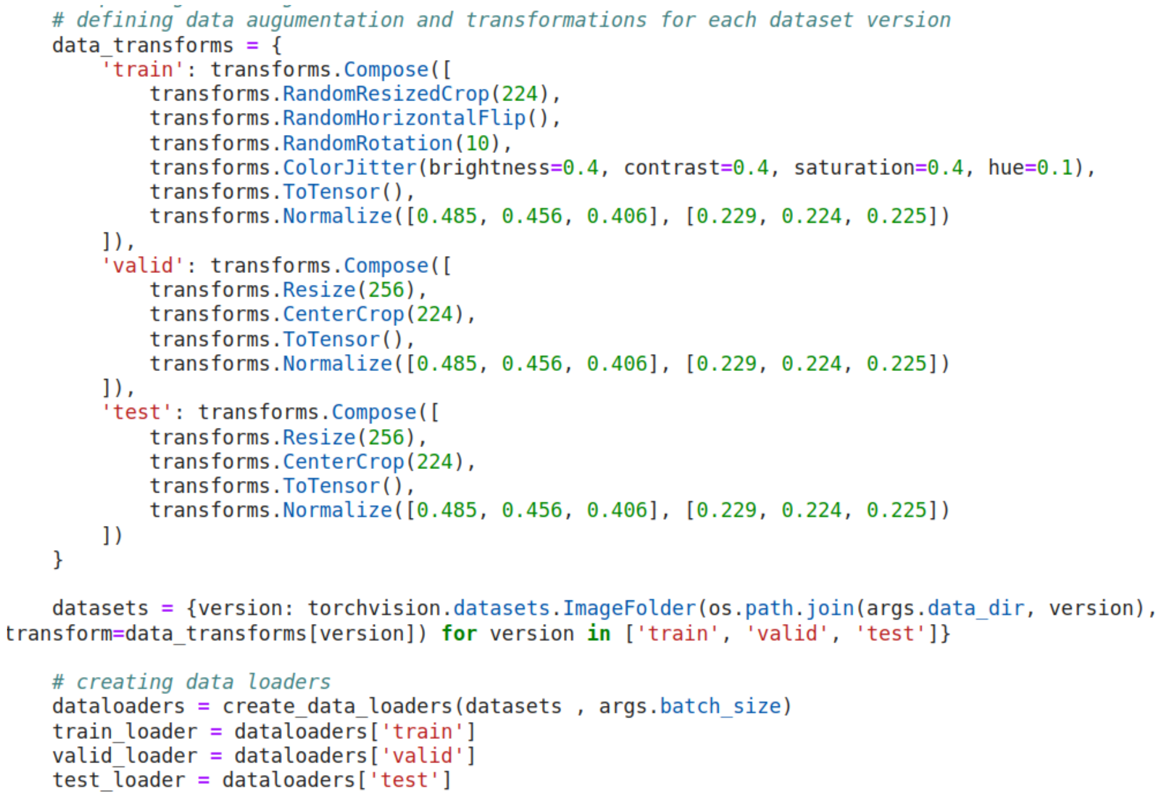
```
splitting Images with 1 objects...
splitting Images with 2 objects...
splitting Images with 3 objects...
splitting Images with 4 objects...
splitting Images with 5 objects...
```

Uploading the data to AWS S3

```
[ ]: %%capture
     ! aws s3 sync train_data/processed/ s3://sagemaker-us-east-1-031699854505/data/
```

Additional data processing is performed in the training scripts, where the data is augmented(only train data) and normalized, before going through the model.

Figure 6 - Data Processing 2



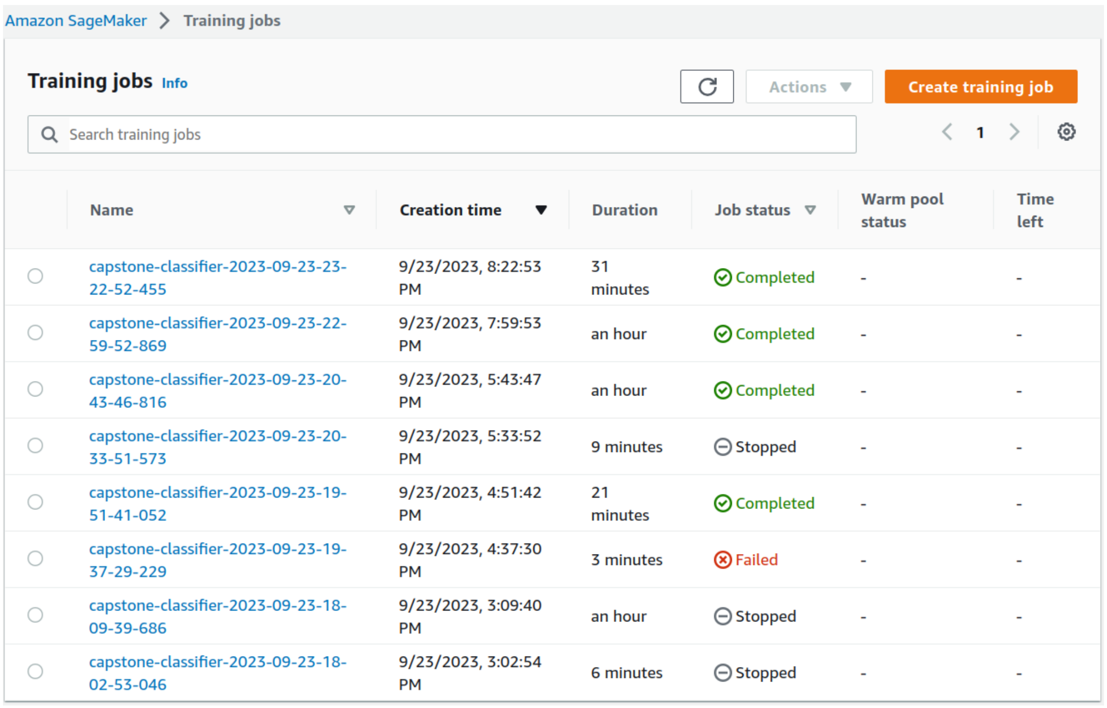
```
    # defining data augmentation and transformations for each dataset version
    data_transforms = {
        'train': transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.RandomRotation(10),
            transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'valid': transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'test': transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])
    }

    datasets = {version: torchvision.datasets.ImageFolder(os.path.join(args.data_dir, version),
transform=data_transforms[version]) for version in ['train', 'valid', 'test']}

    # creating data loaders
    dataloaders = create_data_loaders(datasets , args.batch_size)
    train_loader = dataloaders['train']
    valid_loader = dataloaders['valid']
    test_loader = dataloaders['test']
```

As you can see in the above screenshot, images in the validation and testing sets are only resized and cropped, but images in the training set go through some data augmentations, like rotation and HorizontalFlip, that can increase the model's generalization capabilities

3.3 Implementation and Refinement

The model training was performed using sagemaker estimators. Experiments with different configurations of hyperparameters, profiling, and debugger rules, instance configurations were performed.

Figure 7 - Training jobs run

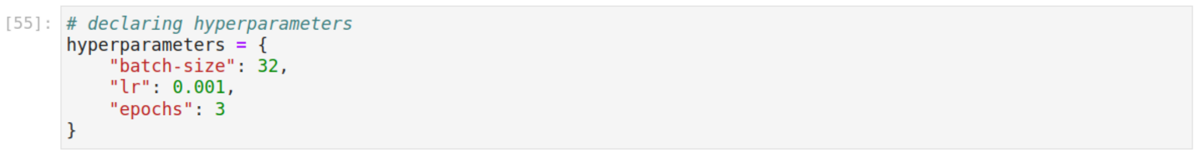


Name	Creation time	Duration	Job status	Warm pool status	Time left
capstone-classifier-2023-09-23-23-22-52-455	9/23/2023, 8:22:53 PM	31 minutes	Completed	-	-
capstone-classifier-2023-09-23-22-59-52-869	9/23/2023, 7:59:53 PM	an hour	Completed	-	-
capstone-classifier-2023-09-23-20-43-46-816	9/23/2023, 5:43:47 PM	an hour	Completed	-	-
capstone-classifier-2023-09-23-20-33-51-573	9/23/2023, 5:33:52 PM	9 minutes	Stopped	-	-
capstone-classifier-2023-09-23-19-51-41-052	9/23/2023, 4:51:42 PM	21 minutes	Completed	-	-
capstone-classifier-2023-09-23-19-37-29-229	9/23/2023, 4:37:30 PM	3 minutes	Failed	-	-
capstone-classifier-2023-09-23-18-09-39-686	9/23/2023, 3:09:40 PM	an hour	Stopped	-	-
capstone-classifier-2023-09-23-18-02-53-046	9/23/2023, 3:02:54 PM	6 minutes	Stopped	-	-

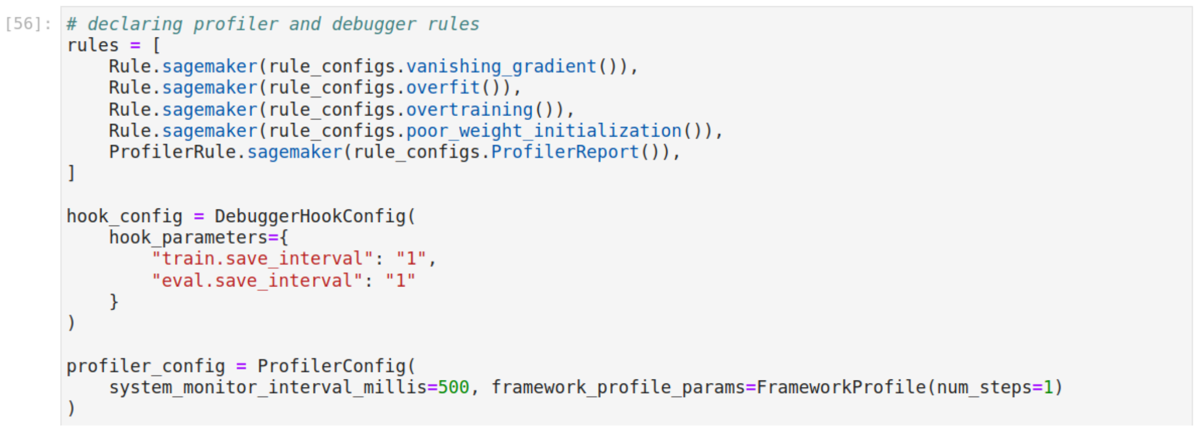
The workflow is as follows:

1. Before starting the training itself, it is necessary to set hyperparameters values, debugger, and profiler rules and configurations

Figure 8 - hyperparameters and rules configuration



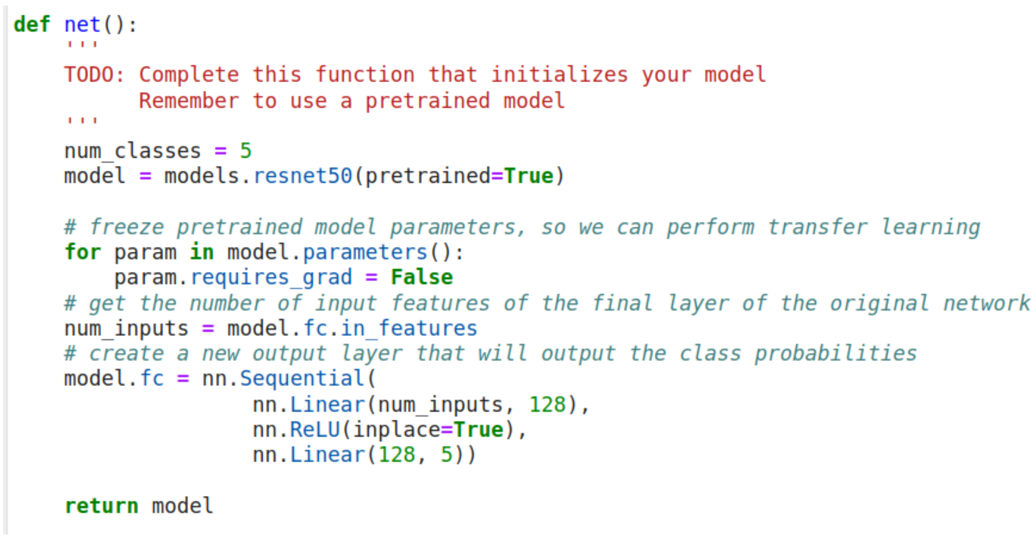
```
[55]: # declaring hyperparameters
      hyperparameters = {
          "batch-size": 32,
          "lr": 0.001,
          "epochs": 3
      }
```



```
[56]: # declaring profiler and debugger rules
      rules = [
          Rule.sagemaker(rule_configs.vanishing_gradient()),
          Rule.sagemaker(rule_configs.overfit()),
          Rule.sagemaker(rule_configs.overtraining()),
          Rule.sagemaker(rule_configs.poor_weight_initialization()),
          ProfilerRule.sagemaker(rule_configs.ProfilerReport()),
      ]

      hook_config = DebuggerHookConfig(
          hook_parameters={
              "train.save_interval": "1",
              "eval.save_interval": "1"
          }
      )

      profiler_config = ProfilerConfig(
          system_monitor_interval_millis=500, framework_profile_params=FrameworkProfile(num_steps=1)
      )
```

2. Prepare the training script. The entry point of this project is located in a file called train.py. In this file, you can find the model definitions, train and test loops, and data augmentation and normalization pipelines.

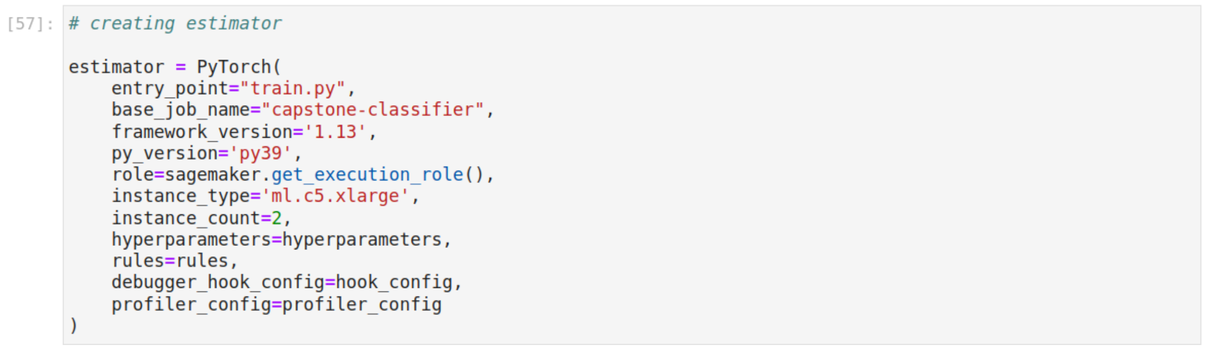
Figure 9 - Model architecture



```
def net():
    '''
    TODO: Complete this function that initializes your model
          Remember to use a pretrained model
    '''
    num_classes = 5
    model = models.resnet50(pretrained=True)

    # freeze pretrained model parameters, so we can perform transfer learning
    for param in model.parameters():
        param.requires_grad = False
    # get the number of input features of the final layer of the original network
    num_inputs = model.fc.in_features
    # create a new output layer that will output the class probabilities
    model.fc = nn.Sequential(
                   nn.Linear(num_inputs, 128),
                   nn.ReLU(inplace=True),
                   nn.Linear(128, 5))

    return model
```

3. Create the estimator. This is where you specify your entry point, runtime environment, number of instances where the training will happen, the type of instance, and other training parameters like profiler rules, debugger rules, and hyperparameters. Note that I am performing Multi-instance training because the job is run in two ml.c5.xlarge instances.

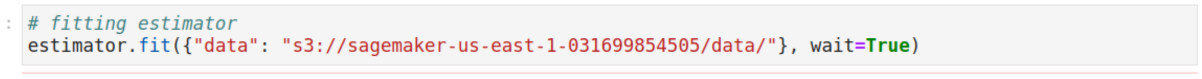
Figure 10 - Estimator creation



```
[57]: # creating estimator

      estimator = PyTorch(
          entry_point="train.py",
          base_job_name="capstone-classifier",
          framework_version='1.13',
          py_version='py39',
          role=sagemaker.get_execution_role(),
          instance_type='ml.c5.xlarge',
          instance_count=2,
          hyperparameters=hyperparameters,
          rules=rules,
          debugger_hook_config=hook_config,
          profiler_config=profiler_config
      )
```

4. Fit the estimator

Figure 11 - Fitting estimator

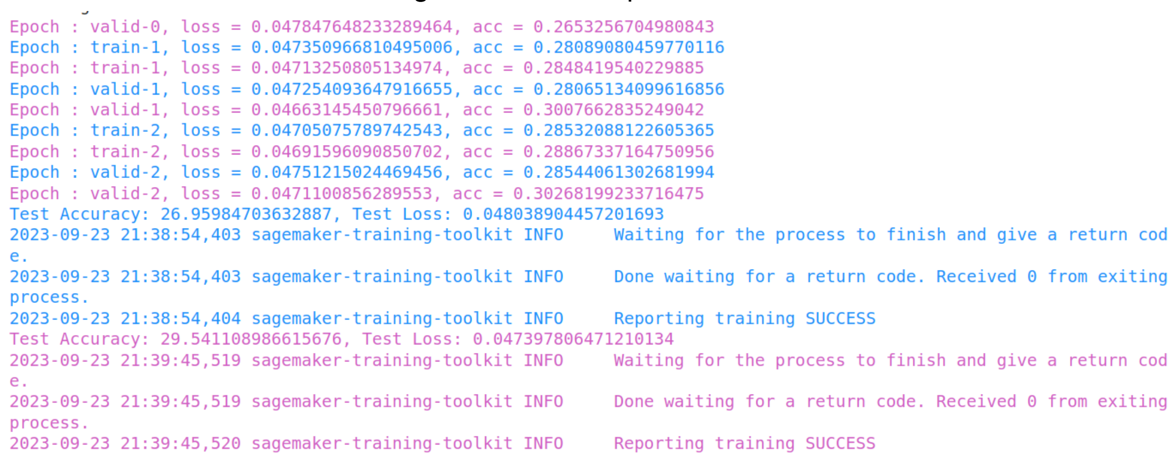


```
: # fitting estimator
  estimator.fit({"data": "s3://sagemaker-us-east-1-031699854505/data/"}, wait=True)
```

4. Results

4.1 Training results

Figure 12 - model performance



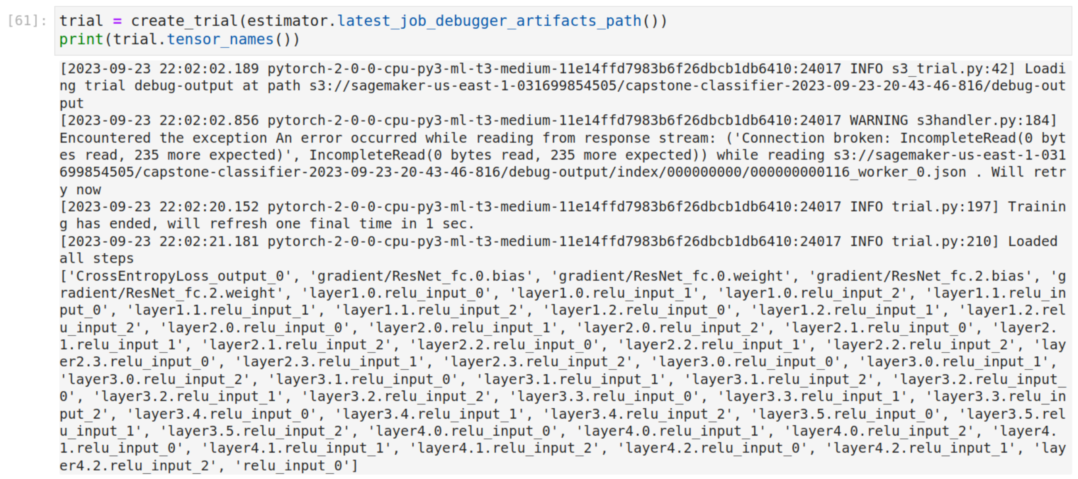
```
Epoch : valid-0, loss = 0.047847648233289464, acc = 0.2653256704980843
Epoch : train-1, loss = 0.047350966810495006, acc = 0.28089080459770116
Epoch : train-1, loss = 0.04713250805134974, acc = 0.2848419540229885
Epoch : valid-1, loss = 0.047254093647916655, acc = 0.28065134099616856
Epoch : valid-1, loss = 0.04663145450796661, acc = 0.3007662835249042
Epoch : train-2, loss = 0.04705075789742543, acc = 0.28532088122605365
Epoch : train-2, loss = 0.04691596090850702, acc = 0.28867337164750956
Epoch : valid-2, loss = 0.04751215024469456, acc = 0.28544061302681994
Epoch : valid-2, loss = 0.0471100856289553, acc = 0.30268199233716475
Test Accuracy: 26.95984703632887, Test Loss: 0.048038904457201693
2023-09-23 21:38:54,403 sagemaker-training-toolkit INFO     Waiting for the process to finish and give a return cod
e.
2023-09-23 21:38:54,403 sagemaker-training-toolkit INFO     Done waiting for a return code. Received 0 from exiting
process.
2023-09-23 21:38:54,404 sagemaker-training-toolkit INFO     Reporting training SUCCESS
Test Accuracy: 29.541108986615676, Test Loss: 0.047397806471210134
2023-09-23 21:39:45,519 sagemaker-training-toolkit INFO     Waiting for the process to finish and give a return cod
e.
2023-09-23 21:39:45,519 sagemaker-training-toolkit INFO     Done waiting for a return code. Received 0 from exiting
process.
2023-09-23 21:39:45,520 sagemaker-training-toolkit INFO     Reporting training SUCCESS
```

As can be seen, The model does not do a great job at classifying the images. The test set accuracy is only about 27%. Resnet is mainly a feature extractor model. In order to achieve higher performance, a more complex architecture like FasterRCNN should be used, once identifying the number of objects in an image is closer to an object detection task. Since the focus of this project is not on producing a highly accurate model, but rather on constructing the training pipeline using Sagemaker, I will leave FasterRCNN transfer-learning as an exercise for the reader.

4.3 Profiling and Debuging

In order to evaluate possible improvements to the training pipeline, I have attached debugger and training rules to the estimator (see the above prints). Sagemaker debugger allows us to monitor the weights, biases, and gradients tensors through the training process:

Figure 13 - Sagemaker debugger monitored tensors

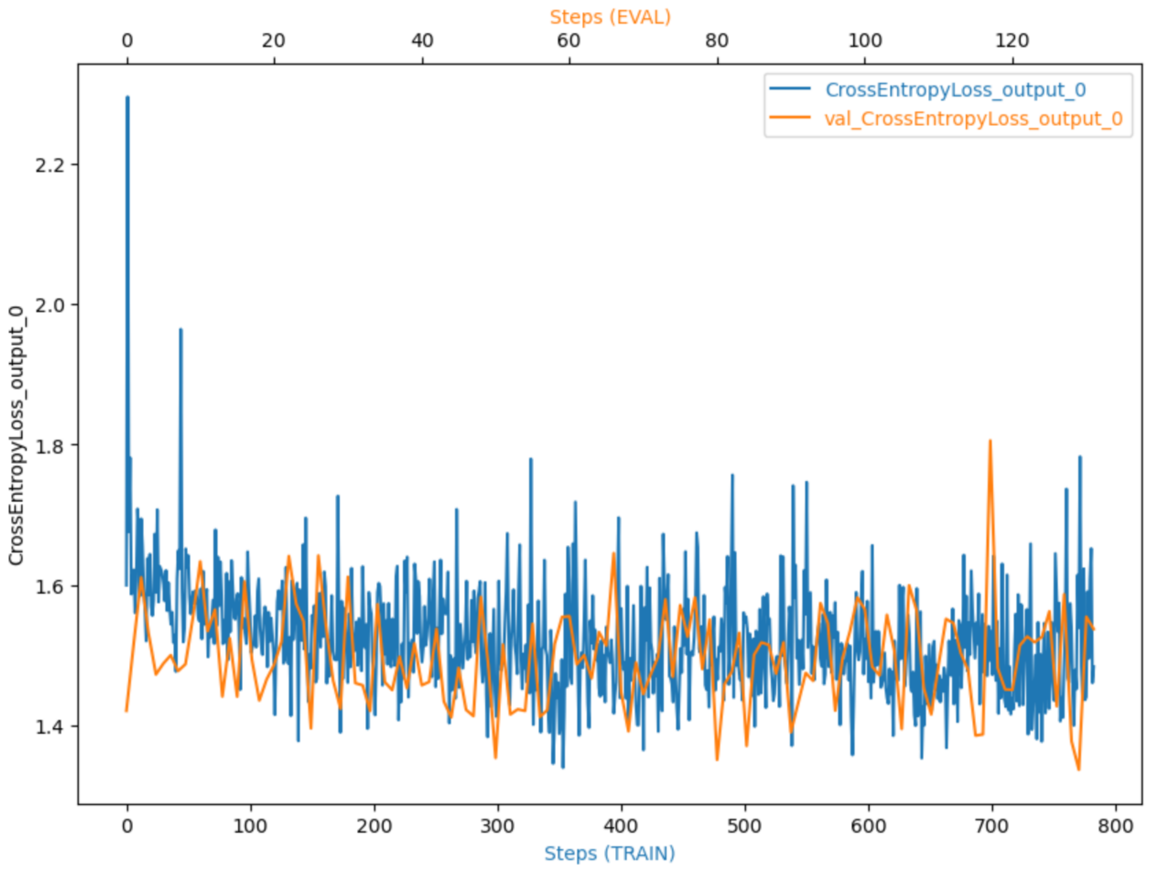


```
[61]: trial = create_trial(estimator.latest_job_debugger_artifacts_path())
      print(trial.tensor_names())
```

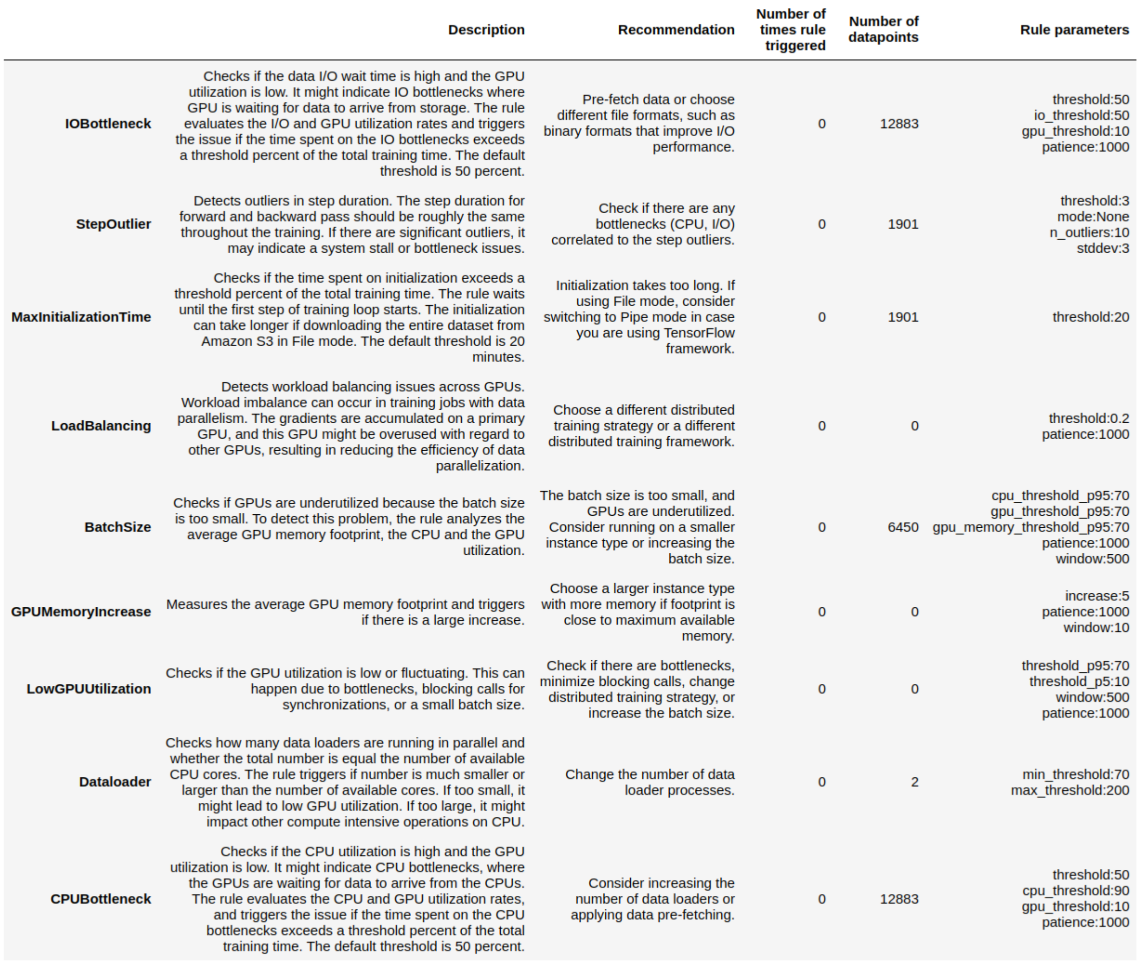
```
[2023-09-23 22:02:02.189 pytorch-2-0-0-cpu-py3-ml-t3-medium-11e14ffd7983b6f26dbcb1db6410:24017 INFO s3_trial.py:42] Loadi
ng trial debug-output at path s3://sagemaker-us-east-1-031699854505/capstone-classifier-2023-09-23-20-43-46-816/debug-out
put
[2023-09-23 22:02:02.856 pytorch-2-0-0-cpu-py3-ml-t3-medium-11e14ffd7983b6f26dbcb1db6410:24017 WARNING s3handler.py:184]
Encountered the exception An error occurred while reading from response stream: ('Connection broken: IncompleteRead(0 byt
es read, 235 more expected)', IncompleteRead(0 bytes read, 235 more expected)) while reading s3://sagemaker-us-east-1-031
699854505/capstone-classifier-2023-09-23-20-43-46-816/debug-output/index/000000000/000000000116_worker_0.json . Will retr
y now
[2023-09-23 22:02:20.152 pytorch-2-0-0-cpu-py3-ml-t3-medium-11e14ffd7983b6f26dbcb1db6410:24017 INFO trial.py:197] Trainin
g has ended, will refresh one final time in 1 sec.
[2023-09-23 22:02:21.181 pytorch-2-0-0-cpu-py3-ml-t3-medium-11e14ffd7983b6f26dbcb1db6410:24017 INFO trial.py:210] Loaded
all steps
['CrossEntropyLoss_output_0', 'gradient/ResNet_fc.0.bias', 'gradient/ResNet_fc.0.weight', 'gradient/ResNet_fc.2.bias', 'g
radient/ResNet_fc.2.weight', 'layer1.0.relu_input_0', 'layer1.0.relu_input_1', 'layer1.0.relu_input_2', 'layer1.1.relu_in
put_0', 'layer1.1.relu_input_1', 'layer1.1.relu_input_2', 'layer1.2.relu_input_0', 'layer1.2.relu_input_1', 'layer1.2.rel
u_input_2', 'layer2.0.relu_input_0', 'layer2.0.relu_input_1', 'layer2.0.relu_input_2', 'layer2.1.relu_input_0', 'layer2.
1.relu_input_1', 'layer2.1.relu_input_2', 'layer2.2.relu_input_0', 'layer2.2.relu_input_1', 'layer2.2.relu_input_2', 'lay
er2.3.relu_input_0', 'layer2.3.relu_input_1', 'layer2.3.relu_input_2', 'layer3.0.relu_input_0', 'layer3.0.relu_input_1',
'layer3.0.relu_input_2', 'layer3.1.relu_input_0', 'layer3.1.relu_input_1', 'layer3.1.relu_input_2', 'layer3.2.relu_input_
0', 'layer3.2.relu_input_1', 'layer3.2.relu_input_2', 'layer3.3.relu_input_0', 'layer3.3.relu_input_1', 'layer3.3.relu_in
put_2', 'layer3.4.relu_input_0', 'layer3.4.relu_input_1', 'layer3.4.relu_input_2', 'layer3.5.relu_input_0', 'layer3.5.rel
u_input_1', 'layer3.5.relu_input_2', 'layer4.0.relu_input_0', 'layer4.0.relu_input_1', 'layer4.0.relu_input_2', 'layer4.
1.relu_input_0', 'layer4.1.relu_input_1', 'layer4.1.relu_input_2', 'layer4.2.relu_input_0', 'layer4.2.relu_input_1', 'lay
er4.2.relu_input_2', 'relu_input_0']
```

Accessing the tensors, we can, for instance, plot train and validation losses along the optimization steps:

Figure 14 - Loss Monitoring



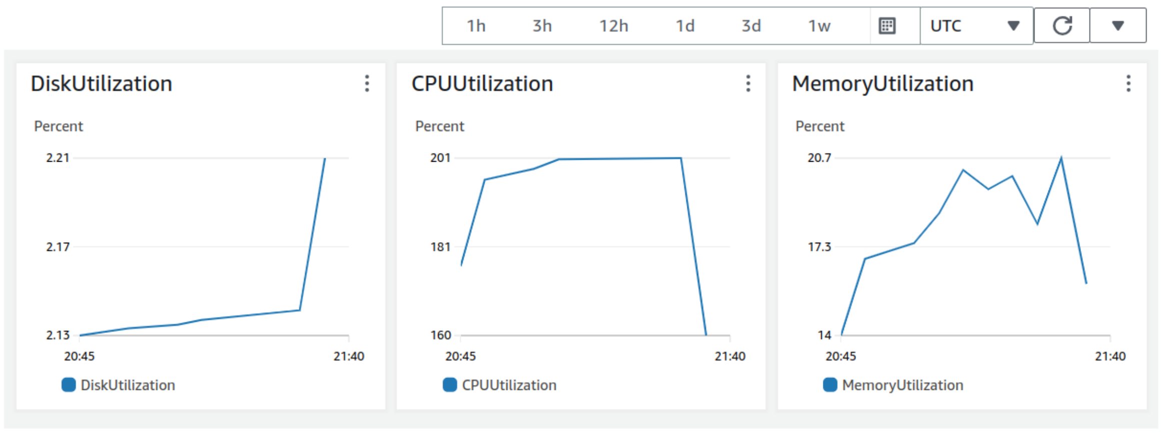
Sagemaker profiler generates a report saved in the same S3 bucket where the training model artifacts are saved. You can download it and check whether any of the alerts were triggered during the training process:



	Description	Recommendation	Number of times rule triggered	Number of datapoints	Rule parameters
IOBottleneck	Checks if the data I/O wait time is high and the GPU utilization is low. It might indicate IO bottlenecks where GPU is waiting for data to arrive from storage. The rule evaluates the I/O and GPU utilization rates and triggers the issue if the time spent on the IO bottlenecks exceeds a threshold percent of the total training time. The default threshold is 50 percent.	Pre-fetch data or choose different file formats, such as binary formats that improve I/O performance.	0	12883	threshold:50 io_threshold:50 gpu_threshold:10 patience:1000
StepOutlier	Detects outliers in step duration. The step duration for forward and backward pass should be roughly the same throughout the training. If there are significant outliers, it may indicate a system stall or bottleneck issues.	Check if there are any bottlenecks (CPU, I/O) correlated to the step outliers.	0	1901	threshold:3 mode:None n_outliers:10 stddev:3
MaxInitializationTime	Checks if the time spent on initialization exceeds a threshold percent of the total training time. The rule waits until the first step of training loop starts. The initialization can take longer if downloading the entire dataset from Amazon S3 in File mode. The default threshold is 20 minutes.	Initialization takes too long. If using File mode, consider switching to Pipe mode in case you are using TensorFlow framework.	0	1901	threshold:20
LoadBalancing	Detects workload balancing issues across GPUs. Workload imbalance can occur in training jobs with data parallelism. The gradients are accumulated on a primary GPU, and this GPU might be overused with regard to other GPUs, resulting in reducing the efficiency of data parallelization.	Choose a different distributed training strategy or a different distributed training framework.	0	0	threshold:0.2 patience:1000
BatchSize	Checks if GPUs are underutilized because the batch size is too small. To detect this problem, the rule analyzes the average GPU memory footprint, the CPU and the GPU utilization.	The batch size is too small, and GPUs are underutilized. Consider running on a smaller instance type or increasing the batch size.	0	6450	cpu_threshold_p95:70 gpu_threshold_p95:70 gpu_memory_threshold_p95:70 patience:1000 window:500
GPUMemoryIncrease	Measures the average GPU memory footprint and triggers if there is a large increase.	Choose a larger instance type with more memory if footprint is close to maximum available memory.	0	0	increase:5 patience:1000 window:10
LowGPUUtilization	Checks if the GPU utilization is low or fluctuating. This can happen due to bottlenecks, blocking calls for synchronizations, or a small batch size.	Check if there are bottlenecks, minimize blocking calls, change distributed training strategy, or increase the batch size.	0	0	threshold_p95:70 threshold_p5:10 window:500 patience:1000
Dataloader	Checks how many data loaders are running in parallel and whether the total number is equal the number of available CPU cores. The rule triggers if number is much smaller or larger than the number of available cores. If too small, it might lead to low GPU utilization. If too large, it might impact other compute intensive operations on CPU.	Change the number of data loader processes.	0	2	min_threshold:70 max_threshold:200
CPUBottleneck	Checks if the CPU utilization is high and the GPU utilization is low. It might indicate CPU bottlenecks, where the GPUs are waiting for data to arrive from the CPUs. The rule evaluates the CPU and GPU utilization rates, and triggers the issue if the time spent on the CPU bottlenecks exceeds a threshold percent of the total training time. The default threshold is 50 percent.	Consider increasing the number of data loaders or applying data pre-fetching.	0	12883	threshold:50 cpu_threshold:90 gpu_threshold:10 patience:1000

4.4 Cost Analysis

An essential point in industrialized machine-learning applications is to use the proper resources. Overprovisioning resources may lead to a substantial amount of money wasted. I monitored the instance metrics during the training process and found the following:



We can see that the maximum workloads for disk, CPU, and memory utilization were 2.21%, 20.1%, and 20.7%, respectively. These metrics point out that we could run our training jobs in smaller, cheaper instances if we were to orchestrate such a pipeline to run, for instance, once a week. Doing this would save some costs and make our bosses happier!

5. Conclusion

Throughout this project, AWS Sagemaker was used to build, experiment, and optimize different aspects of a machine-learning pipeline. It provides the tools for data exploration and data preprocessing as well as training, debugging, profiling, and monitoring.

These capabilities allowed the execution of jobs with different configurations and the evaluation of the strengths and weaknesses of the approach used for solving the problem. Besides that, it was showcased how to use tools like Sagemaker profiler, and Cloudwatch logs to optimize resource usage and save some costs.

Steps that could be explored in the future are:
 Model deployment
 Training with a more suited neural network architecture.