Brandy Griffin

Department of Science, Technology, Engineering & Math, Houston

ITAI 1378 Computer Vision

Fall 2024

Professor Patricia McManus

Sunday, November 17, 2024

Midterm Object Detection Challenge

Objective

Object detection is an exciting field in computer vision. It's all about making computers recognize and locate objects in pictures, which can be really useful in areas like sports analysis, tracking inventory in warehouses, and even smart city security. In this project, we had to build a decent object detection model without needing high-powered hardware. The model used was called SSD MobileNet V2, which is lightweight and efficient. To make it work better, techniques like transfer learning, data augmentation, and model tuning were used to get the best results possible. I did all of this in Google Colab, which offers free access to a GPU but has some limitations.

By selecting a small dataset and trying out various configurations, it turned out that even with fewer resources, a smaller model could still handle object detection quite well.

Introduction

Identifying and labeling objects in images is both challenging and intriguing. It requires detecting specific items and marking them, often by placing boxes around each object to make them stand out. You see it used in all kinds of places, like counting people in crowds, tracking animals in wildlife studies, or even in automated retail systems. However, training object detection models usually requires a lot of computing power. For my project, I wanted to see if I could get good results even on limited resources. I used Google Colab, which provides a free GPU, though it's not as powerful as what professionals use. This setup is similar to what many students and beginner researchers use since high-end resources aren't always available.

The main goals of this project were to make SSD MobileNet V2, a lightweight object detection model, as efficient as possible without sacrificing accuracy; to use techniques like transfer learning and data augmentation in a limited environment; and to keep track of my experiments so others could repeat them if they wanted to. By testing and tuning different parts of the model, I aimed to show that we can get good results even with some resource limitations.

Initial Setup and Baseline Model Review

I chose to use Google Colab for this project because it's accessible and has a free GPU. Colab is popular with students because it lets us work with deep learning without needing expensive computers. I started by setting up the coding environment, making sure all the necessary libraries like TensorFlow, Keras, numpy, and matplotlib were installed. Although Colab usually has these libraries pre-installed, I double-checked to make sure they were compatible, as Colab updates them from time to time, which can cause issues with custom code.

While setting up, I found that I needed to go into another notebook called "Copy of Snippets." This notebook had extra code I needed to run to make sure all the libraries were properly installed. This step was really helpful since it added any missing parts that the project might need and helped avoid compatibility issues. Running these snippets made everything run a lot smoother later.

For my model, I chose SSD MobileNet V2, which was available in a notebook called *Student_Notebook_LAB_Object_Detection_transfer_learning.ipynb*. SSD MobileNet V2 is known for being fast and efficient, making it great for mobile devices or systems with low power. SSD, which stands for Single Shot MultiBox Detector, can detect objects in one go, making it quick and not too memory heavy. MobileNet V2, the backbone of the model, has fewer parameters because it uses something called depthwise separable convolutions. By recording the initial training and validation accuracy, along with the loss values, I got a baseline for how the model performed before making any changes.

Dataset Selection and Preparation

Choosing the right dataset was important because I needed one that wouldn't overwhelm Colab's limited memory. I looked at a few open-source datasets, including PASCAL VOC 2007, COCO-minitrain, Oxford-IIIT Pet Dataset, and CIFAR-10. PASCAL VOC 2007 and COCO-minitrain are commonly used but can be heavy on resources. The Oxford-IIIT Pet Dataset includes images of 37 pet breeds, which could work well on Colab. But I ultimately chose CIFAR-10 because it's small and manageable, with 60,000 images across 10 categories. Even though CIFAR-10 is more for classification, its size made it great for quick experiments, letting me train the model faster without hitting Colab's memory limits.
Once I chose CIFAR-10, I did some Exploratory Data Analysis (EDA) to understand the dataset better. The EDA step was one of the longest and most challenging parts, as it involved loading and inspecting a significant amount of data to ensure each class was balanced and that the images were clear and consistent. Randomly sampling images from each category helped confirm there was enough variety in the classes. Verifying that data shapes matched the model's input requirements took some time, but it was an important step to ensure the dataset was correctly prepared for training.

Data Cleaning and Augmentation

Data augmentation was really important for making the model more adaptable, but it took a lot of time to set up. With CIFAR-10's small dataset, I wanted to make sure the model could handle variations. I used TensorFlow's *ImageDataGenerator* to create real-time variations of each image during training. Some techniques I used included:

- **Rotation**: Rotating images up to 20 degrees so the model could learn to recognize objects from different angles.
- **Shifting**: Moving images slightly in different directions to show objects in different positions.
- **Flipping**: Flipping images horizontally to give the model different orientations of the same objects.

These methods helped expand the dataset's diversity without taking up extra memory, which was important with Colab's limited resources. Running data augmentation in real-time added around 20–30 minutes per epoch, as each variation had to be generated and processed on the fly. It was a challenging process that required a lot of GPU power, but it made the model much more versatile.

Model Architecture and Transfer Learning

I chose SSD MobileNet V2 for this project because it balances efficiency and accuracy well. This model uses depthwise separable convolutions to reduce the number of parameters, making it faster and less demanding on memory. SSD (Single Shot MultiBox Detector) also allows the model to process images in one pass, making it faster than models that require multiple passes.

To make the model work better for CIFAR-10, I used transfer learning. MobileNet V2 was pre-trained on ImageNet, so I froze most of its layers to keep the general features it learned and then added custom layers on top to handle CIFAR-10's specific categories. This made training faster, as the model only needed to learn new patterns in the custom layers, without re-learning everything. To align with accuracy and efficiency goals, the model was compiled with a categorical cross-entropy loss function after the initial setup was complete.

Training Process and Performance Monitoring

In the training phase, real-time data augmentation was applied to create diverse image variations, which helped the model become better at generalizing across different types of data. I set aside part of the dataset for validation to monitor how well the model was learning. I also used *ModelCheckpoint* to save the best version of the model, so if the model's performance dropped in later stages, I could go back to the best-performing version without starting from scratch.

Training the model was one of the longest and most difficult steps in the project. Each training epoch took over an hour to complete, mainly because SSD MobileNet V2, despite being efficient, still has a lot of layers and parameters to process. The real-time data augmentation further added to the time, as the model had to handle transformed images constantly. Creating training and validation curves allowed me to keep an eye on how the model was learning and adjust if there were signs of overfitting or underfitting. Watching these graphs helped me catch any issues early. For example, if training accuracy kept going up while validation accuracy started to fall, it was a clear sign that the model was overfitting—memorizing the data instead of understanding the actual patterns. If both accuracies were low, it meant the model was underfitting. I made adjustments like adding dropout and using regularization to improve the model's generalization and prevent overfitting.

After training, I tested the model on a separate test set to see how it would perform on new data. High test accuracy meant the model was learning well and not just memorizing training data.

Hyperparameter Tuning and Advanced Evaluation

Far as adjusting the learning rate was key to getting the model to train effectively. Testing different rates—like 0.01, 0.001, and 0.0001—helped find the most suitable balance. Higher rates sped up learning but sometimes made the model unstable, while lower rates provided more stable training at a slower pace. Finding the right rate made a noticeable difference in how smoothly the model learned. I found that 0.001 was a good middle ground, allowing the model to learn steadily without slowing down too much.

To get more detailed feedback, I used a confusion matrix, which showed where the model made correct predictions and where it made mistakes. This matrix also revealed certain pairs of classes the model often mixed up, likely because they looked similar. These insights helped me improve the model by using more targeted data augmentation or by adding more examples of tricky classes.

Explanation of Code Execution Time and Common Errors

Some code sections, especially model training, data augmentation, and the tasks within EDA, took a long time to run. SSD MobileNet V2 is efficient, but each epoch could still take over an hour because of the model's complexity. Data augmentation was also time-intensive because it generated new image variations on the spot, adding 20-30 minutes per epoch. Tasks in the EDA markdown cell, like inspecting and cleaning data, also took a while, as I needed to go through images carefully and make sure each class was balanced. I ran into a few errors when loading the libraries. For example, some model configurations needed specific data shapes, so I had to adjust the input layer to match. Colab's limited memory also caused issues with larger batch sizes, so I lowered them to avoid memory errors. The extra code snippets from the *Copy of Snippets* notebook were helpful for solving compatibility issues with libraries, which ensured I had all the necessary dependencies loaded correctly.

Risk Management and Reflections

I encountered a few major challenges, like data quality issues and the memory limits of Colab. Improving data quality meant I had to carefully clean the dataset and use data augmentation to keep things consistent. To manage Colab's limited GPU resources, I used strategies like smaller batch sizes and early stopping, which helped avoid system overload and crashes. Additionally, each team member contributed reflections on what they learned throughout the project, sharing insights gained from handling these challenges. We faced challenges like long training times and troubleshooting code, but using tools like GitHub helped us solve problems together. The project taught me the value of collaboration and staying organized.

Innovation and Deliverables

To make the project better, I tried exploring other models, like VGG16 and ResNet50, but MobileNet V2 turned out to be the best fit for my needs. We also discussed adding ensemble methods to improve accuracy further, but time was a limitation. The final deliverables included a project report with performance metrics and a clear, documented code base. I made sure the code was easy to follow, with instructions for replication.

Conclusion

This project taught me a lot about building an object detection model with limited resources. Although training, data augmentation, EDA, and tuning took a long time, they were necessary to improve performance. Managing risks, reflecting on challenges, and encouraging innovation helped me create an efficient and accurate model. This experience gave me a strong foundation in computer vision and deep learning, and I'm excited to apply these skills in future projects.