Zhuohan Yu ML2 Final Report

**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.

**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

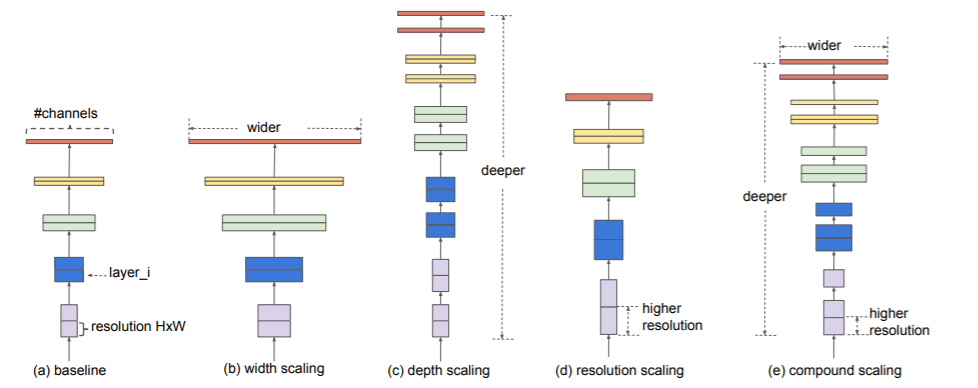
c9: talking to passenger

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.

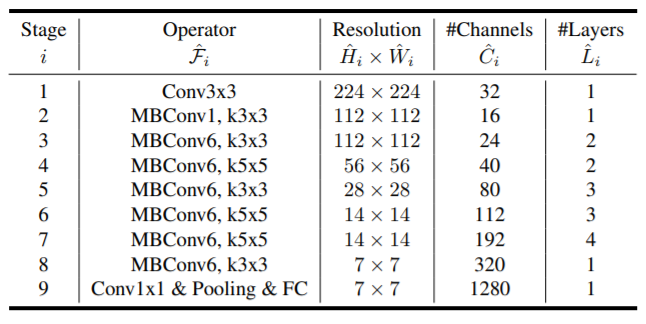
**Model Description**

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:

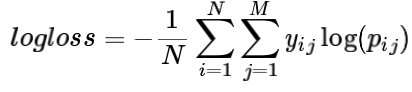


**Experiment Setup**

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.



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.

**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.

What I Learned:

1. The way of using pretrained model and how to modify them.
2. The way of read data by using batches.
3. The way of adding callbacks and learning rate scheduling.
4. The way of doing data augmentation.
5. The way of doing ensemble and cross validation.

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.

My Work:

**Role:**

Create ideas (and assign specific works for teammate)

Making decisions

Trying most of the pretrained Models

**Code:**

Rough draft of test\_model

Defined the model

Rewrite batch and read\_data

Data augmentation

Combine different version of codes

Fixing bugs (i.e. the bug of KFold)

Callbacks and learning rate scheduling

**Report:**

Write experiment, introduction, dataset, and EfficientNet, refernce part

**Powerpoint:**

Write experiment setup, introduction, dataset, and EfficientNet part in report

**Percentage of copied code:**

16.279%