Machine Learning 2 Final Report

Zhuohan Yu, Zeqiu Zhang

Introduction

Distracted driving is one of the main reasons for car accidents. According to CDC's data, it causes about 425,000 people injured and 3,000 people killed every year. In order to improve these alarming statistics, and better insure drivers, by testing whether dashboard cameras can automatically detect drivers engaging in distracted behaviors.

This project aims at predicting drivers' 10 possible distraction movements based on the train dataset using Computer Vision models.

In this report, we will first describe the dataset and details of models we used. Then we will show how we set up the experiments like how to implement the network. Afterward, the results will be presented and interpreted. Lastly conclusions will be drawn and we will show possible improvements in the future.

Dataset description

State Farm gave an image dataset including 10 different classes of driver behaviors.

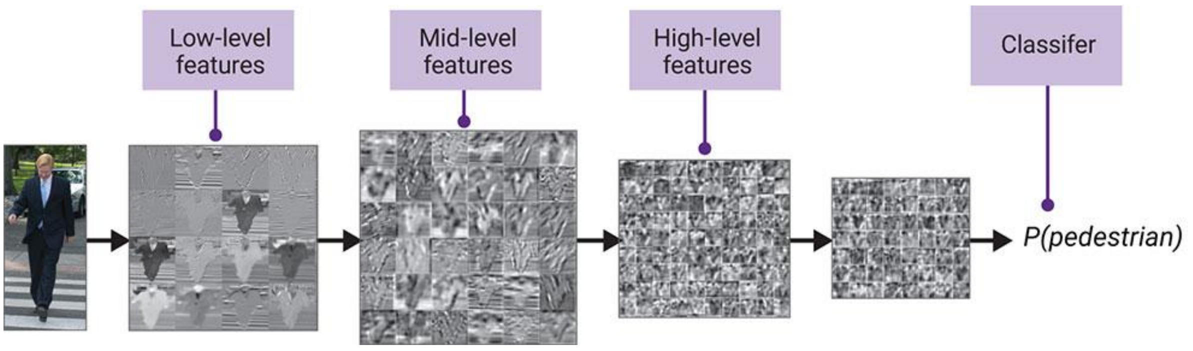
- c0: safe driving
- c1: texting - right
- c2: talking on the phone - right
- c3: texting - left
- c4: talking on the phone - left
- c5: operating the radio
- c6: drinking
- c7: reaching behind
- c8: hair and makeup
- c9: talking to passenger

All images are of the 640*480 pixel size. 19060 of them are train data and 79726 of them are test data. There is also an excel file that records image ids, classnames, and driver ids.

Since the test data size is much larger than the train data, the biggest challenge would be to avoid overfitting.

Model description

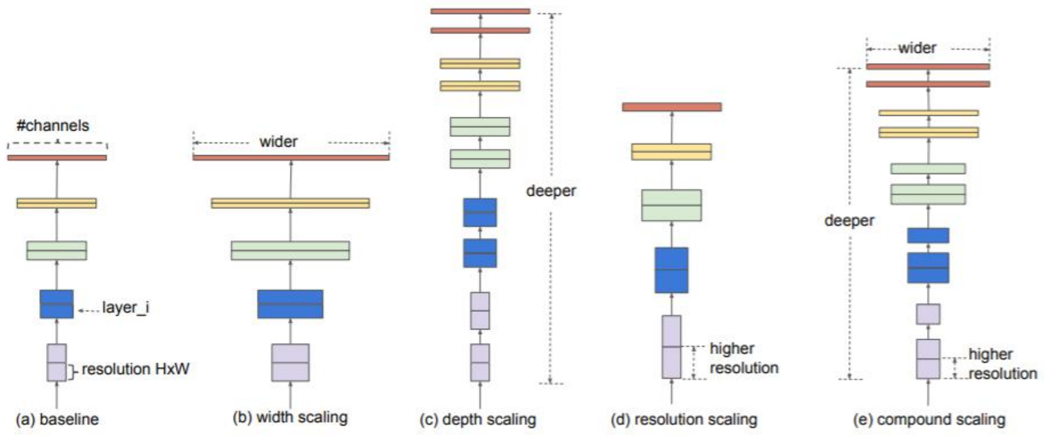
The main idea of our network is transfer learning, so selecting a suitable pretrained model is significant for us. We tried different pretrained computer vision models including VGG, Resnet, DenseNet, EfficientNet, most of which are Convolutional Neural Network (CNN). CNN is known as a desired network architecture to deal with pictures as it could capture important features from the input images by creating feature maps. It works like below:



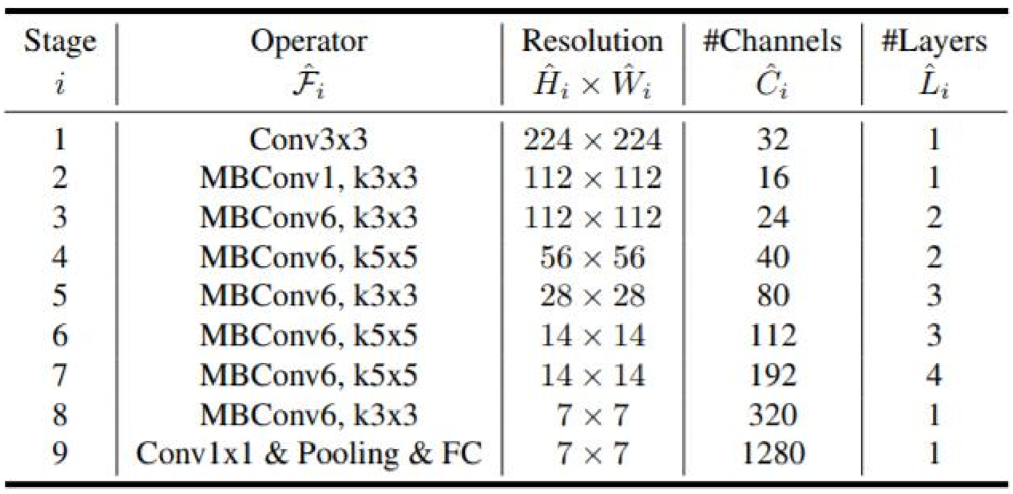
As we can see, CNN could get very detailed features from a picture for classification. Besides, compared to multiple layer perceptron (MLP), CNN could save a lot of memory because the size of a layer would be smaller thanks to convolution calculation. That's why CNN architecture is suitable for picture classification.

After comparing the performance of these different models, we focus on EfficientNetV2B2 and EfficientNetV2B3 as they show better performance. It's not a surprise that EfficientNet has higher performance due to its state of the art architecture. To understand the architecture of EfficientNetV2, we need to focus on EfficientNet first. This network is developed by leveraging a multi-objective neural architecture search which optimizes accuracy and FLOPs.

Compared to other pretrained models, EfficientNet used an efficient "scaling" technique that could scarl the ConvNets balancedly from three dimensions: width, depth, and resolution. Width means the number of neurals, depth means the number of layers, resolution means the image sizes; in previous works, people have realized the relation between model performance and these three parameters and were enabled to tune them manually. However, the process of tuning these parameters costs lots of time and yields suboptimal results. In order to solve this inefficient problem, EfficientNet will scale these three dimensions with a set of fixed scaling coefficients that are determined by a small grid search on the original small model.



After scaling from a small network, EfficientNetB0 was obtained, which is the baseline for EfficientNet and B0 means the scaling limit of the model. And the architecture looks like this:

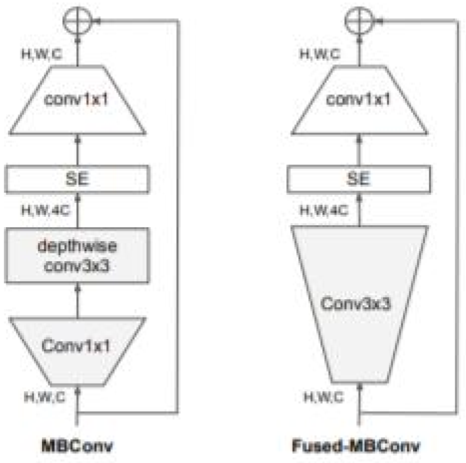


Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

And EfficientNetV2 is a different version of EfficientNetl. Instead of scaling all the stages equally, it adapted a non-uniform scaling strategy to get the architecture. And the architecture of EfficientNetV2 looks like this:

Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1

Here are the architectures of MBConv and Fused-MBConv.



The SE module in this picture is a module of attention mechanism, which attaches a weight to the feature map and gets adaptive average pooling.

Experiment Setup

First we rewrote our example code by using Mini-Batch to solve the problem of "out of memory." For batch size, we tried different common ones like 8, 16, 32 and compared them to decide which one to use for this project. Because the data size is large, considering GPU memory we didn't try batch size larger than 32. At last we used 16. Unlike other people in this competition who read all the data into CPU and then feed them into GPU, we read data to CPU using batch too which saves a lot of time for reading data and won't require a large size of CPU.

Then, we chose two kinds of splitting methods: regular train-test split and Kfold split. The regular split just splits data into one train & val group, we can use this group to test our pretrained model efficiently. The Kfold split will split the dataset into at least two train & val groups, so they allow us to do cross validation and leverage the possible overfitting issue. The split ratio for regular split is 0.2, but for Kfold, this will change depending on the number of folds we split (when fold is 5, ratio is 0.2).

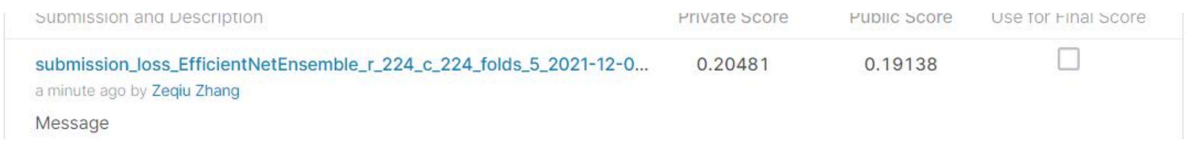
In the model creation part, we tested multiple pretrained models, excluded the top fully-connected layer and set the weight to "Imagenet." Also, since our data is easy to get a high accuracy in train set, we add a dropout and pooling layer after pretrained models. For optimizers, SDG is more likely to get an overfitting and Adadelta is slower to converge, so we decided to use Adam. And we use categorical_crossentropy as a loss function to indicate performance. Besides, we used checkpoint and early stopping callbacks to make sure that we could get the best model and result. Furthermore, instead of using Exponential Learning Rate Scheduling, we used ReduceLROnPlateau callback because the convergence speed of our training is relatively fast, it is unnecessary to change learning rate frequently. Since the class distribution of our dataset is balanced, we just use the accuracy as our validation metric. The competition metric used multi-class logarithmic loss.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij})$$

We tuned other parameters, such as dropout rate and number of epochs,in the same way as batch size. We got the balance between overfitting and underfitting, when dropout rate equals 0.5. And 10 epochs will make sure our model gains enough training time. We used 224 as image size for all models, because this is the image size of many pretrained models. Also, the same image and batch size will be easier for us to ensemble. Specifically, we didn't try nfolds bigger than 5, because a larger number of folds would take much more time to finish.

After we decided which pretrained models we wanted to focus on, we used Kfold split to divide training data into different folds, train and save these models individually. Then, we use these models to predict unlabeled test data provided by kaggle individually, and ensemble test results that come from different folds but the same model together to get a final prediction. This is our first ensemble, 'Kfold ensemble.' By using Kfold, we got 5 models trained totally separately, and they got their own validation accuracy, while they were only ensemble together to get a test submission file for the final result (upload to kaggle to get a private score) instead of ensemble together to get a new validation accuracy.

By using Kfold method, we determined 5 model results for EfficientNetV2B2, and 5 model results for EfficientNetV2B3. Since these two models have different architectures and we both got good performance in their Kfold ensembles, they will be the final ten models we decide to add to our other ensemble, 'model ensemble.' In this part, we used two for loops that the inner loop combined results of the same models, the outer loop combined results of different models. After combining and ensembling all models, we got our highest private score 0.20481.



Submission and Description	Private Score	Public Score	Use for Final Score
submission_loss_EfficientNetEnsemble_r_224_c_224_folds_5_2021-12-0... a minute ago by Zeqiu Zhang Message	0.20481	0.19138	☐

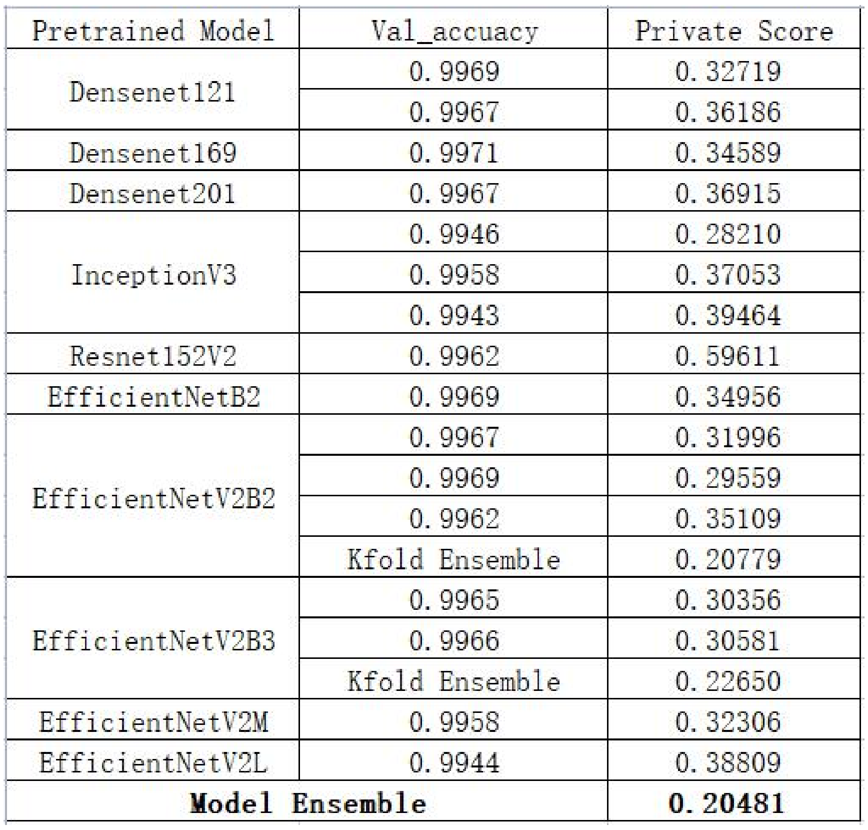
Results

First we start with a relatively smaller size model, ResNet50, and use it as the baseline to tuning parameters like batch size and learning rate, which could save some time and computation power. There is no obvious difference between batch 8 and batch 16, while they are better than batch 32. The reason why large batch size doesn't perform well might be because of the limit of GPU memory. For efficiency, we chose batch 16 for experiments afterward.

Due to the large amount of test data, we also considered using data augmentation to increase the number of our training samples. In order to create images that are similar with test data, we used random contrast, random brightness, and random crop to simulate different drivers' images. However, even if we only augment 10% of training data with small random parameters and combine them with original train data, our validation accuracy still drops significantly. Therefore, we canceled the augmentation part.

InceptionV3 showed a nice score at first but its validation accuracy is low and private score might also go down after more tries, so we discard that model thinking InceptionV3 is unstable. ResNet152V2 is a model many other participants in the competition used but didn't perform well according to the following table result. Likewise, DenseNet also didn't perform very well, the reason of which might be that it didn't have sophisticated architectures like MBConv in EfficientNet which could get desired performance with less parameters. However, we expected EfficientNetV2M and EfficientNetV2L to get higher scores because they are of larger size but they didn't perform better. The reason might be that they need larger image sizes to show their edges. However, model ensemble requires the image size to be the same, so we didn't change it for them. Besides, it is really time consuming to train that V2L model.

So we finally settled down for EfficientNetV2B2 and EfficientNetV2B3.They gave us good performance and efficient convergence speed. We got better results after using Kfold Ensemble on them individually, so we decided to ensemble the 10 models together (5 EfficientNetV2B2 folds and 5 EfficientNetV2B3 folds) to get the final result and it’s the highest score we got. This score in the competition could get the 69th position, so there should be something more sophisticated to do for improvements.



Pretrained Model	Val_accuacy	Private Score
Densenet121	0.9969	0.32719
	0.9967	0.36186
Densenet169	0.9971	0.34589
Densenet201	0.9967	0.36915
InceptionV3	0.9946	0.28210
	0.9958	0.37053
	0.9943	0.39464
Resnet152V2	0.9962	0.59611
EfficientNetB2	0.9969	0.34956
EfficientNetV2B2	0.9967	0.31996
	0.9969	0.29559
	0.9962	0.35109
	Kfold Ensemble	0.20779
EfficientNetV2B3	0.9965	0.30356
	0.9966	0.30581
	Kfold Ensemble	0.22650
EfficientNetV2M	0.9958	0.32306
EfficientNetV2L	0.9944	0.38809
Model Ensemble		**0.20481**

Summary and conclusions

In conclusion, EfficientNetV2 performed the best among all the models we tried. Kfold and ensembling different models could further improve the performance better than any single model. This project also told us that it is necessary to read data into a CPU using batches could save time for loading and save the memory of a CPU. Data augmentation in our case didn’t help to increase the performance, which is also a surprise. And the error analysis also might help us to improve the model, if finding some similarities between images we didn’t predict correctly. And For future improvement, we should consider some more sophisticated augmentation like Mixup or CutMix which we didn’t use could help to raise the score. Another improvement that might have an effect will be using larger image sizes for running

EfficientNetV2M and EfficientNetV2L. If they show better performance, ensemble them together could get a higher score.

Besides, in this dataset, one driver could have a lot of images and these images are the image of the driver in a different timelines, which could form a complete gif. If that information could feed to the model, maybe people can get better results. That model may need a recurrent architecture to capture the sequence information because the previous input would have an influence on the output.

References

https://www.kaggle.com/c/state-farm-distracted-driver-detection/data Kaggle: State Farm Distracted Driver Detection

Tan, M., & Le, Q. (2019, May). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning* (pp. 6105-6114). PMLR.

Tan, M., & Le, Q. V. (2021). Efficientnetv2: Smaller models and faster training. *arXiv preprint arXiv:2104.00298*.

State Farm distracted driver detection. Kaggle. (n.d.). Retrieved December 6, 2021, from https://www.kaggle.com/c/state-farm-distracted-driver-detection/discussion/22631.

State Farm distracted driver detection. Kaggle. (n.d.). Retrieved December 6, 2021, from https://www.kaggle.com/c/state-farm-distracted-driver-detection/discussion/22906.

State Farm distracted driver detection. Kaggle. (n.d.). Retrieved December 6, 2021, from https://www.kaggle.com/jiaodong/vgg-16-pretrained-loss-0-23800.