ML Capstone report

Sebastian Aranguiz
Dec. 22nd, 2023

Machine Learning Engineer Nanodegree

Definition

Project Overview

Distribution centers are increasingly relying on robotics to streamline their operations, and many of these robots are now equipped with cameras that can capture images of objects being shipped. However, accurately counting objects in bins remains a difficult task for distribution centers. Inaccurate inventory counts may lead to stockouts, overstocking, and discrepancies in delivery requests. Despite the increasing reliance on robotics for inventory management, accurate object counting in bins remains a critical challenge.

In the realm of machine learning (ML), pipelines play a crucial role in automating and streamlining the entire ML workflow, from data preparation to model deployment and monitoring. This project aims to shed light on the process of constructing an ML pipeline using AWS SageMaker, a comprehensive cloud-based platform for ML development and deployment. A substantial dataset, gathered from a real-world fulfillment center, serves as the pipeline's input. This dataset is employed throughout the training, validation and testing stages.

Problem Statement

There is a need for automated object counting systems that can accurately count items in bins at distribution centers. Such systems would utilize computer vision techniques to process images of bins and determine the number of objects present in it.

One approach is using image-based methods, which employ machine learning algorithms to analyze two-dimensional images. The algorithms are trained on labeled datasets to identify and count objects in images [1].

To achieve reliable predictions for any input image, the development of an ML pipeline is crucial. Several considerations must be addressed, including evaluating the data to determine if any preprocessing is necessary prior to training and selecting a pre-trained image prediction model to fine-tune with our dataset.

Metrics

To assess the effectiveness of the ML pipeline in developing an accurate object counting system, accuracy will be employed as the most relevant metric.

Accuracy is a measure of how well a model performs. In simple terms, accuracy is the proportion of predictions that the model makes correctly [2]. Formally, accuracy is defined as:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Accuracy is the most commonly used metric to evaluate classification models, including object counting models [3]. In this project, accuracy was calculated for the training, validation, and testing datasets, providing a comprehensive evaluation of the model's performance across different datasets.
Employing accuracy as the main metric is well-aligned with the project's objective of developing an accurate object counting system. Accuracy directly measures the model's ability to make correct predictions, which is crucial for ensuring the reliability of object counting.
In addition to this quantitative metric, visual inspection of model predictions was also conducted during the project development to identify any potential misclassifications or anomalies. This qualitative assessment will complement the accuracy measurements and provide a more holistic understanding of the model's performance.

Analysis

Data Exploration and Visualization

The project utilizes a subset of the Amazon Bin Image Dataset [4], which consists of a large set of images (500,000) of bins containing one or more items. Due to computing power constraints, a smaller subset of the dataset was actually used for training.
Each image is associated with metadata, including the number, dimensions, and types of objects. Our primary focus, nevertheless, lies in identifying the number of items within each bin (1 to 5 objects).
A Python script was provided to download a reduced set of 6,140 images and categorize the images into folders based on their class. The five classes represent the number of items being packed for delivery in a warehouse.
To gain insights into the data distribution, identify patterns, and detect anomalies, exploratory data analysis (EDA) was conducted on the reduced dataset. The analysis was inspired by a blog on EDA for image classification [5].
The key aspects of the analysis included:

Random sampling of images

A quick visual verification of the dataset was done with a script employed for visualizing random sample images from each class.

After reviewing several samples and getting familiar with the dataset, it was evident that most of the images of items captured the bin entirely and took up the whole image size. In addition, it was challenging to determine the exact number of items present in the bin. Here a few examples of a sampling of the class 4:

Figure 1: Sampling of images for a specific class

Image size

The image size was plotted to gain a better understanding of the raw data. A specific script was used to graph the width and height in pixel values. The analysis revealed that the minimum height for each class was slightly over 200 pixels, and the minimum width was 300 pixels for all classes. These are the visualizations per class:

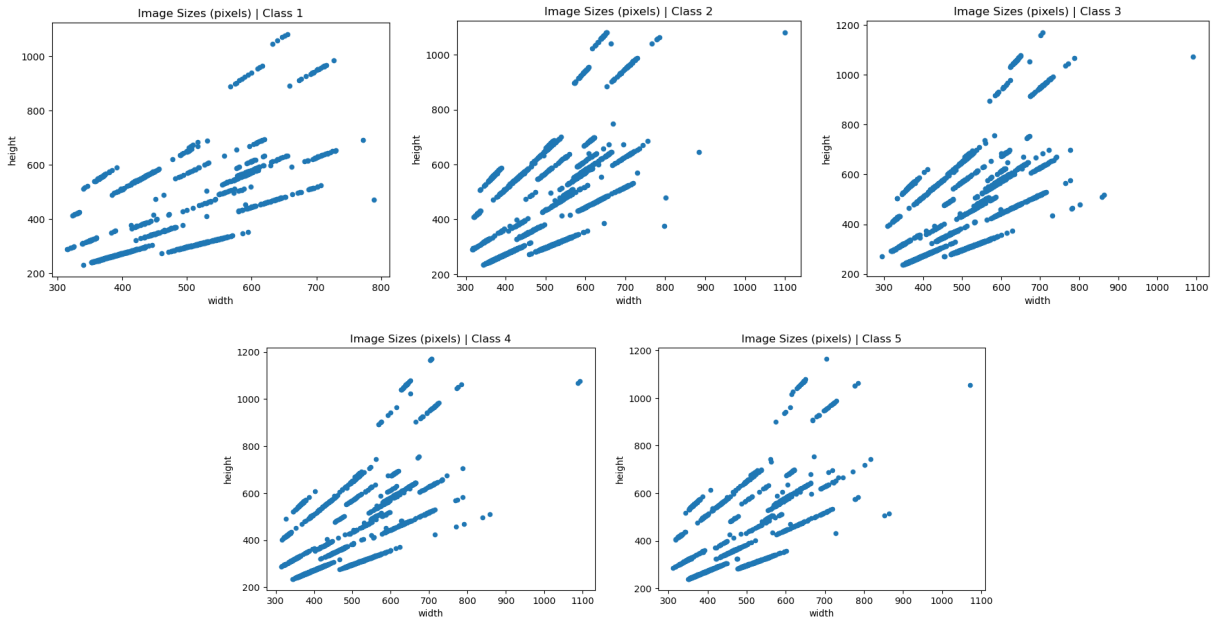

Figure 2: Image size in pixels by class

Class imbalance

It was crucial to determine if the five classes were equally represented. This was not the case, as the number of images per class varied significantly. An imbalanced dataset can lead to biased predictions, overfitting, and inaccuracy. The class imbalance is shown in the following bar chart:

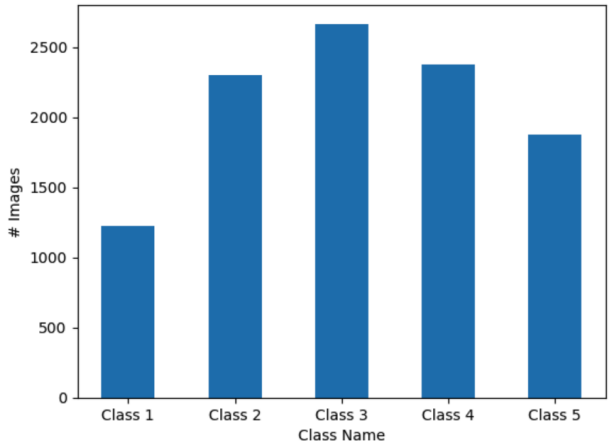

Figure 3: Number of images by class

By understanding these three aspects of the dataset, informed decisions about data preprocessing and model training can be made. For more details about the scripts implementation, execution and results, check out the *Dataset_EDA.ipynb* Jupyter Notebook.

Algorithms and Techniques

The primary objective of counting objects in images was addressed by utilizing PyTorch, a versatile ML framework commonly employed for computer vision applications. The project's scripts were built upon an image classifier implemented in PyTorch that leverages pre-trained models, enabling fine-tuning on a custom dataset tailored to the task of object counting.

To effectively optimize hyperparameters during the training phase, AWS SageMaker was employed. Additionally, this tool was utilized for thorough analysis and debugging of both the training and evaluation processes.

The project's core algorithm, *train.py*, was designed for training a convolutional neural network (CNN) for image classification. The code implements a supervised learning approach and fine-tunes the model on a custom dataset of images for a 5-class classification task.

Here's a breakdown of the main steps involved:

1. Data Loading and Preprocessing:

    a. The *create_data_loaders* function creates data loaders for training, validation, and testing sets.
    b. It applies transformations like resizing, random horizontal flipping, and converting images to tensors.
2. Model Architecture and Initialization:
    a. The *net* function defines the CNN model architecture.
    b. It uses a pre-trained model and freezes most of its layers, keeping the last fully connected layer for classification.
3. Loss Function and Optimization:
    a. The criterion variable defines the cross-entropy loss function for multi-class classification.
    b. The optimizer variable uses the Adam optimizer to optimize the weights of the last fully connected layer.
4. Training and Validation:

    The *train* function iterates over the training data multiple times (epochs) and performs the following steps:

    i. Forward pass: The model processes the input images and generates predictions.
    ii. Loss calculation: The loss function measures the accuracy of the predictions.
    iii. Backward pass: The gradients of the loss with respect to the model parameters are computed.
    iv. Weight update: The optimizer updates the weights using the calculated gradients.
    v. Validation: The model's performance is evaluated on the validation data.
5. Testing and Evaluation:
    a. The *test* function evaluates the trained model on the testing data by calculating the loss and accuracy.
    b. It prints the overall performance metrics.

In accordance with the distinct phases of the pipeline development, three script variants were developed and tailored to their specific roles:

- *train.py*

This is a base script that can run locally in any machine and is the starting point for running experiments with a minimal dataset. The main goal of this script was to make sure that: training, validation and testing algorithms work well with the dataset; models can be saved; and that raw predictions can be made by the model. With a lightweight version of the dataset, iterations over the model training are very quick and one gets familiar with the training algorithm for future debugging, fixes or enhancements. Original source: pytorch-image-classifier (other of my personal projects).

- *hpo.py*

Script to be used specifically for Hyperparameter Optimization. It is based on *train.py* and has the ability to access environment variables that AWS Sagemaker uses internally. Also, a few adjustments have been made to add logging functionality and work with AWS Sagemaker configuration and its cloud environment.

- *train_debug.py*

Script to be used specifically for Debugging and Profiling. It's based on *hpo.py* and implements hooks for debugging and profiling both training and validation phases.

These scripts can be used together with any of the computer vision pre-trained models available in the Pytorch repository. See a full list on the Pytorch website.

Benchmark

ResNet is a well-known pre-trained model for understanding images, particularly in tasks like sorting pictures into categories. In this project, however, a newer model called **ResNeXt** was explored, which claimed to have better results. This model was introduced in the "*Aggregated Residual Transformations for Deep Neural Networks*" paper [6] and uses smart design principles and specific settings to potentially improve how we find objects in pictures.

Research has shown that the detection of objects in images may be significantly enhanced by adopting ResNeXt. This improvement is attributed to the model's distinctive use of cardinality, employing a homogeneous design and a multibranch architecture characterized by a parsimonious set of hyperparameters [7].

Thus, a quick benchmark was conducted as an experimental prototype to confirm the better performance of ResNeXt (specifically, the *resnext101-32x8d version)* against a traditional *resnet50* model. The implementation of the benchmark was realized on AWS Sagemaker JumpStart (see more details in the *Models_benchmark.ipynb* notebook).

Following the model training and assessing outcomes over 10 epochs, the following results were observed:

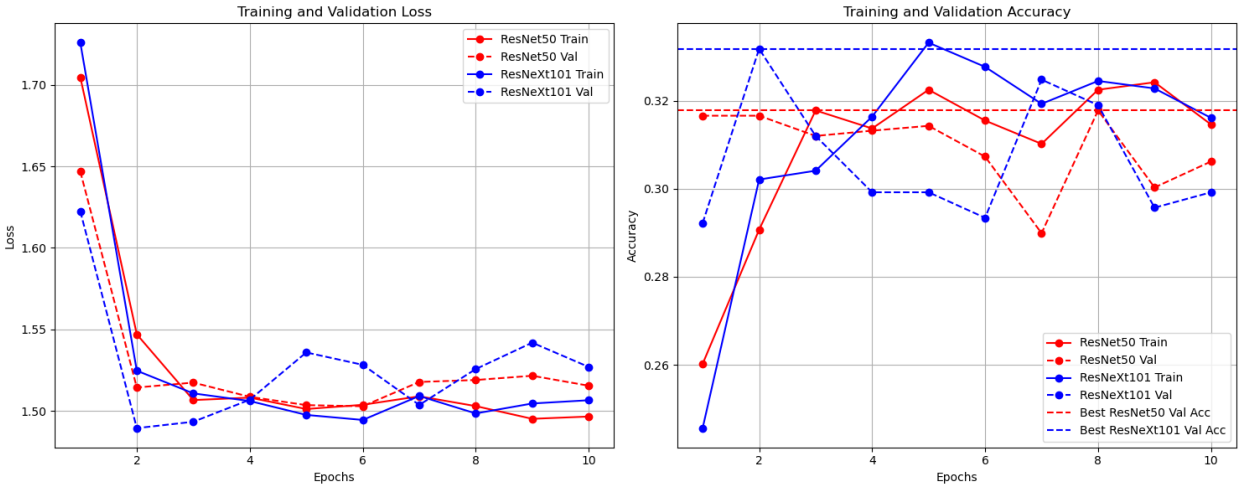

Figure 4: Training/validation loss and accuracy

We see a general trend of decreasing training and validation loss for both models, especially until the 2-3 epochs. The loss values are in a reasonable range, showing that both models are learning effectively from the training data.
To determine which model performs better, we can compare the best validation accuracy achieved by each model. These were the final results:

- ResNet50 achieved the best validation accuracy of approximately 0.3178.
- ResNeXt101 achieved the best validation accuracy of approximately 0.3318.

Therefore, if we focus on the best validation accuracy, ResNeXt101 outperformed ResNet50.

Methodology

Data Preprocessing

In response to the key takeaways from the EDA, the following measures were decided in order to pre-process the images for the subsequent training, validation and testing steps.

Random sampling of images

A thorough examination of the dataset revealed that images typically captured the full bin with items, making cropping an unnecessary and potentially detrimental step.

Image size

Based on the visualizations, it was determined that the minimum height for each class is 200 pixels and the minimum width is 300 pixels. Given these dimensions, resizing all images to a uniform size of 224 x 224 pixels would be a suitable approach for training,

validation, and testing purposes. This source was also useful for determining the selected image resolution. Resizing was incorporated into the transform steps during the creation of data loaders for the training job.

Class imbalance

To address the imbalanced nature of the dataset, several techniques can be employed

- Undersampling: This method involves removing samples from the majority class until the classes are evenly distributed.
- Oversampling: In contrast, oversampling involves increasing the number of samples in the minority classes.
- Augmentation: This technique involves generating new artificial images from existing ones to increase the diversity of the dataset.

The classes 1 and 5 are currently underrepresented in the dataset. To simplify the process, we will only employ the undersampling technique to maintain a consistent number of 1,228 images per class, matching the number of images in class 1. In practice, the Amazon Bin dataset is a vast repository of over 500,000 images, allowing for further data addition, if the project requires scaling and additional computing resources.

To achieve a fixed number of images per class and split between training, validation, and testing, the split-folders library was utilized in a 70%-20%-10% proportion.

Implementation

The benchmark marked a solid starting point for initiating the training of the ResNeXt model. Initially, the learning rate was set to the default value assigned by Sagemaker JumpStart, and a batch size of *5* was also set. However, it remained uncertain whether these hyperparameter values were optimal for fine-tuning the model on our dataset at this stage.

Consequently, hyperparameter optimization was performed using the AWS Sagemaker SDK with the objective of minimizing the average test loss. After doing some research, appropriate ranges for the hyperparameters were selected. Eight instances were executed to explore the most effective combination of learning rate and batch size for the validation step.

A few challenges arose during the implementation of hyperparameter tuning, including:

- **Selection of the instance type**, which needed to be at least: *ml.m5.2xlarge* (8 vCPU, 32 GiB of memory). Attempting to run training jobs with instances offering fewer resources (which could be more cost-effective) led to memory issues.
- **Images dataset size**, comprising 6,140 images, a total of 341 MB. Each epoch required approximately 20 minutes to execute, and computational resources were unfortunately limited.

- **Number of epochs**. Determining the number of epochs became an important decision, considering the constrained resources. One cost-saving alternative available was the use of "spot instances," although this often led to training interruptions, requiring a restart from the beginning.

Due to these challenges and keeping the cost factor in mind, it was decided that the hyperparameter optimization phase was trained for only two epochs. The results were the following:

	lr	train-batch-size	TrainingJobName	TrainingJobStatus	FinalObjectiveValue	TrainingStartTime	TrainingEndTime	TrainingElapsedTimeSeconds
0	0.005964	"64"	pytorch-training-231211-0942-008-fc105588	Completed	1.6096	2023-12-11 10:29:14+00:00	2023-12-11 11:12:00+00:00	2566.0
7	0.004065	"256"	pytorch-training-231211-0942-001-ba8463c3	Completed	1.5544	2023-12-11 09:43:42+00:00	2023-12-11 10:28:29+00:00	2687.0
5	0.008156	"64"	pytorch-training-231211-0942-003-1f216aa6	Completed	1.5325	2023-12-11 09:43:39+00:00	2023-12-11 10:27:11+00:00	2612.0
3	0.007681	"128"	pytorch-training-231211-0942-005-6e08a23f	Completed	1.5163	2023-12-11 10:24:33+00:00	2023-12-11 11:09:50+00:00	2717.0
2	0.001000	"32"	pytorch-training-231211-0942-006-ea2d02a1	Completed	1.4858	2023-12-11 10:27:21+00:00	2023-12-11 11:04:11+00:00	2210.0
6	0.007023	"32"	pytorch-training-231211-0942-002-64e034ca	Completed	1.4621	2023-12-11 09:43:34+00:00	2023-12-11 10:21:45+00:00	2291.0
4	0.009897	"64"	pytorch-training-231211-0942-004-5985befd	Completed	1.4602	2023-12-11 09:43:51+00:00	2023-12-11 10:29:03+00:00	2712.0
1	0.003746	"32"	pytorch-training-231211-0942-007-01841cc6	Completed	1.4424	2023-12-11 10:28:41+00:00	2023-12-11 11:04:45+00:00	2164.0

Refinement

Achieving the optimal combination for learning rate and batch size, among the eight possibilities, enabled us to proceed with refining the model and attempting to enhance its performance. In the next phase, debugging and profiling were undertaken to analyze the training process and pinpoint potential issues in its execution. This analysis was also performed using AWS Sagemaker, and the model training was extended to three epochs.

Following the implementation of necessary "hooks" into the training script, the debugging output revealed the following results during the execution of three epochs:

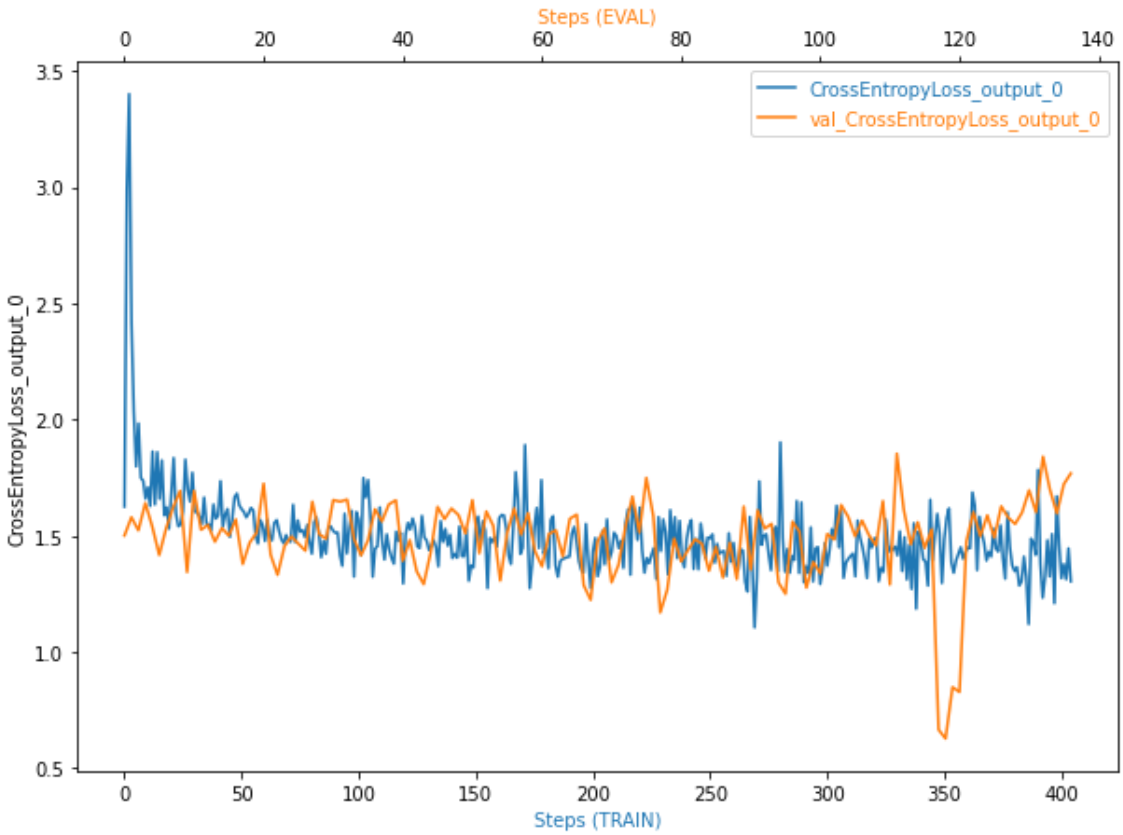

Figure 5: training and validation loss per steps

The graph shows a decreasing trend in cross-entropy loss throughout the training period, indicating that the model is slowly improving in its ability to accurately predict the correct labels. Some fluctuations in the loss value are observed, which may be typical during the training phase as the model refines its predictions. These variations are expected, considering that the model is still in the learning process and may not consistently make perfect predictions.
Overall, the graph suggests that the model is learning well and is on track to achieve good performance on the training data.

Some of the issues reported during debugging and profiling:

ProfilerReport

The rule *StepOutlier* was triggered 28 times, according to the profiling report below. The AWS documentation states this rule helps detect atypical step durations, specifically those larger than n-times the standard deviation of the whole step durations in a time interval. I believe this may be caused by the type of instance used for training the model, which was *ml.m5.2xlarge* (8 vCPU, 32 GiB). In a new training round, in order to solve this issue, I would pick a more powerful instance type, with GPU and monitor closely potential bottlenecks around the processor and I/O.

	Description	Recommendation	Number of times rule triggered	Number of datapoints	Rule parameters
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.	28	581	threshold:3 mode:None n_outliers:10 stddev:3

Overtraining

As discussed in the graph of the cross-entropy loss, it seems that the model is slowly improving his performance in every training step. However, even though the loss is decreasing, the drop is not substantial—it is rather small. Thus, the rule triggers because it thinks the model is not improving anymore. As a potential solution for a further debug iteration, I would review and adjust accordingly the parameters set for the overtraining rule. On the other hand, it may be true that the model does not improve anymore and the loss may start increasing instead of decreasing, clear signs of overfitting. In this case I would set up an early stopping strategy and a deeper analysis would need to be conducted to find out why overfitting occurs.

PoorWeightInitialization

There were issues found for this rule. The technique used for weight initialization needs to be reviewed. Here is what AWS states [8] about this issue:

"Good initialization breaks the symmetry of the weights and gradients in a neural network and maintains commensurate activation variances across layers. Otherwise, the neural network doesn't learn effectively... Too small an initialization can lead to vanishing gradients. Too large an initialization can lead to exploding gradients."

I believe a potential solution to overcome this issue would be trying to initialize the model loading it with *state_dict()* values of previous model training jobs.

Results

Model Evaluation and Validation

The main goal of the project was the implementation of an ML engineering pipeline, and not the accuracy of the final trained model. For this reason, after assessing the results of the debugging and profiling phase, the model was directly deployed to be available for inference.

After deployment on AWS Sagemaker, inference can be performed against a private endpoint. The proposed implementation supports sending a JSON object containing the url of the image to be used for the requested prediction.

There were three ways to invoke the endpoint to make predictions:

- **AWS Sagemaker SDK**. This was performed in the main Jupyter notebook. It required the serialization and deserialization of the JSON object mentioned above.

- **AWS Lambda function**. Having the JSON object with the url as input. The function invokes the deployed endpoint and returns the predictions as output.
- **AWS API Gateway**. In addition, an integration between the AWS lambda function and API Gateway made it possible for external users to make predictions by a public website especially built for this purpose.

After performing several requests, the overall accuracy of the predictions was “one out of three” correct, as the logs reported in the testing phase (at the end of training-validation):

```
Testing loss: 1.4319 #011 Testing Accuracy: 204/615 (33%)
```

Justification

Considering the results obtained in the benchmark stage, ResNeXt's best validation accuracy reached around 33% in just the second epoch, with a dataset of 4,295 images. It's noteworthy to point out that the accuracy did not improve after executing 10 epochs, and, in some cases, values were below 30% afterwards.
On the other hand, when retraining with optimized hyperparameters and a larger dataset of 6,140 images, the best validation accuracy also peaked at 33% in the second epoch. However, it decreased slightly to 32% in the third epoch.
Several points require closer analysis, particularly regarding the model's capacity to increase its accuracy. Here are a few observations that may help explain the model's performance:

- The model rapidly learned general patterns in the data, resulting in an early peak performance during training. However, subsequent epochs did not yield significant further improvement.
- The fact that accuracy didn't improve after the second epoch raises the possibility of overfitting. Running additional training epochs, perhaps exceeding 10, could provide more insights and confirm this hypothesis.
- The quality and diversity of the dataset may play a crucial role. Some images posed challenges even for me to determine the number of items captured. A well-curated dataset is essential for training a model that generalizes effectively across various scenarios.

Conclusions

In this Machine Learning Capstone project, the primary goal was to construct and deploy a robust ML pipeline for object counting in distribution center bins. The project focused on leveraging AWS SageMaker to develop, train, and deploy a ResNeXt-based model for accurate object counting. Below are the key findings drawn from the project:

Key Findings

1. Dataset Analysis

- A comprehensive analysis of the Amazon Bin Image Dataset revealed challenges such as class imbalance, image size variability, and difficulties in determining the exact number of items in some images.
- Insights gained from exploratory data analysis (EDA) influenced crucial decisions in data preprocessing and model training.

2. Algorithm and Techniques

- PyTorch, a powerful ML framework, was employed for implementing a Convolutional Neural Network (CNN) to address the object counting task.
- AWS SageMaker facilitated hyperparameter optimization, training, and evaluation of the model, streamlining the development process.

3. Benchmarking

- A benchmark experiment compared the performance of ResNeXt (specifically, resnext101-32x8d) against ResNet50.
- ResNeXt outperformed ResNet50 in terms of best validation accuracy, demonstrating its potential for object detection tasks.

4. Model Training and Enhancement

- The model training process involved refinement processes, such as debugging and profiling, to enhance its performance.
- Challenges, such as overtraining and poor weight initialization, were identified. If they are addressed, it may contribute to improved model training.

5. Deployment and Inference

- The trained ResNeXt model was deployed on AWS SageMaker for real-time inference.
- Multiple AWS services, including the SageMaker SDK, Lambda function, and API Gateway, were utilized for invoking the deployed model.

6. Model Evaluation

- The final model achieved a testing accuracy of approximately 33%, with room for further optimization.

Recommendations for Future Work

- **Hyperparameter Tuning**: Extending the hyperparameter optimization phase beyond two epochs might uncover additional insights. Choose to tune additional parameters.
- **Enhanced Data Quality**: Continued efforts in dataset curation, including diverse images, would likely improve model performance.
- **Addressing Overfitting**: Implementing an early stopping strategy and deeper analysis could mitigate potential overfitting, ensuring better model generalization.
- **Learning Rate**: Trying a new approach to dynamically adjust its value, employing techniques such as a learning rate schedule or adaptive learning rate.
- **Spot instances and multi-instances training**: Performing a cost analysis for the usage of spot instances. Splitting the train process using more instances.

In conclusion, while the project successfully established an end-to-end ML pipeline for object counting, ongoing refinement and exploration of various challenges are essential for achieving even higher accuracy and robustness in real-world scenarios. The insights gained from this project may serve as a foundation for professionals looking to learn how to build an ML pipeline with AWS SageMaker, as well as for future improvements and advancements in the field of object counting using ML.

References

[1] Liu, X., Ma, L., Zhang, T., & Feng, J. (2019). Object counting in images of distribution center bins using deep learning. In 2019 International Conference on Computer and Communication Systems (ICCCS) (pp. 1031-1035). IEEE.
[2] Google's Machine Learning Foundational course. Classification: Accuracy. https://developers.google.com/machine-learning/crash-course/classification/accuracy
[3] Dr. A. R. K. Reddy and Dr. R. K. Sharma (2021). Object Counting and Localization Using Deep Learning (page 172).
[4] Amazon Bin Image Dataset. Registry of Open Data on AWS. https://registry.opendata.aws/amazon-bin-imagery/
[5] D. Rausch (2021). EDA for Image Classification. https://medium.com/geekculture/eda-for-image-classification-dcada9f2567a
[6] Xie et al (2017). Aggregated Residual Transformations for Deep Neural Networks.
[7] Haryono et al (2023). Oriented object detection in satellite images using convolutional neural network based on ResNeXt. https://onlinelibrary.wiley.com/doi/10.4218/etrij.2022-0446
[8] AWS Sagemaker documentation - PoorWeightInitialization. https://docs.aws.amazon.com/sagemaker/latest/dg/debugger-built-in-rules.html#poor-weight-initialization