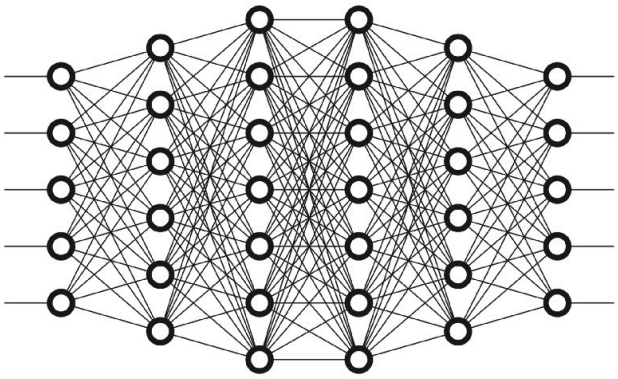
A Quantitative Study of the Different TensorFlow-Lite Models for Image Detection Using a USD Currency Dataset

Sadhil Mehta

Grade 11

Tippecanoe High School - Tipp City

Problem



In the ever-evolving landscape of computer science, the prominence of deep learning has reached unprecedented levels. Deep learning, a subset of artificial intelligence (AI), creates a methodology that mirrors the intricate processing of data by the human brain. While it projects an image of being complex and involuted, deep learning has been integrated into various facets of our technology, ranging from web browsers to traffic cameras and beyond. At its core, a deep learning model shares a fundamental objective with the human brain – the recognition of intricate patterns within data including images, text, and sounds, allowing for it to produce precise insights and predictions. Notably, one of the applications of deep learning lies in image detection and classification. Like how our brains assimilate complex visual inputs, a neural network is trained with an extensive array of images. Through this process, the model discerns patterns within the dataset, enabling it to accurately categorize images. I was introduced to the world of AI image detection last year when I attended the State Science Day celebration where Dr. Tanya Berger-Wolf, the director of the Translational Data Analytics Institute at the Ohio State University, was speaking as a guest speaker. She and her team did a “census” on whales and sharks where they used image object detection to identify differences in the fins to individualize whales and sharks and do tracking on them. This type of research truly captivated me. I wanted to study image detection

extensively on its wide-scale applications. From what I thought, I believed that this system required the use of high-performance machines. However, when I did more research, I found that it is possible to use these image detection models on less powerful machines such as edge devices, which include devices like the Raspberry PI, Android, and IOS devices. I found a software called TensorFlow Lite which helps run complex image detection models on these edge devices with compatible performance. I decided I wanted to research this frontier more.

Research

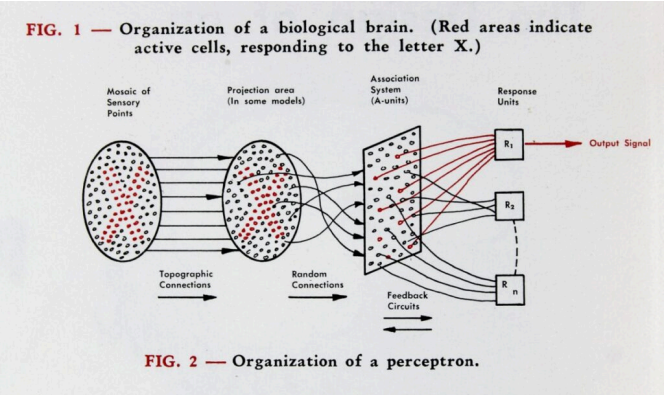
When I started my research on how to run these image detection models on low-power devices, I found I had to use TensorFlow Lite, a way to run image detection neural network models on edge devices. However, the use of this required a deeper understanding of the basics of deep learning and more.

Basics of Deep Learning

Deep learning is a subset of both Artificial intelligence and Machine learning. Artificial Intelligence refers to the technique of a machine to mimic human behavior. Machine learning takes that a step further by mimicking this behavior through algorithms that are trained through data. Finally, deep learning is based on how the human brain works which is through an artificial neural network. The major difference between machine learning and deep learning is that machine learning requires feature engineering which leverages data to create new

variables that aren't in the training set, whereas deep learning has this built into its neural network, thus requiring lots more data.[1]

Neural Networks



These neural networks, as the name might suggest, stem from the model of the neural networks of our brains. The brain is made up of cells called neurons, which send signals to each other through connections known as synapses. Neurons transmit electrical signals to other neurons based on the signals they receive from the other neurons. Artificial Neurons work similarly. An artificial neuron simulates how a biological neuron behaves by collecting together the values of the inputs it receives. If this is above some threshold, it sends its signal to its output, which is then received by other neurons. However, the signals aren't on an equal weight-by-weight basis. Each of its inputs can be adjusted by multiplying it by some weighting factor. Say, if input A is twice as important as input B, then input A would weigh 2. Weights can also be negative if the value of that input is unimportant. Just as it occurs in synaptic connections, each of these artificial neurons is connected to other neurons in the network, whose values are weighted, and the signals propagating through the network are strengthened or dampened by these weight values. When we train a neural network, we adjust for different weight values so that the output matches our initial, correct output. The pioneer of neural network research was Frank Rosenblatt. According to Rosenblatt, the simplest neural networks

involve a minimum of three layers. The first layer is usually the **input layer**. This receives input or some medium, for example, sounds, pixels of an image, etc. These inputs are outputted to a second layer, called the **"hidden layer"**. This hidden layer involves adjusting those weight values to produce a correct output to the final layer, the **output layer**. This final layer is what gives you the answer to what the network has been trained to do. For example, a network can be trained to recognize photos of cats. The output layer of the network would then have two outputs, "cat" and "not cat." Given a dataset of photos that a human has labeled with either "cat" or "not a cat," the network is trained by adjusting its weights so that when it sees a new unlabeled cat photo, it outputs a greater than 90 percent probability that it is a cat, or less than 10 percent if it is not. This final layer can be adjusted by the number of outputs that are needed. For a time, this technology was limited to only three layers and only one "hidden layer." This was changed by the 1980s when people like Geoffrey Hinton were able to develop an algorithm known as **"backpropagation"** to train networks with multiple hidden layers. Networks with many hidden layers are also known as **"multilayer perceptrons"** or "deep" neural networks, hence the term "deep" learning.[1]

Mathematics of a Neural Network

Perceptrons

The main methodology of mathematics that revolves around neural networks deals with perceptrons. Perceptrons — invented by Frank Rosenblatt in 1958, are the simplest neural network that consists of n number of inputs, only one neuron, and one output, where n is the number of features of our dataset. The process of passing the data through the neural network

is known as forward propagation and the forward propagation is carried out in a perceptron. There are three main steps for forward propagation. First, for each input, the input values (x_i) and the weights (w_i) are multiplied and the sum of every multiplied value is taken. Weights — represent the strength of the connection between neurons and decide how much influence the given input will have on the neuron's output. If the weight w_1 has a higher value than the weight w_2, then the input x_1 will have a higher influence on the output than x_2.[2]

$$\sum = (x_1 \times w_1) + (x_2 \times w_2) + \cdots + (x_n \times w_n)$$

Another method of writing out this summation is taking the dot product. The row vectors of the inputs and weights are $x = [x_1, x_2, \ldots, x_n]$ and $w = [w_1, w_2, \ldots, w_n]$ respectively, and their dot product is the summation of all of the weights and inputs multiplied.[2]

$$x.w = (x_1 \times w_1) + (x_2 \times w_2) + \cdots + (x_n \times w_n)$$

$$\sum = x.w$$

Along with this dot product, a certain bias factor is also added to the dot product. This bias, also known as the offset, is necessary in most cases to move the entire activation function to the left or right to generate the required output values.[2]

$$z = x.w + b$$

This function, noted as z, is then passed through a non-linear activation function. Activation functions are used to introduce non-linearity into the output of the neurons, without which the

neural network will just be a linear function. This means there is not a straight-line or direct relationship between an independent variable and a dependent variable. Depending on the activation function, the speed of the network could be different. Perceptrons have a binary step function as their activation function, while another common one is a logistic function also known as a sigmoid. The sigmoid is symbolized by a σ.[2]

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Learning Algorithm

The neural network learns by adjusting its weights and bias (threshold) iteratively to yield the desired output. These are also called free parameters. For learning to take place, the Neural Network is trained first. The training is performed using a defined set of rules, also known as the learning algorithm. The learning algorithm consists of two parts — **backpropagation and optimization**.[2]

Backpropagation, short for backward propagation of errors, refers to the algorithm for computing the gradient of the loss function concerning the weights. However, the term is often used to refer to the entire learning algorithm. The loss function mentioned above is used to get an estimation of how far we are from our desired solution. Generally, mean squared error is chosen as the loss function for regression problems and cross-entropy for classification problems. The mean square error simply takes the square of the difference between actual (y_i) and predicted value ($\hat{y}_i$).

$$MSE_i = (y_i - \hat{y}_i)^2$$

The loss function is calculated for the entire training dataset and their average is called the Cost function C.[2]

$$C = MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

As part of this learning algorithm, it is necessary to find the best weights and biases for our Perceptron. This is done with the help of the gradients (rate of change) which is how one quantity changes about another quantity. In our case, we need to find the gradient of the cost function for the weights and bias. This can be done via partial derivation. We will use the chain rule to differentiate these [2]

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial w_i}$$

$$\frac{\partial C}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = 2 \times \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)$$

$$\begin{aligned}
\frac{\partial z}{\partial w_i} &= \frac{\partial}{\partial w_i}(z) \\
&= \frac{\partial}{\partial w_i}\sum_{i=1}^{n}(x_i.w_i + b) \\
&= x_i
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \hat{y}}{\partial z} &= \frac{\partial}{\partial z}\sigma(z) \\
&= \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{(1+e^{-z})} \times \frac{e^{-z}}{(1+e^{-z})} \\
&= \frac{1}{(1+e^{-z})} \times \left(1 - \frac{1}{(1+e^{-z})}\right) \\
&= \sigma(z) \times (1 - \sigma(z))
\end{aligned}$$

Once all the differentiation is done, we are left with this equation for the gradient of the cost function.

$$\frac{\partial C}{\partial w_i} = \frac{2}{n} \times sum(y - \hat{y}) \times \sigma(z) \times (1 - \sigma(z)) \times x_i$$

Along with backpropagation, we also have optimization in the learning algorithm. Optimization is the selection of the best element from some set of available alternatives, which is the selection of the best weights and biases of the perceptron. In this case, the gradient descent could be used as the optimization algorithm, which changes the weights and bias, proportional to the negative gradient of the cost function for the corresponding weight or bias. Learning rate (α) is a hyperparameter that is used to control how much the weights and biases are changed. The weights and bias are updated as follows and the backpropagation and gradient descent is repeated until convergence.[2]

$$w_i = w_i - \left(\alpha \times \frac{\partial C}{\partial w_i}\right)$$

$$b = b - \left(\alpha \times \frac{\partial C}{\partial b}\right)$$

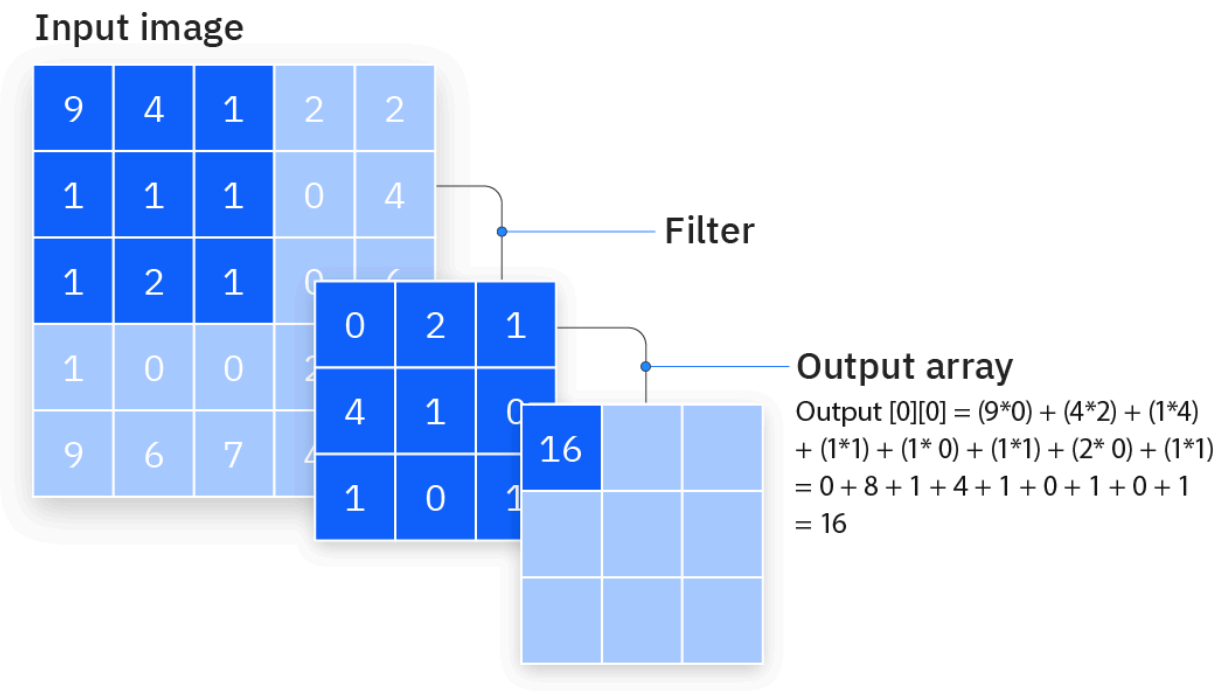
All of these calculations represent the workings of only one neuron in the network. These basic concepts apply to all kinds of neural networks with some modifications.[2]

Once I had researched how traditional neural networks work, I decided to shift my focus to how these can be used to detect images. I noticed that the common network used for image detection is the Convolutional Neural Network (CNN). I decided to research this topic further.

Convolutional Neural Network (CNN)

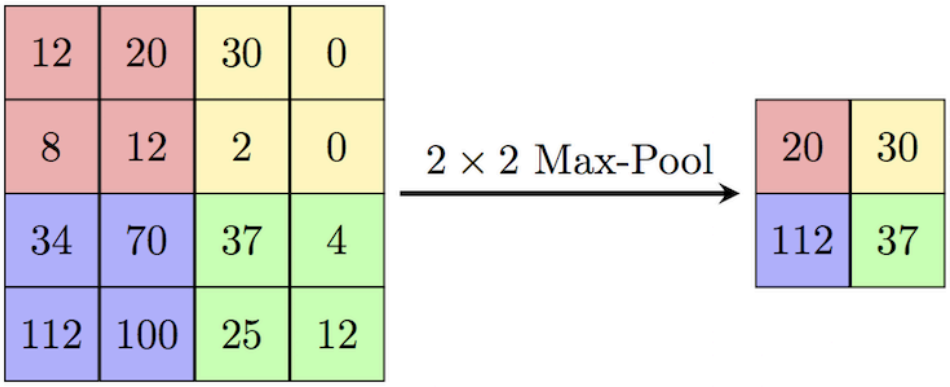
A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm that can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image, and be able to differentiate one from the other. These networks have three layers, a **convolutional layer, a pooling layer, and a fully connected (FC) layer**. The convolutional layer is the core building block of a CNN, and it is where the majority of computation occurs. It requires a few components, which are input data, a filter, and a feature map. When we take a 3D image, we are dealing with three dimensions. All of these will correspond to an RGB in an image. We also have a feature detector, also known as a kernel or a filter, which will move across the receptive fields of the image, checking if the feature is present. This process is known as a convolution. This feature detector is a 2D array of weights, which represents part of the image. They are usually a 3x3 matrix. The filter is then applied to an area of the image, and a dot product is calculated between the input pixels and the filter. This dot product is then fed into an output array. Afterward, the filter shifts by a stride, repeating the process until the kernel has swept across the entire image. The final output from the series of dot products from the input and the filter is known as **a feature map**. Some parameters, like the weight values, adjust during training through the process of backpropagation and gradient descent. However, three hyperparameters affect the volume size of the output that needs to be set before the training of the neural network begins. The first is the number of filters, which affects the depth i of the output and the number of filters determines the number of feature maps. Another is the stride. Stride is the distance, or number of pixels, that the kernel moves over the input matrix.

While stride values of two or greater are rare, a larger stride yields a smaller output. The other is padding. Zero-padding is usually used when the filters do not fit the input image. This sets all elements that fall outside of the input matrix to zero, producing a larger or equally sized output. After each convolution operation, a CNN applies a Rectified Linear Unit (ReLU) transformation to the feature map, introducing nonlinearity to the model.[3]



After this layer, there is the **pooling layer**. Pooling layers, also known as downsampling, conduct dimensionality reduction, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter across the entire input, but the difference is that this filter does not have any weights. Instead, the kernel applies an aggregation function to the values within the receptive field, populating the output array. There are two main types of pooling. The first type is max pooling where the filter moves across the input and it selects the pixel with the maximum value to send to the output array. As an aside,

this approach tends to be used more often compared to average pooling. Average pooling is when the filter calculates the average value within the receptive field to send to the output array as it moves across the input.[3]



The last layer, the fully connected layer, performs the task of classification based on the features extracted through the previous layers and their different filters. In the fully connected layer, each node in the output layer connects directly to a node in the previous layer. While convolutional and pooling layers tend to use ReLu functions, FC layers usually leverage a softmax activation function to classify inputs appropriately, producing a probability from 0 to 1.[3]

Once I had researched overall about how Convolutional neural networks work, I noticed that many of the articles about these networks and image detection, in general, mentioned the use of TensorFlow. I further decided to research TensorFlow and its use in image detection.

TensorFlow and TensorFlow Lite

TensorFlow is an open-source framework tool used by Machine Learning and Deep Learning Engineers that helps to train and execute many neural networks. It helps reduce complex computations on large datasets and allows for better accuracy. TensorFlow has thus become a very high-demand tool for companies. When one uses TensorFlow to implement and train a machine learning algorithm, one typically ends up with a model file that takes up a lot of storage space and needs a GPU to run inference. TensorFlow Lite is a solution for running deep learning models on weaker computers. This uses a custom memory allocator for execution latency and minimum load. It also explains the new file format supported by Flat Buffers. TensorFlow Lite takes existing models and converts them into an optimized version within the sort of .tflite file. One of its major uses is for image detection on mobile devices.[4]

When I researched TensorFlow, I found using TensorFlow Lite for my project would be ideal as I could use my Raspberry PI to run image detection programs. After that, I went to the TensorFlow Lite Model Zoo and explored it. I saw that there were numerous models to choose from, all catering to different needs. To choose which one to study, I was going to rely on benchmark testing, however, it was nowhere to be found. I pondered if I could do this testing on my own and figure out which of these models is truly the best image detection model on the Raspberry PI. These are the ones I selected:

1. ***SSD-MobileNet-v2***
2. ***SSD-MobileNet-v2-FPNLite-320x320***
3. ***EfficientDet-d0***

The next question was which data to use. I wanted to use something readily available to me, something of practicability that would be mostly uniform, yet still be a challenge for the

models. I decided to identify US Dollar Paper Currency. This would be something that has real usability in the real world for visually impaired people for reading money amounts and helping them identify their money.

SSD-MobileNet-v2

The SSD-MobileNet-v2 model in TensorFlow represents a powerful combination of two key components: the MobileNet-v2 architecture and the Single Shot Multibox Detector (SSD) framework. MobileNet-v2 introduces efficiency to the model through depthwise separable convolutions and inverted residuals, ensuring a lightweight design ideal for deployment on devices with limited computational resources. The incorporation of the SSD framework enhances its object detection capabilities by predicting class scores and refining bounding box coordinates at multiple scales within a single pass, facilitated by anchor boxes.[5]

By using SSD, we only need to take one single shot to detect multiple objects within the image, while regional proposal network (RPN) based approaches such as the R-CNN series need two shots, one for generating region proposals, and one for detecting the object of each proposal. Thus, SSD is much faster compared with two-shot RPN-based approaches. Another defining feature of this model is the integration of a Feature Pyramid Network (FPN), fostering the extraction of multi-scale features crucial for accurately identifying objects of diverse sizes within an image. The lateral connections within the FPN enable the model to seamlessly combine high-level semantic information with fine-grained details, contributing to its robust performance in real-time scenarios.[6]

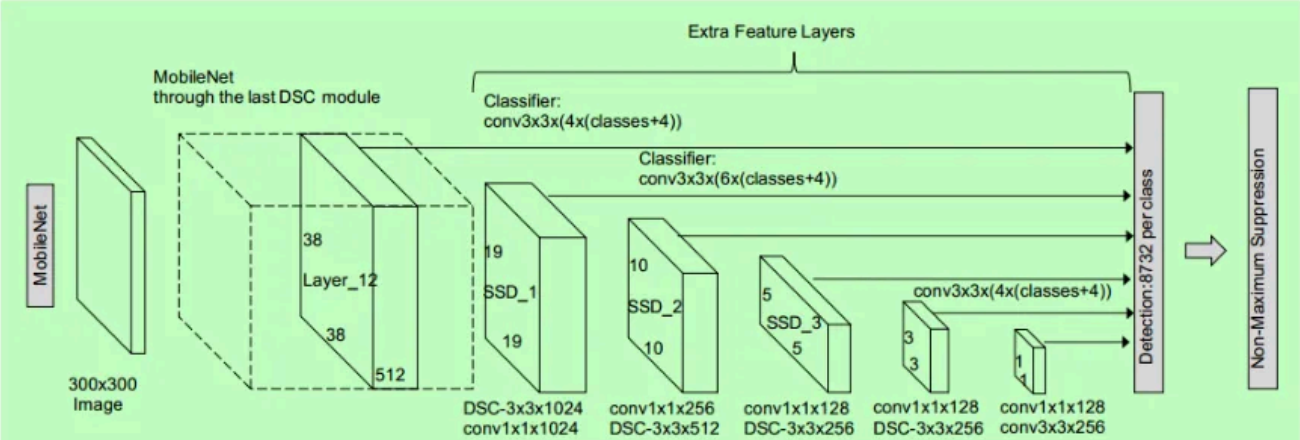


Figure 6. Mobile-Det: SSD-based detection with MobileNet as backbone, modified based on [14]

Notably, SSD-MobileNet-v2's efficiency is further underscored by its post-processing steps, including Non-Maximum Suppression (NMS), which ensures that only the most confident predictions are retained, thereby enhancing the accuracy and precision of object detection. This combination of lightweight architecture, multi-scale feature learning, and efficient post-processing makes SSD-MobileNet-v2 a versatile solution for real-time object detection tasks, finding applications in various domains where computational resources are constrained, such as mobile and edge computing environments.[7]

SSD-MobileNet-v2-FPNLite-320x320

The SSD-MobileNet-v2-FPNLite-320x320 model in TensorFlow is a sophisticated architecture designed for efficient and accurate real-time object detection. At its core is the MobileNet-v2 backbone, which is known for its lightweight design, making it well-suited for deployment on devices with limited computational resources.[5] The inclusion of Feature Pyramid Networks (FPNLite) enhances the model's capability to capture multi-scale features within an image, a crucial aspect for accurately detecting objects of varying sizes. FPNLite streamlines the Feature

Pyramid Network architecture for optimal efficiency, contributing to the model's suitability for real-time applications.[6]

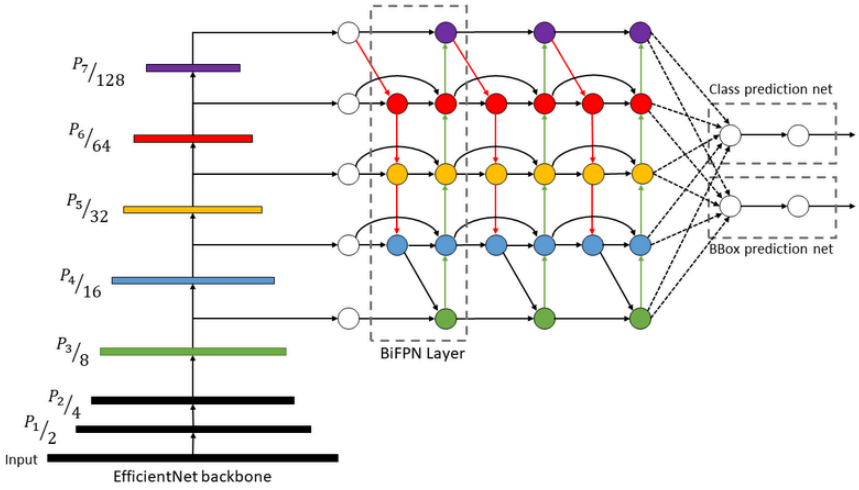
Operating with an input size of 320x320, the SSD-MobileNet-v2-FPNLite-320x320 strikes a balance between accuracy and computational efficiency, making it ideal for scenarios where real-time object detection is paramount. The model predicts class scores and refines bounding box coordinates at multiple scales, facilitated by strategically placed anchor boxes that accommodate objects of diverse shapes and sizes. Post-processing involves the application of Non-Maximum Suppression (NMS), ensuring that only the most confident and non-redundant predictions are retained.[8]

The efficient design of the SSD-MobileNet-v2-FPNLite-320x320 makes it a valuable asset for deployment on resource-constrained devices, such as mobile phones and edge computing platforms. Its ability to seamlessly integrate multi-scale features, coupled with optimized architectural choices, positions it as a robust solution for real-time object detection tasks in diverse applications, ranging from mobile-based image recognition to edge devices.[8]

EfficientDet-d0

The EfficientDet-d0 model in TensorFlow is used to pursue efficiency without compromising accuracy in the use of object detection. At its core is the EfficientNet-B0 backbone, which adopts a novel approach to model scaling by optimizing depth, width, and resolution. This compound scaling allows EfficientDet-d0 to achieve a balance between computational efficiency and high-performance feature extraction. The model then employs a Bidirectional Feature Pyramid Network (BiFPN) to efficiently fuse multi-scale features bidirectionally. This ensures that both

low-level and high-level features are seamlessly integrated, enhancing the model's ability to detect objects across various scales within an image.[9]



EfficientDet-d0 excels in generating predictions through anchor boxes and predefined bounding boxes at different scales and aspect ratios. By predicting bounding box coordinates and class scores for each anchor box at multiple feature levels, the model becomes adept at detecting objects of different sizes. The integration of Non-Maximum Suppression (NMS) in the post-processing stage refines the predictions, ensuring only the most reliable and non-redundant detections are retained.[9]

One of the distinguishing features of EfficientDet-d0 lies in its "d0" scaling variant, emphasizing a balanced configuration that optimally utilizes computational resources while maintaining commendable accuracy. This makes EfficientDet-d0 particularly well-suited for deployment on resource-constrained devices, such as mobile and edge computing platforms. Overall, the model showcases a blend of efficient architecture, multi-scale feature learning, and robust

post-processing, positioning it as a solution for real-time object detection applications where efficiency is paramount.[9]

Procedure

Materials

Hardware:

1. **Raspberry PI 4**
2. **Webcam**
3. **USD bills - $100, $10, $1, $50, $5**

Software:

1. **Anaconda Powershell**
2. **Python Version 3.9**
3. **Tensorflow Version 2.8**
4. **Google's performance benchmark tool**
5. **Cartucho mAP Library**

Computer Procedure

1. **Gather and Label Training Images**

Before we start training, we need to gather and label images that will be used for training the object detection model. The training images should have a variety of backgrounds and lighting conditions. For our testing, I decided to identify **US Dollar Paper Currency**. This would be something that has real usability in the real world as a uniform study. I made sure to include all denominations including $1, $5, $10, $20, $50, and $100. I focused on taking pictures at different angles, distances, perspectives, and the front and back of all the bills. I also utilized images from the Roboflow database to increase the size of the dataset for better detection. The final image dataset consisted of around 3000 images.

2. **Install Anaconda**

Anaconda simplifies package management and deployment by providing a comprehensive set of pre-installed libraries and tools. Anaconda will help us in running Python and other sets of libraries which are prerequisites for running TensorFlow. After installing Anaconda, create an environment that runs and supports Python version 3.9 using these commands:

```
conda create --name myenv python=3.9

conda activate myenv
```

Clone the TensorFlow library from GitHub using this command. The GitHub folder has some useful Python routines that will be used later.

```
git clone --depth 1 https://github.com/TensorFlow/models
```

3. **Install TensorFlow**

I decided to target TensorFlow 2.8 as most of the tutorials I researched were using this version.

```
pip install TensorFlow==2.8.0

pip install TensorFlow_io==0.23.1
```

4. **Create the CSV files to generate label files**

   The labelImg application creates the label information in Pascal VOC format. We will need to convert that data into CSV using a Python subroutine.

5. **Convert the CSV file into tfrecord file**

   The TensorFlow training requires the creation of a tfrecord file, TFRecord is a custom TensorFlow format for storing a sequence of binary records. The CSV data is converted into tfrecord by the following subroutine.

```
tf.python_io.TFRecordWriter
```

6. **Start training with the specific type of model from the TensorFlow model zoo.**

```
python
/content/models/research/object_detection/export_tflite_graph_tf2.py
```

   This step takes anywhere from 3 hrs - 5 hrs.

7. **After the training is complete the model needs to be converted to TFlite**

   The following Python routine will need to be executed.

```
import TensorFlow as tf
```

```
converter =
tf.lite.TFLiteConverter.from_saved_model('/content/custom_mo
del_lite/saved_model')
tflite_model = converter.convert()
```

8. **Transfer the file model to Raspberry PI**

   There are different steps required to create the basic environment to run the TFlite model. These steps include the installation of the required versions of Python and the TensorFlow libraries.

9. **Test on Raspberry PI**

   Test the model with a TensorFlow Python subroutine that uses the webcam to identify the test video stream.

10. **Calculate Inference**

    Use Google's performance benchmark tool for TensorFlow Lite models

```
wget -O benchmark_model
https://storage.googleapis.com/TensorFlow-nightly-public/pro
d/TensorFlow/release/lite/tools/nightly/latest/linux_arm_ben
chmark_model_plus_flex
```

11. **Calculate MAP scores**

    Use the open-source mAP calculator available from the Cartucho Map library.

    https://github.com/Cartucho/mAP

The mAP score was calculated by running the model of a set of unseen test images by the trained model. The test was run on 50+ images.

12. **Observe the FPS for the running script.**

13. **Repeat steps 6 through 13 to test different models.**

Measures

1. **Inference Speed**- The inference speed of a model is the amount of time it will take to process the inputs it receives and generate an output. When the model runs, the inference of the model involves propagating the input through each layer of the network, performing layer operations, and calculating values of nodes and neurons until the final output is produced. The time it takes to do all of those processes is the inference time. The more layers and nodes a model has, the longer inferencing takes.
2. **Frames Per Second (FPS)**- The FPS is the rate at which frames will appear on the screen. While it may seem arbitrary to know this, it still helps us to test for speed. This isn't a true measurement of the model's inference speed, as FPS is restricted by the amount of time it takes to open OpenCV to have frames from the camera and how fast it can produce the labeled images on the screen, however, it does give us a good idea on the speed of the model on your application.
3. **Mean Average Precision (mAP)**- To measure the accuracy of our model, we use mAP. This is used to evaluate many other popular object detection neural networks. The formula for this is based on a few sub-metrics. The first is the confusion matrix. In the confusion matrix, there are four attributes. There are true positives when the model

predicts a label and matches correctly as per ground truth, true negatives when the model does not predict the label and is not a part of the ground truth, and false positives when the model predicted a label, but it is not a part of the ground truth, and false negatives when the model does not predict a label, but it is part of the ground truth. The mAP also takes into account the Intersection over the Union, which indicates the overlap of the predicted bounding box coordinates to the ground truth box. Higher IoU indicates the predicted bounding box coordinates closely resemble the ground truth box coordinates. The final thing which is taken into account is the precision. Precision measures how well you can find true positives (TP) out of all positive predictions. It is a measure of “when your model predicts, how often does it predict correctly?” For instance, the precision is calculated using the IoU threshold in object detection tasks. Finally, there is recall which is how well you can find the true positives out of all of the predictions. Recall is a measure of “Has your model predicted every time that it should have predicted?” After you calculate the curve for the precision and recall metrics, you calculate the integral of the curve which correlates to the average precision.

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} \text{AP}_i$$

Hypothesis

From my research on every model, running these TensorFlow Lite networks on an application with limited computational power such as my Raspberry PI, I should see varying values of inference speed, FPS, and a mAP score. When analyzing the SSD-MobileNet-v2-FPNLite-320x320 model, it is most likely to have the fastest inference speed. The backbone of this model, the MobileNet-v2, is specifically designed to be used on edge devices. The strategic use of depth-wise separable convolutions should allow for a more streamlined design. The use of FPNLite also should emphasize computational efficiency. These architectural choices, combined with a modest input size of 320x320, should lead to faster inference speeds, making it well-suited for real-time applications on resource-constrained devices. SSD-MobileNet-v2 shares similarities with its "Lite" variant but may exhibit slightly slower inference speeds due to potential differences in model complexity and size. While both models leverage MobileNet-v2 for lightweight feature extraction, the "Lite" version places a stronger emphasis on streamlined efficiency, contributing to faster predictions. EfficientDet-d0, designed to offer an optimal trade-off between accuracy and efficiency, should also provide competitive inference speeds. However, its overall architecture complexity, even though optimized for efficiency, may introduce additional computational overhead compared to the other models. I believe that these results will be mirrored when dealing with FPS. When dealing with a mAP score, EfficientDet-d0 should stand out for its emphasis on accuracy, employing compound scaling and a bidirectional Feature Pyramid Network (BiFPN) for efficient feature fusion. This should

make it a strong contender for scenarios where precision is key. Next would be the SSD-MobileNet-v2. This is the perfect middle ground between efficiency and precision with accuracy through its MobileNet-v2 backbone and multi-scale predictions.

SD-MobileNet-v2-FPNLite-320x320, designed for efficiency, strikes a balance between accuracy and real-time processing on resource-constrained devices. While its streamlined architecture may have limitations in handling very complex scenes, it should remain competitive in terms of accuracy, however, it should still have lower accuracy compared to the other two.

RESULTS

MAP Scores

SSD MobileNet V2											
IoU	***0.5***	***0.55***	***0.6***	***0.65***	***0.7***	***0.75***	***0.8***	***0.85***	***0.9***	***0.95***	**Avg**
Denomination											
100Dollar	50.00%	50.00%	50.00%	50.00%	50.00%	37.50%	20.83%	4.17%	0.00%	0.00%	31.25%
10Dollar	100.00%	100.00%	100.00%	100.00%	82.81%	82.81%	40.10%	22.92%	0.00%	0.00%	62.86%
1Dollar	100.00%	100.00%	100.00%	68.15%	25.93%	11.11%	1.23%	1.23%	0.00%	0.00%	40.77%
20Dollar	88.47%	88.47%	88.47%	81.80%	61.62%	40.61%	11.92%	0.00%	0.00%	0.00%	46.14%
50Dollar	71.97%	71.97%	71.97%	71.97%	26.01%	15.53%	2.78%	2.78%	0.00%	0.00%	33.50%
5Dollar	100.00%	100.00%	100.00%	75.38%	59.75%	35.00%	0.00%	0.00%	0.00%	0.00%	47.01%
Avg	85.07%	85.07%	85.07%	74.55%	51.02%	37.09%	12.81%	5.18%	0.00%	0.00%	**43.59%**

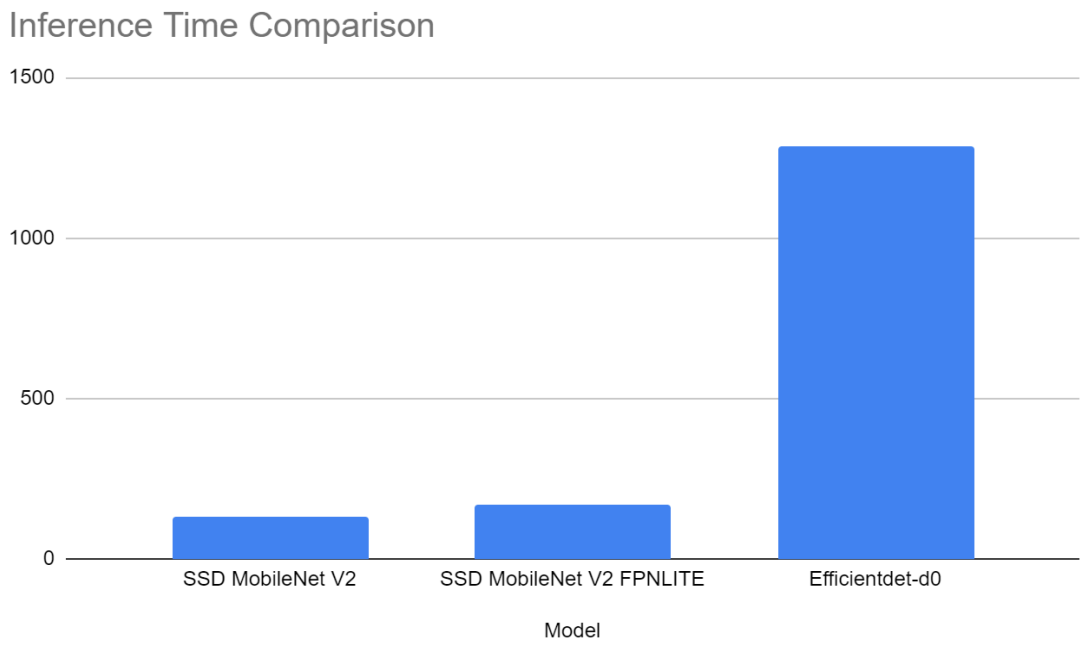
SSD MobileNet V2 FPNLITE											
IoU	***0.5***	***0.55***	***0.6***	***0.65***	***0.7***	***0.75***	***0.8***	***0.85***	***0.9***	***0.95***	**Avg**
Denomination											
100Dollar	100.00%	100.00%	100.00%	100.00%	100.00%	43.75%	7.29%	4.17%	0.00%	0.00%	55.52%
10Dollar	100.00%	87.50%	87.50%	87.50%	87.50%	66.07%	66.07%	28.57%	7.14%	0.00%	61.79%
1Dollar	88.89%	88.89%	88.89%	40.87%	40.87%	1.85%	1.85%	1.85%	0.00%	0.00%	35.40%
20Dollar	87.65%	87.65%	87.65%	77.78%	77.78%	43.49%	14.29%	0.00%	0.00%	0.00%	47.63%
50Dollar	31.67%	31.67%	31.67%	31.67%	31.67%	13.89%	8.33%	8.33%	8.33%	0.00%	19.72%
5Dollar	87.50%	87.50%	87.50%	53.29%	31.93%	15.64%	7.14%	2.50%	0.00%	0.00%	37.30%
Avg	82.62%	80.53%	80.53%	65.19%	61.63%	30.78%	17.50%	7.57%	2.58%	0.00%	**42.89%**

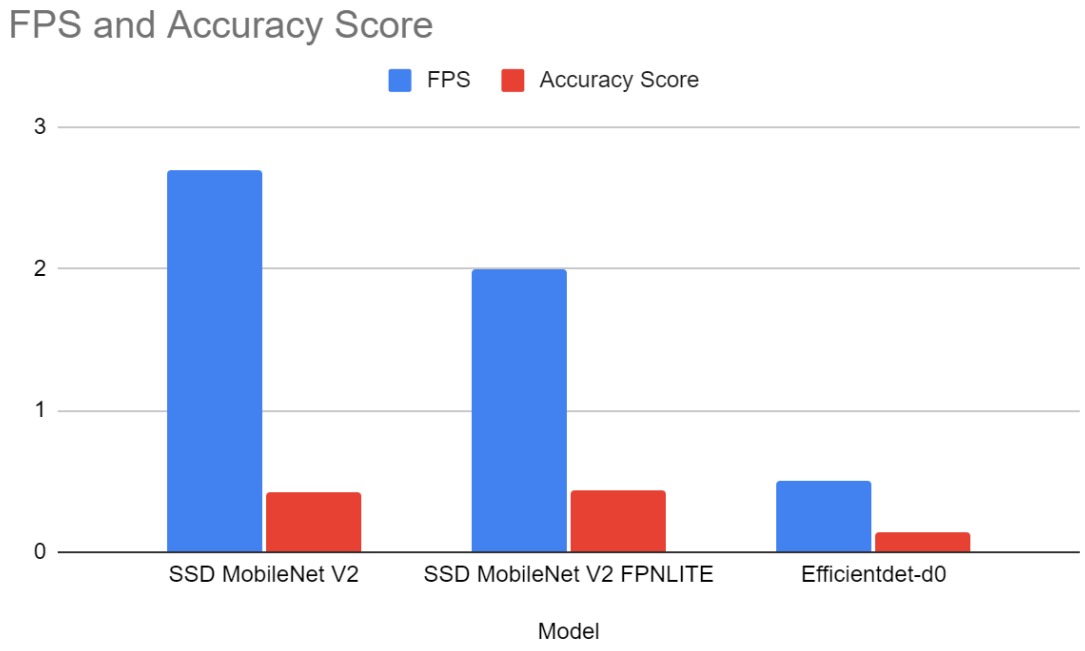
Efficientdet-d0											
IoU	***0.5***	***0.55***	***0.6***	***0.65***	***0.7***	***0.75***	***0.8***	***0.85***	***0.9***	***0.95***	**Avg**
Denomination											
100Dollar	12.50%	12.50%	12.50%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	3.75%
10Dollar	46.88%	46.88%	46.88%	25.37%	20.01%	9.37%	0.69%	0.00%	0.00%	0.00%	19.61%
1Dollar	11.11%	11.11%	11.11%	11.11%	11.11%	11.11%	0.00%	0.00%	0.00%	0.00%	6.67%
20Dollar	13.68%	13.68%	13.68%	4.04%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	4.51%
50Dollar	98.72%	98.72%	84.94%	63.21%	6.85%	2.08%	0.00%	0.00%	0.00%	0.00%	35.45%
5Dollar	25.00%	25.00%	25.00%	25.00%	25.00%	0.00%	0.00%	0.00%	0.00%	0.00%	12.50%
Avg	34.65%	34.65%	32.35%	21.45%	10.50%	3.76%	0.12%	0.00%	0.00%	0.00%	**13.75%**

Speed Measurements

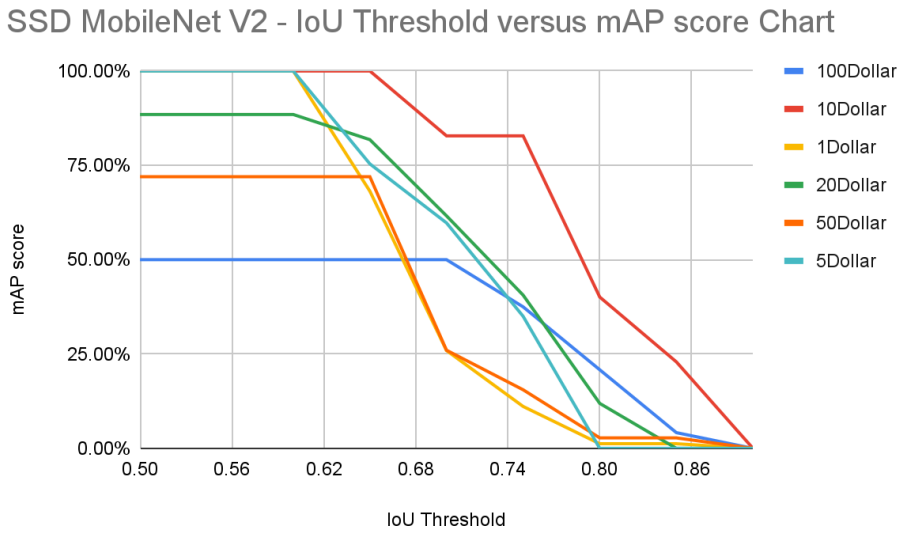
Model	Inference Time (ms)	FPS	Accuracy Score
SSD MobileNet V2	170.21	2.7	43.59%
SSD MobileNet V2 FPNLITE	134.51	2.0	42.89%
Efficientdet-d0	1284.34	.5	13.75%

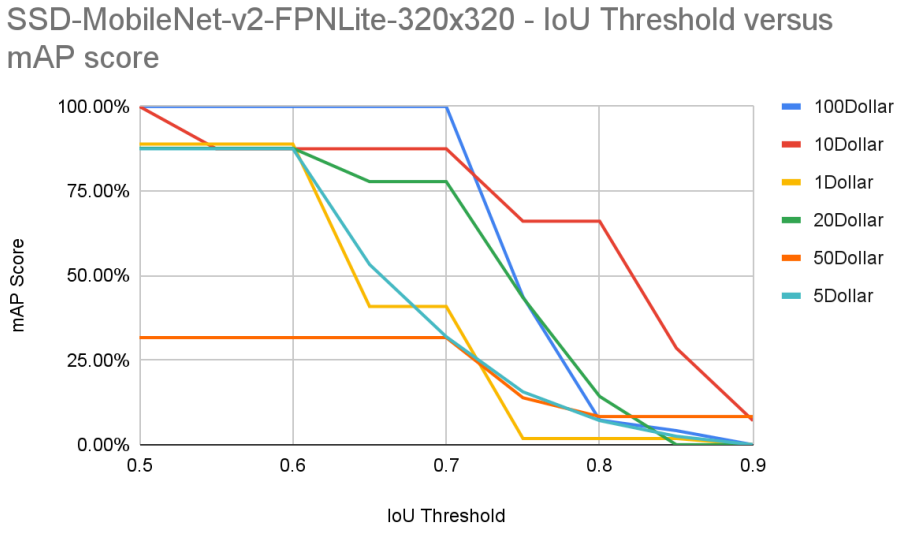
Overall speed and accuracy Graphs

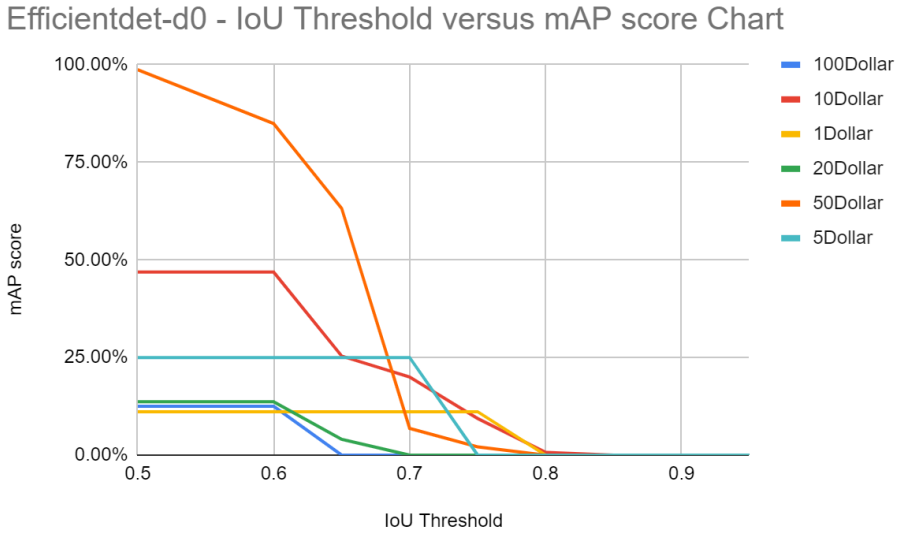




IoU Threshold versus mAP Graphs







Conclusion

SSD-MobileNet-v2

When I ran the tests, the SSD-MobileNet-v2 showed great results. I only trained the model for 40,000 steps yet the model had a decent mAP score along with a great inference speed. With an inference time of 170.21 ms, an overall mAP score of 42.89%, and the best FPS rate of 2.7, the quantized version of this model strikes a good balance between speed and accuracy. When dealing with 100-dollar bills, the mAP score of the model was around 50% when dealing with low IoU thresholds, but as the IoU threshold grew, the mAP score saw a steep drop. With 10 dollar bills, the model did exceptionally with a 100% mAP rate from 0.5 to 0.65 IoU threshold and continued a strong score past the 0.7 threshold, with a slow decline. It was a very similar story to the 1 dollar bill which saw one of the least steep declines with good mAP scores around the 0.65-0.7 threshold. The 20-dollar bill had the most steady scores. The scores were very similar and ideal to the 0.75 threshold. The 50-dollar bill also had consistent scores with mid to low scores until the 0.75 IoU threshold. The 5-dollar bill had a great start with a 100% mAP score but had the steepest decline reaching a 0% mAP score before any other denomination.

SSD-MobileNet-v2-FPNLite-320x320

SSD-MobileNet-v2-FPNLite-320x320 was the most surprising model. The model had a similar overall mAP score of 42.89% to its non-Lite counterpart. It also had the fastest speed 134.51 ms and a FPS of 2.0. When tackling the $100 bill, the model had great accuracy with a 100 mAP

score till the 0.7 IoU threshold with a slow decline in mAP score after that. The 10-dollar bill also had consistent results with the slowest decline and the model was even able to accurately predict some of the bills at the 0.9 threshold. The 1 dollar bill had good results as well with accurate scores, but a steep decline. There was a similar story with the $20 bill. The 50-dollar bill had the overall worst mAP scores, yet it had an overall slow decline in the mAP score as the model was able to guess some correctly even around the 0.9 IoU threshold. The 5-dollar bill had good accurate results as well with a slow and steady decline in mAP score towards higher thresholds.

Efficientdet-d0

The Efficient-d0 was the worst model in terms of both speed and accuracy. The speed of the model was leagues behind the others at 1284.34 ms and an FPS of 0.5. The overall mAP score was around 13.75%. With the 100-dollar bill, the model did the worst with an initial mAP score of 12.50%, but later it had a mAP score of zero only at the 0.65 IoU threshold. The model also struggled with the 10-dollar bill with low rates, but it still was able to guess some right up to the 0.8 threshold. The 1-Dollar had low, but consistent results with around a 11.11% mAP score. The 20 Dollar bill had similar results with low, but consistent rates. However, with the 50-dollar bill, the model had a much better initial score of 98.72%, but the score would decline rather quickly as the threshold grew. Finally, with the $5 bill, the model had a consistent rate of 25.00%, but it would be 0% by the 0.75 IoU threshold.

Application

Overall, my hypothesis was partially correct. I was correct in predicting that the SSD-MobileNet-v2-FPNLite-320x320 would have the fastest speeds due to its FPN-lite incorporation and its MobileNet backbone. I was also correct in predicting the order of the next two being the SSD-MobileNet-v2 and then Efficientdet-d0. However, I was incorrect in predicting that the Efficientdet-d0 was going to be very accurate. It was actually the worst in its mAP score while the SSD-MobileNet-v2-FPNLite-320x320 ended up having a really good score with the SSD-MobileNet-v2 having in all the best scores. This could be possible because due to the low power of the Raspberry PI, the Efficientdet-d0 wasn't able to train correctly and the low amount of power in the hardware led to an overall decline in even accuracy. The other reason could be due to the dataset. When training, the dataset is divided into 80% training, 10% validation, and 10% testing. It is possible that the data chosen for training had a nonproportionate amount of pictures of 50-dollar bills versus other ones. This would also explain the differences between each of the different denominations between all the models.

After considering all the results, I believe that the SSD-MobileNet-FPNLite-320x320 is the most ideal to use on edge devices with low computing power. This model has exceptional accuracy (even when trained on a small dataset) while still running fast enough to achieve near real-time performance. If you need high accuracy and your application can wait a few hundred milliseconds to respond to new inputs, this is the model for you.
SSD-MobileNet-FPNLite-320x320 has great accuracy with a small training dataset and still has near real-time speeds, so it's great for proof-of-concept prototypes. Second, it would be the

SSD-MobileNet-v2. The SSD-MobileNet architecture is popular for a reason. This model has great speed while maintaining the best overall accuracy. If speed is the most crucial aspect of your application, and a good accurate model is needed, this one would be ideal. This leaves the EfficientDet-D0 as the worst-performing out of the three. It has an overall worse speed and accuracy than both of them.

However, it is important to take things with a grain of salt. There is so much to be explored in the world of deep learning. These models can also be altered as well. Possibly the incorporation of the d0 scaling factor in the MobileNet models can help increase their accuracy or the creation of a lite version of the Efficient-d0 model can help it run better on edge devices. This research, while helpful, is a small part of the quest to incorporate deep learning everywhere on the planet. The use of these models in edge devices can help put this technology in areas of the world where access to computing resources is low, yet the need is still present. These models and deep learning in general can be used in areas no matter the resources and can truly pave the future in every industry and place on the planet.

References

1. https://computerhistory.org/blog/how-do-neural-network-systems-work/
2. https://towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba
3. https://www.ibm.com/topics/convolutional-neural-networks
4. https://www.youtube.com/watch?v=5J72iMSQmy8
5. https://arxiv.org/pdf/1801.04381.pdf
6. https://arxiv.org/pdf/1512.02325.pdf
7. https://github.com/saunack/MobileNetv2-SSD?tab=readme-ov-file
8. https://docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/object-detection/mobilenetv2-ssd-fpn
9. https://openaccess.thecvf.com/content_CVPR_2020/papers/Tan_EfficientDet_Scalable_and_Efficient_Object_Detection_CVPR_2020_paper.pdf

Technical Resources

1. https://github.com/EdjeElectronics/TensorFlow-Lite-Object-Detection-on-Android-and-Raspberry-Pi/blob/master/deploy_guides/Raspberry_Pi_Guide.md
2. https://universe.roboflow.com/objectdetection-nhb0l/usd-money/dataset/2

Image Credits

1. Myself
2. https://www.freecodecamp.org/news/want-to-know-how-deep-learning-works-heres-a-quick-guide-for-everyone-1aedeca88076/
3. https://computerhistory.org/blog/how-do-neural-network-systems-work/
4. https://towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba
5. https://towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba

6. https://towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba
7. https://towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba
8. https://towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba
9. https://towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba
10. https://www.ibm.com/topics/convolutional-neural-networks
11. https://paperswithcode.com/method/max-pooling
12. https://medium.com/@techmayank2000/object-detection-using-ssd-mobilenetv2-using-tensorflow-api-can-detect-any-single-class-from-31a31bbd0691
13. https://www.researchgate.net/figure/EfficientDet-D0-architecture-EfficientNet-B0-34-is-the-backbone-network-multiple_fig2_365439360

A Quantitative Study of the Different TensorFlow-Lite Models for Image Detection Using a USD Currency Dataset

Sadhil Mehta
11th Grade - Tippecanoe High School - Tipp City OH 45371

Abstract

In the ever-evolving landscape of computer science, the prominence of deep learning has reached unprecedented levels being integrated into every single part of society. One of the most intriguing parts of deep learning is image detection. Image detection, despite its applications, is limited by the requirement of powerful hardware to run image detection neural networks. I aimed to explore the realm of image detection on edge devices by comparing three different TensorFlow-Lite Networks:

- SSD-MobileNet-v2
- SSD-MobileNet-v2-FPNLite-320x320
- EfficientDet-d0

These networks differ in their network architecture and feature extraction. These networks were compared on their inference speed, accuracy via the mAP measurement, and FPS. All models were trained on the same dataset of US Dollar Bills and then executed on a Raspberry PI by using Python. The hypothesis was that the SSD-MobileNet-v2-FPNLite-320x320 would excel in speed, but lack in accuracy due to its limited input size and a Feature Network Pyramid Lite version. The EfficientDet-d0 was expected to lack in speed, but have good accuracy due to its Bidirectional Feature Pyramid Network and its d0 scaling factor, while SSD-MobileNet-v2 was supposed to be the middle ground. My results were partially correct as the SSD-MobileNet-v2-FPNLite-320x320 was the fastest, while the SSD-MobileNet-v2 was the most

accurate. The limited computational power of the Raspberry PI hampered the EfficientDet-d0's performance.

In all, I was able to prove that these models could be run on virtually any low-power device with decent accuracy and speed and offered insight into which models can excel in these conditions.