Object Detection for Identifying Traffic Congestion (Using IBM Developer Tools)

Balsharan Bedi, Eisa Adil, Sami Ali Choudhry, Shiv Sondhi
School of Computer Science
University of Windsor
Windsor, Canada
(bedib, adile, choud116, sondhis) @uwindsor.ca
110010312, 110012666, 105187035, 105216319

***Abstract*— Over the years, cities have gotten busier and very often we find ourselves searching for time to complete tasks or fulfill responsibilities. Being stuck in a traffic jam is the last thing we want interrupting our busy schedules. Despite several large highways and better planned cities, traffic congestion still remains a problem in major cities around the world. Not only is it annoying, but can also cause inconvenience to services like ambulances, fire trucks and law-enforcement. Additionally, traffic jams can be hugely detrimental for the air quality of a city and especially the residents who live near major roadways.**

In this project, we use deep learning techniques for object localization and count the number of objects that have been detected. We focus on detecting cars and aim to provide some statistics on traffic buildup and the amount of vehicles that go past a particular junction in a given period of time. Our implementation uses a lightweight architecture and convolution model that enables it to run on real-time video streams as well as on static images. We made use of tools available on the IBM Developer network - like IBM Cloud services and IBM's Watson API.

I. INTRODUCTION

To achieve our aim of real-time object detection and counting cars passing through a given section of the road, we have made use of the Watson Machine Learning API and IBM Cloud storage services [1]. To facilitate the user's interaction with our system, we have created a web application that takes a video as the input and counts the number of cars detected in that video. IBM's Developer tools allowed us to quickly train our deep learning model and store it on the cloud. The Watson ML API also provides architectures that have been pre-trained on larger and more general datasets like ImageNet and Microsoft COCO, which enables us to fine-tune the pre-trained models on car-specific data. The model we have selected is pre-trained on the MS COCO dataset for object detection. It can detect up to 80 different objects in images[3]. These IBM tools helped us manage and save time and allowed us to learn about the available deep learning architectures for our task and find the best fit for our use-case.

The application can be used in several useful ways. In areas where there is high traffic congestion, it can be used for statistical analysis and to generate traffic data. It can also be useful to count cars stranded in a natural disaster for quick relief. With slight modifications, our application can be used in cases such as counting the number of vehicles that drive past a red light or stop sign; identifying speeding cars and can help engineers from other disciplines make decisions on things like the number of lanes a highway should have, traffic signal timings and the number of turning lanes an intersection should have. Measuring traffic volume is an amazingly important task and with slight modifications to our code we can expand detection and counting to other vehicles and pedestrians as well. All in all, the future scope for this application is tremendous.

In the subsequent sections we discuss our project in more detail. Section II describes the data that we have used to train the model, Section III describes our methodology and how we used the IBM tools. After these sections we discuss the model parameters and the training process. We clarify why we have selected the models we did and give some details on the deep learning architecture used. Finally in Section V, we describe the process of building the web interface for our application and sign off with some thoughts on future work.

II. DATA

A. DataSet

The major contribution of the data set comes from Pexels [2] which is a stock image and video database. It uses a large chunk of free-to-download images of many objects and videos of many scenes. We found a number of generalized data (image and video) which we refined by manually filtering out redundant videos and images. We chose to use only video data to train our model since many images had plain backgrounds, had a close-up shot of the car, or showed only interiors of cars.

Once we had the videos, we proceeded by firing the required components in them which in this case is the cars. Later, we trained the model on this annotated video data as per parameters which will help in achieving the most accurate count of the cars. In order to achieve better results and make our model more robust, we used videos having different perspectives of camera, which are detailed as follows:

1) Close-up View: The close-up view will be used in order to obtain the information of incoming cars and detect important features which are needed to train the data. It shows only the front-view of cars and all cars were clearly visible.

Fig. 1. Close-up View

2) Eagle-eye View: This view will be used in order to obtain the information of incoming and outgoing cars, the major goal is to contribute in the counting of fast moving cars and detect cars from their top-views. The cars in this view are also generally smaller due to the height of the camera.

Fig. 2. Eagle-eye View

3) Side View: This view of the camera majorly contributes to help align and adjust as per different orientations. It helps the model detect cars from their side-view. Since it captures traffic moving in both directions, it helped the model learn features on both sides of the cars.

Fig. 3. Side View

III. METHODOLOGY

Our aim is to build a web application in which a custom model is used that detects cars. The first step in our process was to find the data as described in Section II. Once downloaded, IBM Cloud Object Storage was used to store all the training data. Although we used video data for training, the data was stored as frames of images in the cloud. After storing the data, the Watson Machine Learning model was trained on the data to fine-tune it to our needs. For the sake of experimentation, we ran the model without any fine-tuning and verified that it works anyway. However, retraining the model on our video data produced much better results in terms of detection accuracy and stability of the bounding box.

To train the model, we had to download the Watson Command Line Interface (CLI) or Cacli for short. Cacli was used to interface with the server on which our model was being trained. The progress can also be tracked using a GUI on the IBM Developer Dashboard which lets you track the cloud instances you are running. Training on the cloud did not take too long thanks to IBM's powerful servers. Once trained, the model was downloaded and connected to the web application using a NodeJS library provided by IBM. This helped us run the model on real time videos for object detection on the downloaded model without the need to make API calls each time for object detection.

Once we verified that our model could successfully detect and localize cars from videos in real-time, we worked on the counting logic. Although this seems like a trivial task, we were faced with several roadblocks. For instance, our model breaks down each video into its constituent frames and makes predictions on each frame - therefore, simply counting the number of objects detected is not a working solution because it will repeatedly count the same object. We explored a few solutions which are discussed later in the report.

A. Workflow

Figure 4 shows the workflow of our process discussed above with the legend given below. It flows from the right to the left.

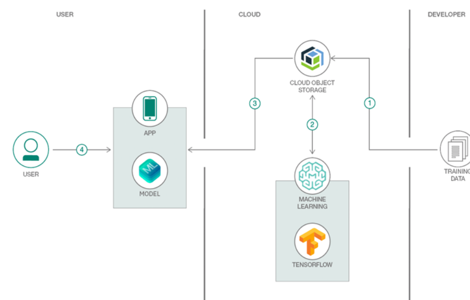

Fig. 4. Project Workflow

1) Store the training images (video frames) on IBM Cloud Object Storage.
2) The Watson Machine Learning model accesses the training data stored in the cloud for training and the trained model is stored back on Cloud Object Storage.
3) The trained model is glued to the web application. A library created by IBM helps us in making inferences on the trained model within our web application.

4) Live stream of traffic or recorded video is streamed to the web application in real time to perform object detection.
5) Car counting functionality is implemented in the web application using JavaScript.

B. Preparing the Training Data

For the Watson Machine Learning model to be trained effectively, the data of images or videos must be prepared using annotations. In order to accomplish this task, we used the Cloud annotations that simplified the task of creating annotations. In this process, we simply uploaded a training video to the annotations interface. The video was split into many frames and each image had to be associated with annotations. For the first 50-80 frames, we manually drew bounding boxes around cars using tools provided in the interface. An example of this can be seen in Figure 5.

After annotating around 50-80 images, the annotations tool learns to find the objects we are detecting and automatically created the bounding boxes for the rest of the frames. After going through the automatically annotated images and making changes wherever necessary, our annotated data was ready.

One thing to note is that the training images must resemble the testing images - they should be similar in some ways. This is so that the model learns valuable features and can perform well during the test phase. Of course, you can never be sure of how the test data is going to look; so care must be taken to include as much variation as possible to the training data. After generating annotations, the Watson Machine Learning API performs some pre-processing steps on the data. This pre-processing usually depends on the hyperparameters of the model being used. In our case, the training images are resized to 300x300 pixels as this is the requirement of our MobileNet ConvNet.

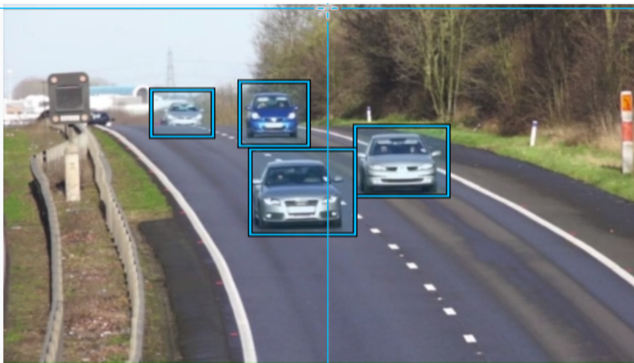

Fig. 5. Annotating images / video frames

C. Creating and Training the ML Instance

Our Watson Machine Learning model was built using the TensorFlow framework. An instance of the ML model was created in the cloud using the IBM Cloud Dashboard. Now, a model is ready and available in the cloud, and needs to be trained on our training data. As mentioned, in order to train the model, we used the Cloud Annotation CLI (cacli). The major advantage of using the SSD architecture with MobileNet is that the run-time is very low and accuracy is only partially worse than the state-of-the-art - this makes it ideal for use in web and mobile applications. After the model is trained, it can either be downloaded or used directly in the web application.

More information about the SSD architecure and MobileNet is provided in Section IV but to summarise the reasons for our choice; they solve two major problems with traditional deep learning architectures: slow speed and high amount of computation required.

D. Creating a Web-Interface for the Model

Once the model was trained, we first downloaded it to our local system to save a copy of the model and maybe use it in a server-side application later. For our current project, we created a web-page to demonstrate the working of our model. On the web-page, a video can be played and the cars will be detected in real-time. To use the fine-tuned model in the web-application, IBM Cloud Annotations provides a NodeJS library to connect the downloaded model instances with the React web-interface. This library helps us infer the model and draw boxes on the car for object detection. The car counting functionality is then implemented depending on the geometric position of these boxes.

IV. TRAINING PROCESS

The Single-Shot Detector (SSD) [4] is a object detection architecture that is able to detect objects in an image in a single-shot. This is as opposed to region-proposal networks like the R-CNN family that use a two-shot approach. In the SSD, a Convolutional Neural Network (CNN) is first used to generate a feature map of the input image. A 3x3 sliding window approach is used at different levels of the feature map generation process to detect objects in the image with a certain confidence. Applying the sliding window at different levels ensures that the object can be found at different scales and sizes.

The CNN architecture used in our model is MobileNet. This choice of CNN architecture is lightweight and allows the model to be run on mobile devices and in web-applications. MobileNet uses depthwise convolutions followed by pointwise convolutions which are together called a depthwise separable convolution block. The depthwise convolutions apply convolution kernels across each channel of the image separately. The output of each depthwise convolution in MobileNet is always equal to the number of input channels, i.e. the convolution operation produces just one activation map per channel. The pointwise convolution is a normal convolution operation using a 1x1 kernel. This can be used to increase or decrease the number of channels by outputting a weighted sum across channels. Together, they produce the same effect as a normal convolution but with lesser computational cost.

There are two versions of MobileNet available, the object detector that we selected uses V1.0 which works in the way

described above. The newer version makes some modifications to the above process with only little improvements in accuracy.

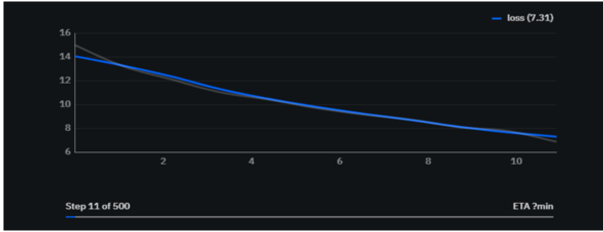

Fig. 6. Plotting loss throughout training - 1

A. Loss Function

The model uses a multi-task loss function. This means that the loss function aims to optimize two objectives - the confidence of the object (car) being present in the 3x3 window, and a regression loss (L2-norm) for the bounding-box co-ordinates. Therefore the final loss value reflects the performance of the model in:

- Detecting a car in the image, and
- Generating a bounding box that correctly bounds the car.

The multi-task loss function will give one common metric that measures both aspects.

Figures 9, 7 and 8 show the decrease in loss over 500 steps while training our model for the first epoch.

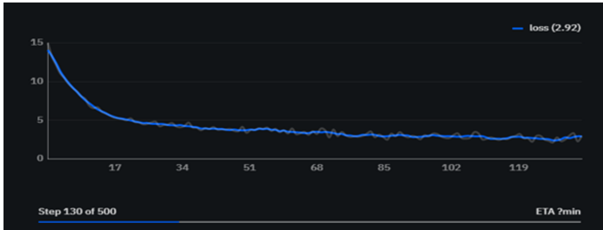

Fig. 7. Plotting loss throughout training - 2

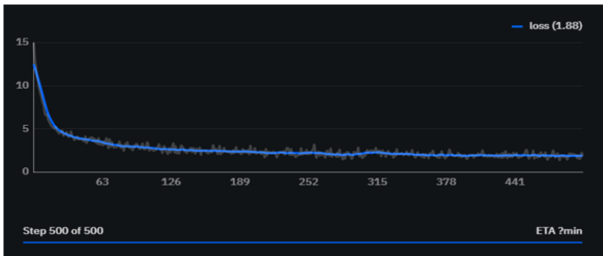

Fig. 8. Plotting loss throughout training - 3

B. Transfer Learning

As mentioned, the model described at the top of this section was already trained on the Microsoft COCO dataset for the object detection task. This enabled the model to detect upto 80 objects in images and videos. For our purpose we decided to further fine-tune the pre-trained model to teach it features specific to our use-case of detecting cars. Since the model was pre-trained on an extensive database of annotated images, the model accuracy was already acceptable to begin with. However, the accuracy improved satisfactorily after further training on our own data. We trained the model twice (2 epochs) with 500 steps per epoch.

V. CREATING THE USER INTERFACE

The user interface for this web application was developed using React, which is a JavaScript library that is used to build Single Page Applications (SPA) using the MVC paradigm. Its advantage is that the components in the web application can be updated dynamically with relative ease [5]. React renders on a NodeJS server, and therefore works universally.

After the model is trained, a model file consisting of the weights and biases of the CNN is extracted. This model is saved and stored in the directory of the web application [6].

The IBM Cloud Annotations framework provides a NodeJS library that can be used to connect the downloaded model to the web application. This helps us in making inferences on the trained model within the application. It takes a frame-by-frame input of images in which there are one or more cars, and then returns the dimensions and position of the region in which a car is found. We then draw boxes around this region for visualization in our web interface.

We then implement the functionality to count cars within our web application. It takes either a live stream or recorded video as input, passes it to the IBM library to return box co-ordinates, and then we visualize the car boxes and display the number of cars as output. This helps us in performing real time object detection and car counting on a road.

Cars are counted by selecting a region in the input stream, in which if a car passes through, it gets counted to the total. This works in a similar fashion to a toll gate, meaning that when a car passes through a specific range of co-ordinates of the stream canvas, we count it as a car that has passed through. This helps in counting a car only once, rather than at each frame multiple times.

After the car is counted, we increment a counter variable that is defined in the JavaScript Document Object Model. This variable is displayed immediately upon change using React's dynamic components. This gives the illusion that the cars are being counted as they pass through our selected region in the video stream. This count is then displayed on the web page as its being updated after each passing car.

VI. FUTURE WORK

The final count method of our application has an error rate of less than 10% - out of 50 cars on the road it misses anywhere between 3 to 5 cars. We have ideas for improving this even further and plan to follow up on that to increase the counting accuracy.

In the task of detection, the model performs well for several videos that were presented to it. Most cars are detected with high confidence too.

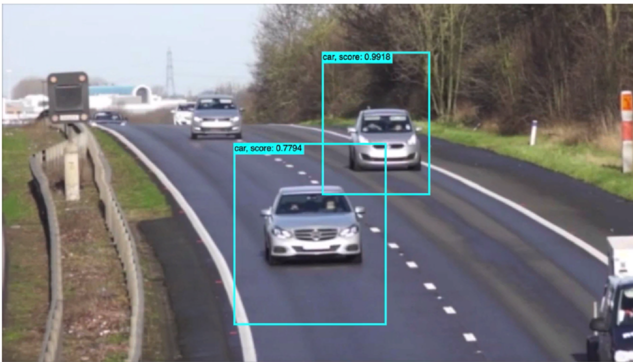

Fig. 9. React Web Application for counting the cars that pass through a region in the video stream

For localization, our model again does a good job. Although some cars are detected and localized much later into the video, every car is localized with a bounding box at least once while it is visible in the frame. Our aim for the future is to improve on these and some other minor issues with the program, such as lag.

Further, we can try experimenting with different combinations of object localization methods, ConvNets and model hyperparameters to see what works best for us.

Apart from improving the performance of our implementation, we could also extend functionality to count cars that run a red light or stop sign; count the number of say, red cars or trucks on a road etc. Traffic data, especially the volume of traffic in an area is useful for civil engineers because it helps them make better decisions while building roads or during redevelopment projects. We are open to new ideas and extending our project into other domains.

For the user interface, we plan on making an interactive and user-friendly dashboard that provides metrics such as number of cars passed through in real time. We also intend on making this dashboard cross-platform, so that this application can be used by, for example, a traffic police through a mobile phone.

REFERENCES

[1] “Introduction,” Cloud Annotations. [Online]. Available: https://cloud.annotations.ai/workshops/object-detection/index.html. [Accessed: 29-Feb-2020].

[2] "Vehicles On Highway At Fast Speed" [Video], https://www.pexels.com/video/vehicles-on-highway-at-fast-speed-2431853/

[3] “Object Detector,” IBM Developer. [Online]. Available: https://developer.ibm.com/exchanges/models/all/max-object-detector. [Accessed: 29-Feb-2020].

[4] D. Impiombato et al., “SSD: Single Shot MultiBox Detector Wei,” Nucl. Instruments Methods Phys. Res. Sect. A Accel. Spectrometers, Detect. Assoc. Equip., vol. 794, pp. 185–192, 2015.

[5] P. Krill, P. Krill, and InfoWorld, “React: Making faster, smoother UIs for data-driven Web apps,” InfoWorld, 15-May-2014. [Online]. Available: https://www.infoworld.com/article/2608181/react–making-faster–smoother-uis-for-data-driven-web-apps.html. [Accessed: 29-Feb-2020]

[6] “Model deployment.” [Online]. Available: https://dataplatform.cloud.ibm.com/docs/content/DO/WML/_Deployment/ModelDeployment.html. [Accessed: 29-Feb-2020]