Vehicular Detection and Localization Using Convolutional Neural Networks

(Interim Report)

Murthy Vaddadi

Illinois Institute of Technology, Chicago
Chicago, Illinois, United States

Email: aditya.murthy1994@gmail.com

Abstract—**Vehicle identification via computer vision is an important task for traffic control, video surveillance, security and authentication. However, there are challenges with image classification of vehicles due to the fine-grained details that are inherently harder for a computer to detect. Using limited data from the Stanford Cars dataset [7], we implement transfer learning on a pre-trained Convolutional Neural Network (CNN) framework to classify vehicles based on 196 classes of different vehicle makes, models and years.**

Keywords— Vehicle Identification, Computer Vision, Fine-Grained Details, Transfer Learning, Convolutional Neural Networks

I. Introduction

Cars are ubiquitous in our daily lives as a means of mobility and transportation. As a result, many different models of cars exist around the world, catering to the specific preferences of drivers in terms of aesthetics and functionality. With around a billion vehicles on the planet, each having their own distinct physical features, the need for efficient and robust identification of such objects has increased over the years. Vehicle recognition has a wide range of applications, such as video surveillance, security, authentication, and even accessibility features for ridesharing apps. In computer vision, fine-grained classification can be a very challenging problem because differences between classes tend to be much more nuanced compared to class differences in tasks such as object detection. In this case, fine-grained classification is defined to be classification with categories that share similar basic features. Cars are a perfect example of this, as they share a common physical structure, but can differ widely in physical sub-features such as height, width, wheel type, etc. Our classification problem is accurately discerning a car's make and model from a digital image of the car. Humans are inherently good at vehicle classification, as they can pick up on small details such as a logo, or some lettering on the vehicle. However, this task has been difficult for computers due to the fine-grained nature of the problem.

This project implements several Convolutional Neural Networks (CNNs), with and without using techniques such as transfer learning to accurately classify cars from an image dataset by vehicle make and model. The input to our algorithm is a set of images from the Stanford Cars dataset, which contains 196 classes of cars. We then use a CNN to output a prediction for the input image. The extreme complexity of the problem, minuscule inter-class separation and a small volume of data in the dataset are some of the main challenges we set to tackle.

II. Related Work

Fine-grained vehicle classification has recently become a popular topic of study amongst computer scientists due to the availability of image datasets and the exponential increase in the computational power available. CNNs are a popular choice for general object recognition/classification, as shown by their ability to achieve state of the art accuracy on generic image classification. Specifically, convolutional neural networks are an effective tool for computer vision and image recognition due to their ability to analyse spatial coherence of images and reduce the number of trainable parameters. We now go over some specific applications of CNNs on fine-grained image classification in the past.

In a survey paper, Zhao et al. proposed an "Ensemble of Network-Based Approach" [2] for subset feature learning. The approach implements two main parts:

- A domain generic convolutional neural network (pre-trained on large-scale dataset of same domain as target dataset and then fine-tuned on the target).
- Several specific convolutional neural networks.

In terms of real-life implementation, Xie et al. [3] used a network of five convolutional layers and two fully connected layers for vehicle recognition on the Stanford Cars dataset. The study utilised hyper-class data augmentation, which augments fine-grained data with a large number of auxiliary images labelled by some hyper-classes. The study achieved an accuracy of 80% with multitask learning, and 86% with the model pre-trained on ImageNet with multitask learning.

Another study used a two-part model:

- VGG16 network structure for vehicle detection of images with complex backgrounds.
- A CNN with a joint Bayesian network for classification.

VGG 16 is a CNN architecture that features 13 convolutional layers and 3 fully connected layers.

Transfer learning is a popular choice for vehicle recognition tasks that use large CNN architectures. In transfer learning, a base network is pre-trained on a dataset to create weights and features. This network, with its trained weights, is then transferred to a new dataset by retaining a subset of the base network's learned weights and features. The overall effect is a classifier that fits the new dataset with state-of-the-art results. One study [4] utilised transfer learning to make predictions on the Stanford Cars dataset, in which a fine-tuned VGGNet model was able to achieve 78.9% accuracy, and 94.2% top-5 accuracy. The top model was GoogLeNet [5] full train which achieved 80.0% accuracy and 95.01% top-5 accuracy.

III. Dataset And Features

We used the Stanford Cars dataset to train and evaluate our vehicle classification models. The dataset contains 16,185 image classification pairs of 196 difference classes. Each of the fine-grained 196 classes is determined by year, make and model of a vehicle. First, we should import and check the original images in the dataset.

Setting up the directories

```
data_dir = 'Dataset'

train_dir = os.path.join(data_dir, 'Car Images', 'Train Images')
test_dir = os.path.join(data_dir, 'Car Images', 'Test Images')

train_annotations_path = os.path.join(data_dir,'Annotations','Train Annotations.csv')
test_annotations_path = os.path.join(data_dir,'Annotations','Test Annotation.csv')
```

Making a new directories

```
data_dir = 'Dataset'

if 'Augmented Train Images' not in os.listdir(os.path.join(data_dir, "Car Images")):
     aug_dir = os.path.join(data_dir, "Car Images", "Augmented Train Images")
     os.mkdir(aug_dir)
     if 'augmented_annotations.csv' not in os.listdir(os.path.join(data_dir,
"Annotations")):
             print("CSV file doesn't exist")

     print("Made directory 'Augmented Train Images'")

aug_dir = os.path.join(data_dir, 'Car Images', 'Augmented Train Images')

aug_annotations_path = os.path.join(data_dir, "Annotations", "Train Augmented
Annotations.csv")
```

Importing the CSV

```
annotation_cols = ["Image Name", "BB x1", "BB y1", "BB x2", "BB y2","Image class"]

train_annotation = pd.read_csv(train_annotations_path, header = 0, names =
annotation_cols)

test_annotation = pd.read_csv(test_annotations_path, header = 0, names = annotation_cols)

car_names_make = pd.read_csv(os.path.join(data_dir, 'Car names and make.csv'), header =
None, names = ["Car Name and Make"])
```

Function to get image path by index

```
def get_path(index, data_type = 0):
     if data_type == 0: # Train Images
             data_dir = train_dir
             data_annotation = train_annotation
     elif data_type == 1: # Test Images
             data_dir = test_dir
             data_annotation = test_annotation
     elif data_type == 2: # Augmentation Train Images
             data_dir = aug_dir
             data_annotation = aug_annotation
             img_file = os.path.join(data_dir, data_annotation["Image Name"][index])
             return img_file
     else:
             print("Please choose 0 for train set, 1 for test set, or 2 for augmented
train set only only.")
     return

     file_path = os.path.join(data_dir, car_names_make.loc[int(data_annotation["Image
class"][index])-1][0])
     img_file = os.path.join(file_path, data_annotation["Image Name"][index])
     return img_file
```

Function to display image by index

```
def show_car(index, data_type = 0):

     if data_type == 0:
             img_path = get_path(index, data_type)
             data_annotation = train_annotation
     elif data_type == 1:
             img_path = get_path(index, data_type)
             data_annotation = test_annotation
     elif data_type == 2:
             img_path = get_path(index, data_type)
             data_annotation = aug_annotation
     else:
             print("Please choose 0 for train set, 1 for test set, or 2 for augmented
train set only only.")
             return

     img = imread(img_path)

     x = data_annotation.loc[index]["BB x1"]
     y = data_annotation.loc[index]["BB y1"]
     w = data_annotation.loc[index]["BB x2"] - x
     h = data_annotation.loc[index]["BB y2"] - y

     fig, ax = plt.subplots(1)
     plt.imshow(img)
     rect = patches.Rectangle((x, y), w, h, linewidth = 2, edgecolor = 'r', facecolor =
'none', label = data_annotation.loc[index]["Image Name"])
     ax.add_patch(rect)

     plt.show()
     print(data_annotation.loc[index]["Image class"])
     print(img.shape)
     print("x: ", x, "y: ", y, "w: ", w, "h: ", h)
```

Let's see some random cars

```
Show_car(69)
Show_car(109)
Show_car(1109)
```

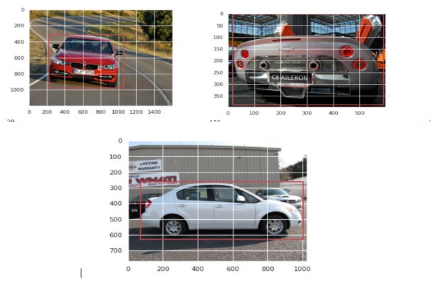


Fig. 1. BMW 3 Series Sedan 2012 (*index number 69*), Spyker C8 Coupe 2009 (*index number 109*) and Suzuki SX4 Sedan 2012 (*index number 1109*)

As shown above, each image contains a car in the foreground against various backgrounds viewed at different angles. The quality of the image, as determined by the camera used, lighting, focal length, and positioning, varies from image to image. Some images are professionally taken shors, while others are relatively low quality images taken from classfied ads on the internet.

Distribution of classes across the dataset

```
sns.set(rc = {'figure.figsize': (16, 9), 'xtick.labelsize': 6})
plt.xticks(rotation = 90)
plt.tight_layout()
sns.countplot(x = train_annotation["Image class"]);
plt.rcParams.update(plt.rcParamsDefault)
```

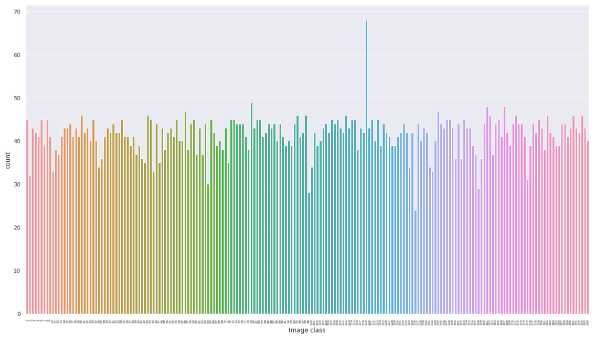


Fig. 2. On average each class has 41.4 images each with an exception of class number 119 *(GMC Yukon Hybrid SUV 2012)*, which has 68 images.

In the provided *'Cars name and make.csv'* that provides mapping of the class numbers to car names and make had an error in the naming of the class index 173 (Ram C-V Cargo Van Minivan 2012), which needed to be fixed.

Fixing the index number 173 in *'Cars name and make'*

```
car_names_make.loc[173][0] = "Ram C-V Cargo Van Minivan 2012"
```

We also needed to verify if there are any other errors in the dataset other that this. This was done via a function that scanned throught all of training dataset and verified the size and the positioning of the bounding box within the image was proper.

Function to check the bounding boxes

```
def compare_BB_size(index, data_type = 0, threshold_max = 0.99, threshold_min = 0.05):
     if data_type == 0:
             img_path = get_path(index, data_type)
             data_annotation = train_annotation
     elif data_type == 1:
             img_path = get_path(index, data_type)
             data_annotation = test_annotation
     elif data_type == 2:
             img_path = get_path(index, data_type)
             data_annotation = aug_annotation
     else:
             print("Please choose 0 for train set, 1 for test set, or 2 for augmented
train set only only.")
             return

     img = imread(img_path)

     img_area = img.shape[0] * img.shape[1]

     x = data_annotation.loc[index]["BB x1"]
     y = data_annotation.loc[index]["BB y1"]
     w = data_annotation.loc[index]["BB x2"] - x
     h = data_annotation.loc[index]["BB y2"] - y

     BB_area = w * h

#    if(BB_area / img_area > threshold_max):
#            print("%d index car's bounding box is way too big, please verify manually"
%i)
     if(BB_area / img_area < threshold_min):
             print("%d index car's bounding box is way too small, please verify manually"
%i)
     if(x < 0 or y < 0 or h > img.shape[0] or w > img.shape[1]):
             print("%d index car's bounding box is at the edge of the image, please
verify manually" %i)
             print("x: ", x, "y: ", y, "w: ", w, "W: ", img.shape[1], "h: ", h, "H: ",
img.shape[0])
```

This resulted in pointing out two anomalies, one of which didn't need fixing, but the other did as the image of the car was rotated but the bounding box was not.

```
Out [20]
5596 index car's bounding box is way too small, please verify manually
7388 index car's bounding box is at the edge of the image, please verify manually
x:  56 y:  89 w:  661 W:  576 h:  430 H:  768
```

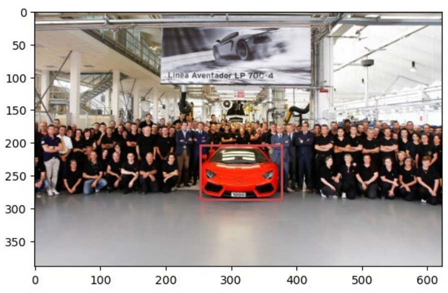


Fig. 3. 5596 car index spans less than 10% of the entire image.

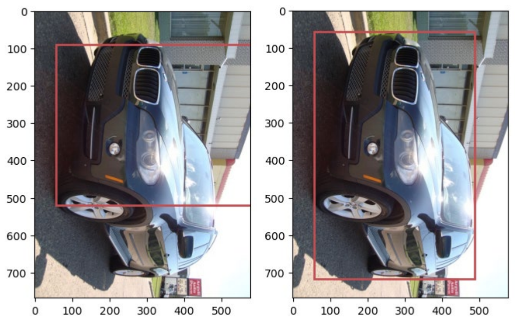


Fig. 4. 7388 car index had misaligned bounding box which was fixed by rotating the bounding box by 90 degrees with respect to the origin (top left corner of the image).

```
# to rotate bounding box clockwise, make x1' = W - y2, x2' = W - y1, y1' = x1, y2' = x2
img = imread(get_path(7388))
W = img.shape[1]
x1_new = W - train_annotation["BB y2"][7388]
x2_new = W - train_annotation["BB y1"][7388]
y1_new = train_annotation["BB x1"][7388]
y2_new = train_annotation["BB x2"][7388]
train_annotation["BB x1"][7388] = x1_new
train_annotation["BB y1"][7388] = y1_new
train_annotation["BB x2"][7388] = x2_new
train_annotation["BB y2"][7388] = y2_new
```

IV. Data Augmentation

Since we do not have nearly enough images per class to train a CNN properly, we needed to perform some augmentations to the train data to create variations of the existing images from it. This ended up increasing the size of the available training data to us by 6 folds and each train image was augmented 5 times. Use of affine transformation was restricted to minimal since they do not work well with bounding boxes. A python library called ImgAug was used to perform the following augmentations.

- Flip right or left
- Flip up or down
- Crop the image by up to 10%
- Scale the image on x or y axis by -20%
- Replace *Superpixels* in the image with noise
- Add gaussian, average, or median blur
- Sharpen the image
- Emboss the image
- Add edge detection or direct edge detection filters
- Add guassian noise
- Drop some parts of the image
- Invert the colours of the image
- Add random values to random pixels
- Multiply random pixels with random values
- Add linear contrast
- Apply elastic transformation
- Apply piecewise affine transformations

Note: Not all of these transformations were applied on every iteration, and to all of the images to create varying outputs.

```
aug_seq = iaa.Sequential([iaa.Fliplr(0.5),
                          iaa.Flipud(0.2),
                          iaa.Sometimes(0.5, iaa.Crop(percent = (0, 0.1))),
#                           iaa.Sometimes(0.5, iaa.GaussianBlur(sigma = (0, 0.5))),
                          iaa.Affine(
                              scale = {"x": (0.8, 1.0), "y": (0.8, 1.0)}, # limit to 100
#                               translate_percent = {"x": (-0.2, 0.2), "y": (-0.2, 0.2)},
# remove maybe?
#                               rotate = (-25, 25),
#                               shear = (-8, 8)
                          ),
                         iaa.SomeOf((0, 5),
                                   [
                                       iaa.Sometimes(0.5, iaa.Superpixels(p_replace = (0,
1.0),
                                                                            n_segments = (20, 200))),
                                       iaa.OneOf([
                                           iaa.GaussianBlur((0, 0.3)),
                                           iaa.AverageBlur(k = (2, 7)),
                                           iaa.MedianBlur(k = (3, 11)),
                                       ]),
                                       iaa.Sharpen(alpha = (0, 1.0), lightness = (0.75,
1.5)),

                                       iaa.Emboss(alpha = (0, 1.0), strength = (0, 2.0)),
                                       iaa.Sometimes(0.5, iaa.OneOf([
                                           iaa.EdgeDetect(alpha = (0, 0.7)),
                                           iaa.DirectedEdgeDetect(alpha = (0, 0.7),
direction = (0.0, 1.0)),
                                       ])),
                                       iaa.AdditiveGaussianNoise(loc = 0, scale = (0.0,
0.05 * 255), per_channel = 0.5),

                                       iaa.OneOf([
                                           iaa.Dropout((0.01, 0.1), per_channel = 0.5),
                                           iaa.CoarseDropout((0.03, 0.15), size_percent =
(0.02, 0.05), per_channel = 0.2),

                                       ]),
                                       iaa.Invert(0.05, per_channel = True),
                                       iaa.Add((-10, 10), per_channel = 0.5),
                                       iaa.Multiply((0.8, 1.0), per_channel = 0.2),
                                       iaa.LinearContrast((0.75, 1.5)),
#                                        iaa.Grayscale(alpha = (0.0, 0.7)),
                                       iaa.Sometimes(0.5, iaa.ElasticTransformation(alpha
= (0.5, 2.5), sigma = 0.25)),

                                       iaa.Sometimes(0.5, iaa.PiecewiseAffine(scale =
(0.01, 0.05)))

                                   ],
                                   random_order = True)], random_order = True)
```

A new DataFrame was created to store the image names and the classes of these augmented images, as well as to store the coordinates of the bounding boxes after augmentation. This DataFrame was subsequently then stored in *'Train Augmented Annotations.csv'* later.

```
aug_annotation = pd.DataFrame(columns = ["Image Name", "BB x1", "BB y1", "BB x2", "BB y2",
"Image class"])
```

A for loop was used to iterated through all the training images and another to run augmentations on them 5 times. The results of which were then pushed to the *'aug_annotation'* DataFrame and to the disk to be used later. This loop was also used to resize the images to a size of 224×224 to be inputted directly into our models.

```
if len(os.listdir(aug_dir)) < 45000:
     for i in range(len(train_annotation)):
             for j in range(5):
                             image = imread(get_path(i))
                             imageH = image.shape[0]
                             imageW = image.shape[1]
                             image = ia.imresize_single_image(image, (224, 224))
                             bbs = BoundingBox(x1 = train_annotation.loc[i]["BB x1"] * 224 /
imageW, x2 = train_annotation.loc[i]["BB x2"] * 224 / imageW, y1 =
train_annotation.loc[i]["BB y1"]  * 224 / imageH, y2 = train_annotation.loc[i]["BB y2"]  *
224 / imageH, label = car_names_make.loc[int(train_annotation["Image class"][i])-1][0])
            image_aug, bbs_aug = aug_seq(image = image, bounding_boxes = bbs)
            bbs_aug = bbs_aug.clip_out_of_image(image)

                             imageio.imwrite(os.path.join(aug_dir, "aug" + str(j) + "_" +
train_annotation["Image Name"][i]), image_aug)
                             df2 = {"Image Name": "aug" + str(j) + "_" +
train_annotation["Image Name"][i], "BB x1": bbs_aug[0][0], "BB y1": bbs_aug[0][1], "BB
x2": bbs_aug[1][0], "BB y2": bbs_aug[1][1], Image class": train_annotation["Image
class"][i]}
                             aug_annotation = aug_annotation.append(df2, ignore_index =
True)

                             if j == 0:
                             imageio.imwrite(os.path.join(aug_dir, "%s"
%(train_annotation["Image Name"][i],)), image)
                             df2 = {"Image Name": train_annotation["Image Name"][i], "BB
x1": bbs[0][0], "BB y1": bbs[0][1], BB x2": bbs[1][0], BB y2": bbs[1][1], "Image class":
train_annotation["Image class"][i]}
                             aug_annotation = aug_annotation.append(df2, ignore_index =
True)

else:
     aug_annotation = pd.read_csv(os.path.join(data_dir, "Annotations", "Train Augmented
Annotations.csv"))
```

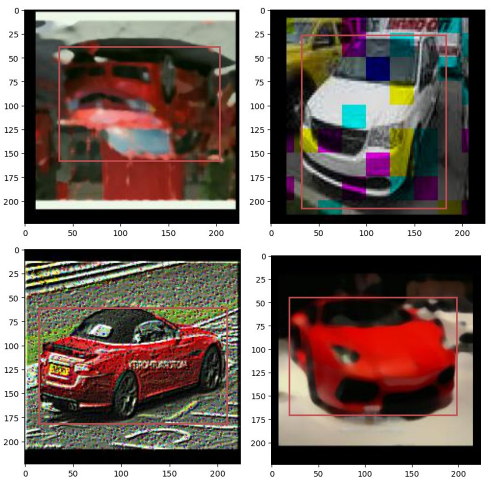


Fig. 5. Some examples of car images after augmentation.

V. METHODS

While we have finished some pre-processing, we are now ready to fit them into the models. We will develop a shallow CNN as a baseline for non-transfer learning. And we will config several deep learning models for comparisons. As the dataset contains about 16000 images, it is not

large enough to fully train the deep learning models. In addition, due to the complexity of deep learning neural networks, we would perform the transfer learning to train those models. Transfer learning is a research problem in machine learning that focuses on storing knowledge gained

while solving one problem and applying it to a different but related problem. It is an optimization that allows rapid progress or improved performance when modelling the second task. Therefore, it is popular in deep learning given the enormous resources required to train deep learning models or the large and challenging datasets on which deep learning models are trained. In this project, we implement the following 2 deep learning networks: ResNet and VGG.

A. *Shallow CNN (Baseline)*

The baseline model is a simple convolutional neural network, the structure is as below:

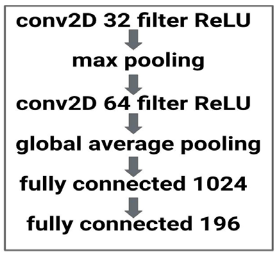


Fig. 6. Structure of the Baseline.

B. *MobileNet*

MobileNet is a class of efficient model for mobile and embedded version of applications. The main idea of MobileNet is to divide traditional convolution layers into depth wise convolution layer and pointwise convolution layer. Consequently, the computation is reduced dramatically thanks to smaller number of parameters compared to VGG.

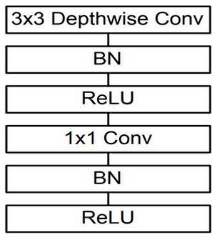


Fig. 7. Architecture of MobileNet.

C. *VGG16*

The VGG network architecture was introduced by Simonyan and Zisserman in 2014. This network is characterized by its simplicity, using only 3×3 convolutional layers stacked on top of each other in increasing depth. Reducing volume size is handled by max pooling. Two fully connected layers, each with 4,096 nodes are then followed by a softmax classifier. The "16" stands for the number of weight layers in the network. Aside from the straight-forward architecture, VGGNet consists of 138 million parameters, which is challenging to train. In order to make training easier, VGG needs to be trained from smaller networks and use them as initialization (pre-training). While the whole VGGNet is slow to train, we will use the pre-trained models. The figure below shows the architecture of VGG.

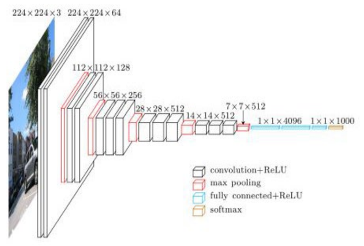


Fig. 8. A visualisation of the architecture of VGG Network.

D. *ResNet*

Kaiming He et al introduced Residual Neural Network, the so-called ResNet, in 2015. It is well known as the solution to solve the network degrading problem. First, we need to take a look at the degradation problem. When deeper networks start converging, with the network depth increasing, accuracy gets saturated and then degrades rapidly. As a result, naively adding layers would lead to higher training errors. In some worst-case scenarios, while accuracy of shallow networks is saturated, the deeper networks is degrading the model. To solve this, the authors introduced the Residual function using F(x)=H(x)-x instead of a direct mapping of H(x). In the network, this is transferred to a Residual block:

$$y = F(x,\ \{W_i\}) + x \tag{1}$$

While the function F (x, {Wi}) is the residual mapping function we need to train.

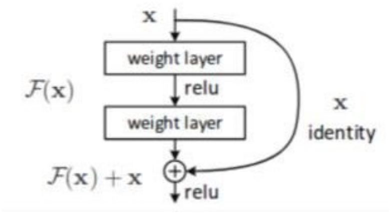

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Fig. 9. A building block of the Residual Network *(ResNet)*.

The design of ResNet networks uses 3×3 filters mostly, down sampling with CNN layers with stride 2 and consists of global average pooling layer and a 1000-way fully connected layer with Softmax in the end.

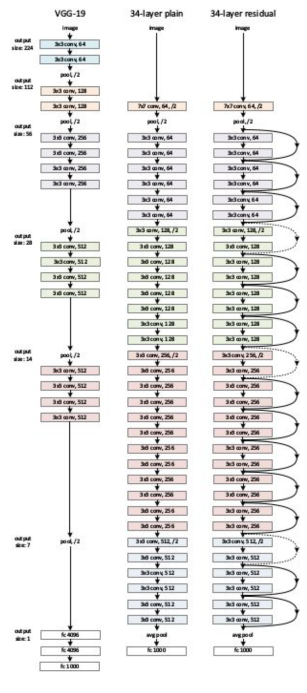
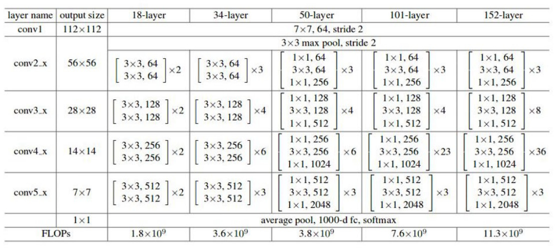


Fig. 10. Plain VGG and VGG with Residual Blocks.

The figure above shows a plain 34-layer network and a 34-layer residual network. From testing, the plain 34-layer network has higher validation error than the 18 layers plain network. This is where we realize the degradation problem. And the same 34-layer network when converted into the residual network has much lesser training error than the 18-layer residual network. Thus, ResNet models can have an incredibly high depth up to 152. In this project, we will implement several ResNet models with different depths.



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig.11.ResNet Architecture with different depths of layer

E. *EfficientDet*

EfficientDet Based on our BiFPN, we have developed a new family of detection models named EfficientDet. In this section, we will discuss the network architecture and a new compound scaling method for EfficientDet. EfficientDet Architecture Figure 16 shows the overall architecture of EfficientDet, which largely follows the one-stage detectors paradigm [27, 33, 23, 24]. We employ ImageNet-pretrained EfficientNets as the backbone network. Our proposed BiFPN serves as the feature network, which takes level 3-7 features {P3, P4, P5, P6, P7} from the backbone network and repeatedly applies top-down and bottom-up bidirectional feature fusion. These fused features are fed to a class and box network to produce object class and bounding box predictions respectively. Like, the class and box network weights are shared across all levels of features.

Compound Scaling Aiming at optimizing both accuracy and efficiency, we would like to develop a family of models that can meet a wide spectrum of resource constraints. A key challenge here is how to scale up a baseline EfficientDet model. Previous works mostly scale up a baseline detector by employing bigger backbone networks (e.g., ResNeXt or AmoebaNet), using larger input images, or stacking more FPN layers. These methods are usually ineffective since they only focus on a single or limited scaling dimensions. Recent work shows remarkable performance on image classification by jointly scaling up all dimensions of network width, depth, and input resolution. Inspired by these works, we propose a new compound scaling method for object detection, which uses a simple compound coefficient φ to jointly scale up all dimensions of backbone, BiFPN, class/box network, and resolution. Unlike, object detectors have much more scaling dimensions than image classification models, so grid search for all dimensions is prohibitive expensive. Therefore, we use a heuristic-based scaling approach, but still follow the main idea of jointly scaling up all dimensions.

Backbone network – we reuse the same width/depth scaling coefficients of EfficientNet-B0 to B6 such that we can easily reuse their ImageNet-pretrained checkpoints.

BiFPN network – we linearly increase BiFPN depth D bifpn (#layers) since depth needs to be rounded to small integers. For BiFPN width W bifpn (#channels), exponentially grow BiFPN width W bifpn (#channels) as similar to [39]. Specifically, we perform a grid search on a list of values {1.2, 1.25, 1.3, 1.35, 1.4, 1.45}, and pick the best value 1.35 as the BiFPN width scaling factor. Formally, BiFPN width and depth are scaled with the following equation: W bifpn = 64 · 1.35φ, Dbifpn = 3 + φ

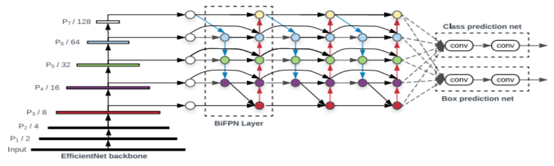


Fig.12.Figure 3: EfficientDet architecture – It employs EfficientNet as the backbone network, BiFPN as the feature network, and shared class/box prediction network.

REFERENCES

[1] 3D Object Representations for Fine-Grained Categorization Jonathan Krause, Michael Stark, Jia Deng, Li Fei-Fei *4th IEEE Workshop on 3D Representation and Recognition, at ICCV 2013* (3dRR-13). Sydney, Australia. Dec. 8, 2013. [pdf] [BibTex] [slides]

[2] Ying Zhao, Jun Gao and Xuezhi Yang, "A survey of neural network ensembles," 2005 International Conference on Neural Networks and Brain, 2005, pp. 438-442, doi: 10.1109/ICNNB.2005.1614650.

[3] Xie, Saining, et al. "Hyper-class augmented and regularized deep learning for fine-grained image classification." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015.

[4] Liu, Derrick, and Yushi Wang. "Monza: image classification of vehicle make and model using convolutional neural networks and transfer learning." (2017).

[5] C. Szegedy et al., "Going deeper with convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 1-9, doi: 10.1109/CVPR.2015.7298594.

[6] Chollet, F. 2016, VGGl6, VGGl9, and ResNet50, v0.I, https://github.com/fchollet/deep-learning-models/releases/tag/v0. I

[7] Katanforoosh, K. 2018, CS230 Code Examples, https://github.com/kiank/cs230-code-examples

[8] Wang, X. 2019. Fine Tune VGG Networks Based on Stanford Cars.
https://github.com/Xiaotian-WANG/Fine-Tune-VGG-Networks-Based -on-Standford-Cars