DSC 672: Predictive Analytics Capstone

Milestone 4 - Intermediate Results Part II

Stanford Car Dataset: Vehicle Recognition Project

Alexandre Girault (Team Manager)
Chris Shaffer
Sean Sungil Kim
Ryoh Shinohara
Yanxi Cai

Spring 2019

Introduction

In working towards Milestone 4, the team attempted to produce more advanced iterations of image classification models to classify car make and model after dedicating extensive amount of time setting up the necessary environment for analysis. The team is split up into two main groups, with two members working on CNNs and three members working on non-NN models. In working with CNNs, the CNN group utilized both custom CNN structures and pre-trained CNNs (ResNet34 and Xception). For the non-NN group, members used some of the most successful algorithms found in Caruna and Niculescu-Mizil's paper discussed in Paper Discussion 2 (SVM, Random Forest, and Boosted Trees).

Due to the size of the dataset, it was imperative that we find a work environment where we could perform model building in an adequate time frame. Memory was a major issue in both groups, requiring most members to utilize AWS services. While utilizing EC2 instances and Sagemaker (AWS product) provided significant boost in performance, setting up the instances took a substantial amount of time for most users due to the lack of experience in using cloud services. This caused delays in generating our initial models especially for the non-NN members. Furthermore, as building models and training the models continued, costs of using AWS services became another limitation.

Boosted Trees and Random Forest

Boosted trees and random forests were one of the most successful algorithms in the paper from Paper Discussion 2, with both algorithms scoring well in image classification tasks. Therefore, we chose these two decision tree-based ensemble learners to compare performances with the CNNs. For boosted trees, we chose XGBoost due to its speed and its successes in image classification competitions prior to the arrival of CNNs.

Though XGBoost appeared to be a viable option in this image classification task in milestone 3, this was not the case after running multiple runs of hyperparameter tuning for this algorithm. Although we used a relatively large EC2 instance (m5.2xlarge instance: vCPU: 8, memory: 32 GiB, dedicated EBS bandwidth: up to 3,500 Mbps, network performance: up to 10 Gbps), frequent memory issues and weak internet connection prevented any hyperparameter tuning iterations to run to completion after two weeks of attempting. Therefore, we focused on using random forests as our decision tree based ensemble learners.

While random forests also had memory issues, we were able to run over 60 hyperparameter combinations. Using sklearn's RandomizedSearchCV() method, we randomly chose different hyperparameter combinations using the hyperparameters shown in Table x and performed 5-fold cross-validations. Although it would have been ideal to use the argument for the number of iterations of randomized searches to be done in the RandomizedSearchCV() method, we were unable to do this due to the excessive memory

requirement for the method. Therefore, we ran one iteration of RandomizedSearchCV() using different random state each round. Through this, we were able to narrow the range of hyperparameters based on the mean error and common hyperparameters found among the top performers across 5-folds to further perform a grid search in the upcoming week. For the purposes of presenting a result for this week, we chose to fit two random forest model based on the best performing models from the randomized grid search. The hyperparameters used were: model 1: n_estimators = 1200, min_samples_split = 5, min_samples_leaf = 4, max_features = 'sqrt', max_depth = 60, and bootstrap = False; and model 2: n_estimators = 600, min_samples_split = 10, min_samples_leaf = 4, max_features = 'sqrt', max_depth = 50, bootstrap = False. Model 1 yielded a mean accuracy of 0.061, weighted F1 score of 0.052, and the confusion matrix as seen in *Figure 3*. Model 2 yielded a mean accuracy of 0.054, weighted F1 score of 0.043, and the confusion matrix as seen in *Figure 4*

Parameters	RandomizedSearchCV()	GridSearchCV
n_estimators	200, 400, 600, 800, 1000, 1200, 1400	600 - 1400
max_features	'sqrt'	‘sqrt’
max_depth	10, 20, 30, 40, 50, 60, 70, 80, 90, 100	40 - 80
min_samples_split	2, 5, 10	2 - 10
min_samples_leaf	1, 2, 4	2 - 5
bootstrap	True, False	False

Figure 1: Random forest hyperparameter list

	Model Parameters	Mean Accuracy	F1 Score (weighted)
1	n_estimators = 1200 min_samples_split = 5 min_samples_leaf = 4 max_features = 'sqrt’ max_depth = 60 bootstrap = False	0.061	0.052
2	n_estimators = 600 min_samples_split = 10 min_samples_leaf = 4 max_features = 'sqrt' max_depth = 50 bootstrap = False	0.054	0.043

Figure 2: Random forest hyperparameter list

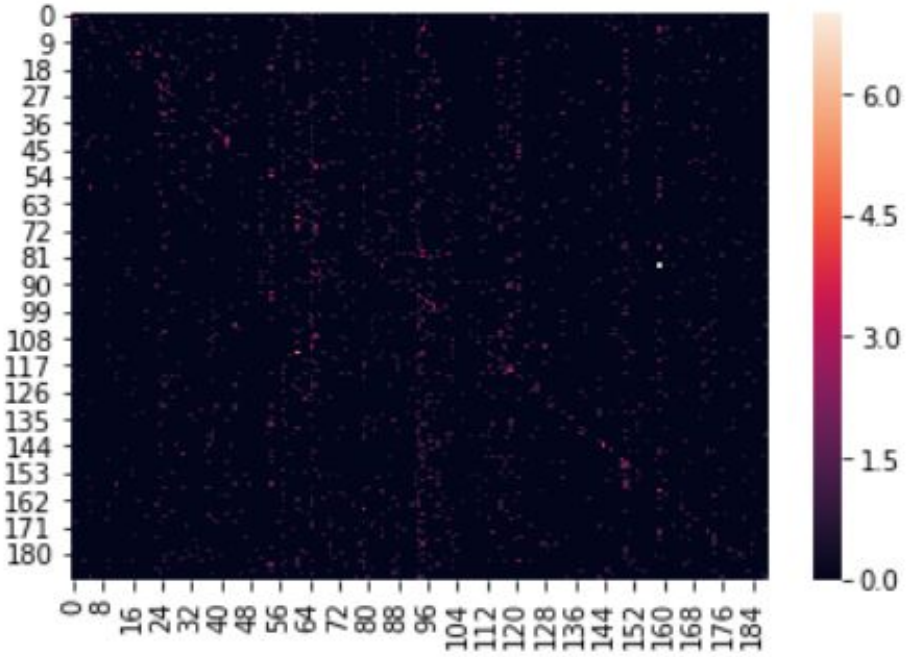


Figure 3: Confusion matrix of random forest model on test set 'Parameters 1'

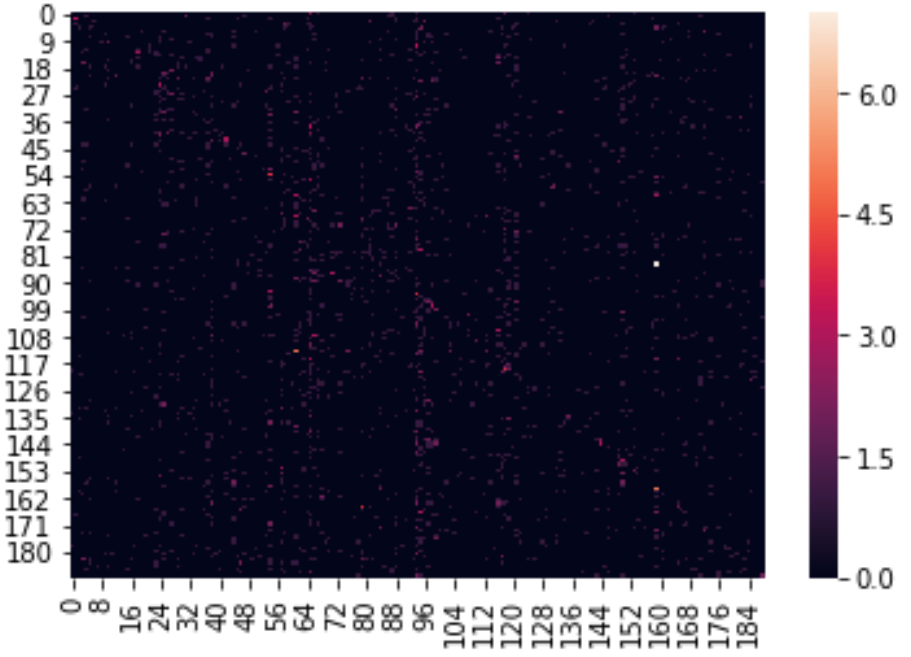


Figure 4: Confusion matrix of random forest model on test set 'Parameters 2'

Support Vector Machines

Support vector machine is supervised learning and was popular in image classification before Deep learning. In this project, we used it as one of our no Deep Learning algorithms. Last time we built initial SVM models using Canny edge detection and grayscale. The performance was inferior with accuracy below 0.01. Since then we have explored several different feature extraction methods to improve the model and got significant progress. Before tuning the parameters, we used the default SVM model, which means C=1, gamma='auto' and kernel='rbf', to fit with different feature extraction algorithms to find the best combination. The testing set performance is shown below, where 'edge' means Canny edge detection, 'Gaussian' means adaptive gaussian thresholding and 'Mean' means adaptive mean thresholding.

Algorithms	Accuracy	Recall	Precision	F-score	Time
Grayscale+edge	0.0050	0.0005	0.0001	0.0000	17072
Grayscale+edge+PCA	0.0388	0.0081	0.0066	0.0068	18.72
Rgb+PCA	0.0388	0.0388	0.0823	0.0409	366.52
GrayScale+PCA	0.0463	0.0463	0.0772	0.0484	329.38
GrayScale+Gaussian+PCA	0.0252	0.0252	0.0357	0.0151	972.00
GrayScale+Mean+PCA	0.0343	0.0343	0.0402	0.0199	920.99

Figure 5 - Testing set performances for SVM

According to the table, PCA on grayscaled images had the best performance, with the highest accuracy, precision, and F score. The complexity (time) is also acceptable. During exploring the model building, we found an abnormal pattern that the smaller image size, the higher the accuracy and vice versa. For example, for 'grayscaled+PCA', when the image size was 100x51, the accuracy was 0.0564, but it would become 0.0645 and 0.0276 as the size of images changed to 50x25 and 300x153 respectively. It violated the common sense because downsizing can cause significant information loss in image data, so when the size is larger, we would expect an improvement in performance but actually, it was the opposite. However, we still can't downsize the images too much even though it produces better performance because it may cause overfitting as well. To balance this tradeoff, we estimated what size to use by visualizing the quality of images with different sizes.

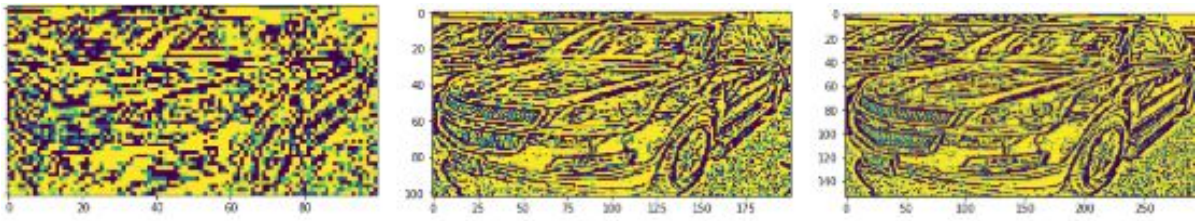


Figure 6 - Adaptive Gaussian Thresholding images with different size

For instance, the Gaussian adaptive thresholding images above from left to right are 100x51, 200x102 and 300x103. We can see that images with size 200x102 are much clearer than those with 100x51, while images with size 300x153 didn't have significant improvement compared to those with size 200x102,and considering larger-size images resulted in lower accuracy and higher computation load, we decided to downsize the images to 200x102. Finally, we tuned the parameter of SVM and it showed when C=10, gamma=0.0001, kernel='rbf', the model had the best performance shown below.

Tuned parameters	Accuracy	Recall	Precision	F-score	Time
C=10, gamma=0.0001,kernel='rbf'	0.0655	0.0655	0.0777	0.0577	330.22

Figure 7 - The best performance of testing set after parameter tuning

Convolutional Neural Networks

Convolutional neural networks (CNNs) are an extension of neural nets adapted for image classification. The model is composed of layers of filters and compressions in sequence. Filter weights are adjusted like those in the neural net for similar purposes. With an appropriate amount of training samples and adequately-tuned parameters, a CNN will automate the feature selection process and accurately classify the input.

DATA AUGMENTATION

Image recognition / classification problems often suffer from a lack of training data. This is the curse of dimensionality, since each image contains a huge amount of data so a very large number of samples is required to best explain the feature space. In lieu of collecting more samples manually, the models are trained with synthetic images. These images are slightly distorted (mirrored, rotated, and sheared to some extent) in order to help the model better generalize the input space. Consequently, this helps combat overfitting, which is useful for models that benefit from long training sessions. The image modification step was only applied to the training images. Test images were left unmodified in order for models to be evaluated with actual image data.

FILTER LAYERS

Convolution layers contain trainable kernels (sets of weights) which transform patches of the input image. These are essentially different kinds of image filters that the model learns to alter and tune during training in order to obtain the most accurate results. A well-trained model contains an array of filters that effectively discriminate between different classes of images. In the Keras package, convolutional layers have several user-defined inputs such as stride length and kernel dimensions. Altering these two parameters in particular can influence model learning and performance, but were not explored in depth here. The default kernel size of 3x3 was used for each models' filter layers. Max-pooling is used after each block of convolution layers. The pooling process summarizes a window of pixels into a single output value. In the case of max-pooling, the window will be condensed into the highest value. This helps maintain the image's contrast between filter blocks. During development, little difference was seen between this and the average-pooling approach.

NEURAL NETWORK

Aggressive dropout rates are utilized to help prevent overfitting. Higher levels of dropouts allow for more epochs to be used during training, thus giving the model more time to discover useful weights. A rate of 0.65 is used in these models. In order to preserve as much of the original information as possible, ReLU activations are used between layers. This type of activation is ideal for preserving the 'shape' of the data by allowing for one-to-one input-output mapping. The final layer applies a Softmax transformation to the final summations. This transforms the output into a set of likelihood values, the highest of which belonging to the best guess.

EVALUATION

Two metrics were monitored during training- accuracy and top-5 accuracy. The top-5 accuracy metric gives another level of feedback during training, since accuracy alone does not give credit to close guesses. The average loss during each epoch can also be monitored for evaluation. However, accuracy is a much more tangible, customer-relevant feature for the proposed vehicle identification application.

Make & Model CNN

Relatively large CNN models were explored for this problem. The following architecture was inspired by the VGG16 CNN model, though it is more compact due to computational limitations. Unlike the previous two CNN architectures, this model stacks convolutions before each pooling layer. This allows for the model to consider a much larger space of possible weights and filter values to explore. In order to accommodate a more

difficult problem, the model was trained for 300epochs, at which point the validation accuracy (blue) seems to approach and level off at 55%. The top-5 validation accuracy (black) approaches 80%.

64	2D Convolution	3 x 3
64	2D Convolution	3 x 3
1	MaxPool	2 x 2
128	2D Convolution	3 x 3
128	2D Convolution	3 x 3
1	MaxPool	2 x 2
256	2D Convolution	3 x 3
256	2D Convolution	3 x 3
256	2D Convolution	3 x 3
1	MaxPool	2 x 2
512	2D Convolution	3 x 3
512	2D Convolution	3 x 3
512	2D Convolution	3 x 3
1	MaxPool	2 x 2
2048	Flatten	2048
2048	Dense	ReLU
2048	Dropout	0.65
512	Dense	ReLU
512	Dropout	0.65
166	Dense	166 - Sofmax

Figure 8 - Custom CNN Model Structure

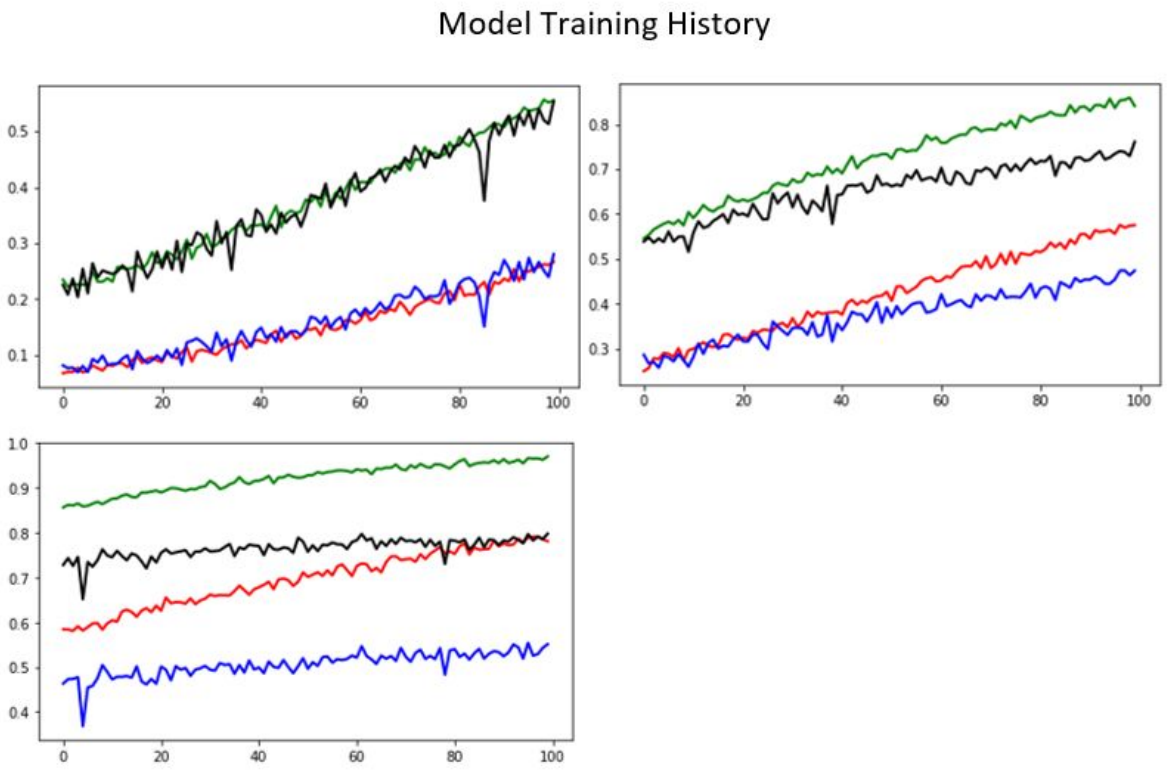


Figure 9 - Custom CNN Training History (300 Epochs).

ResNet34

State-of-the-art ResNet34 CNN model was further developed by going through more training. Excluding the top layer training phase, this model was trained for a total of 100 epochs, utilizing the one-cycle-policy introduced in Milestone 3. The loss vs. learning rate graphs used in determining the optimal learning rate for the one-cycle-policy is demonstrated below. Both the top layer training and unfrozen model training used the one-cycle policy.

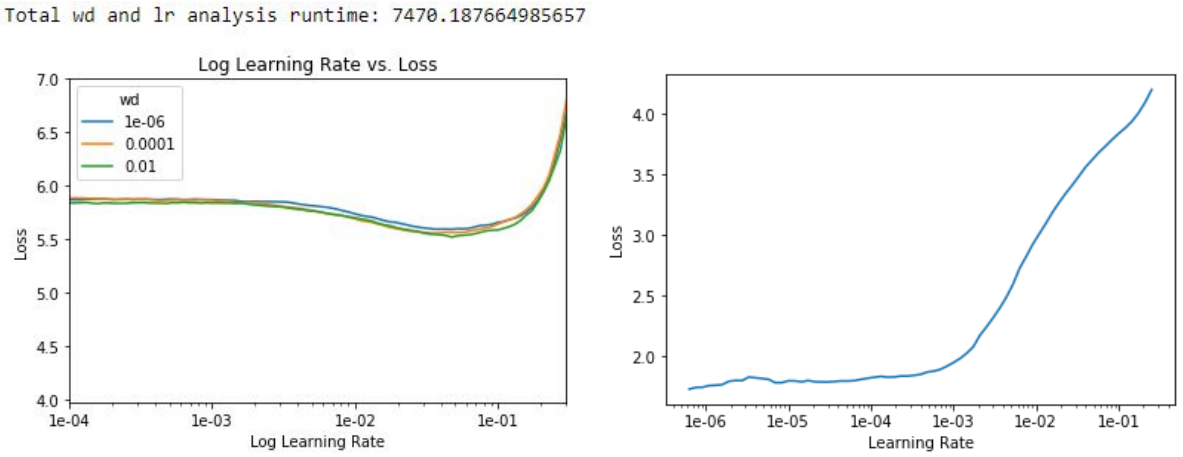


Figure 10 - Log Learning Rate vs Loss Graph Before and After Training the FC Layers for the ResNet34 Model.

Some augmentations and transformations were applied on the image data for the ResNet34 model. This ultimately helps in preventing overfitting, since this process creates a bit of noise within the images. The parameters used for the augmentations are do_flip = True, flip_vert, False, max_rotate = 90. A preview of some of the images are illustrated below.

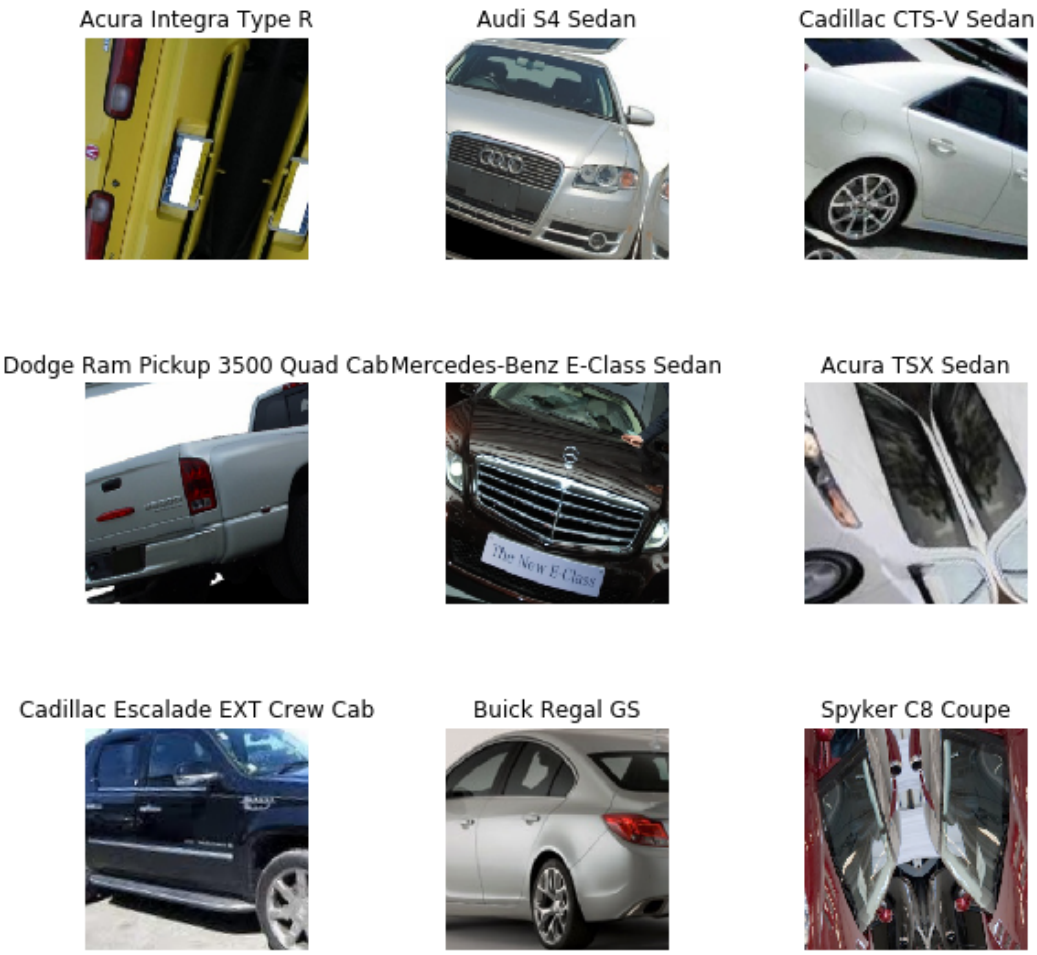


Figure 11 - Preview of the Training Images with Augmentation/Transformation.

After 100 epochs of training, the ResNet34 model was able to obtain 0.7801 training loss and 0.7311 testing loss. Compared to the previous loss vs. iteration graphs that are attached under the Miscellaneous Graphs section at the end of this paper, the gap between the training loss and the testing loss was decreasing. Moreover, both of the training and testing loss lines were decreasing over the training iterations, as illustrated in the table and graph below. The overall decrease in training and testing loss throughout the entire 100 epochs can be found under the Miscellaneous Graphs section as well.

epoch	train_loss	valid_loss	accuracy	time
0	0.828971	0.757601	0.789338	47:57
1	0.823475	0.759355	0.789338	47:17
2	0.834208	0.758517	0.786248	47:51
3	0.851897	0.753168	0.785733	47:02
4	0.821245	0.749253	0.790883	47:52
5	0.804125	0.741682	0.790111	46:54
6	0.818743	0.739885	0.791141	48:05
7	0.801630	0.729439	0.791398	47:00
8	0.798832	0.732448	0.792944	47:57
9	0.780111	0.731119	0.793974	46:53

Figure 12 - ResNet34 Final (10th) Model Performance Table.

```
The precision score is 0.8100481150660742
The recall score is 0.7939737316507854
The fscore score is 0.7940082428478917
```

Figure X - ResNet34 Model Performance Scores.

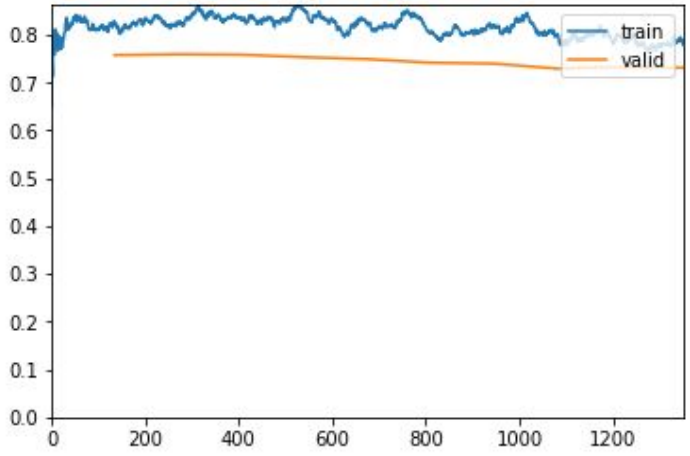


Figure 13 - ResNet34 Final (10th) Loss vs Iterations Graph.

Large gap between the training and the testing loss could be interpreted as the model not being able to perform similarly on training and testing set. Small gap between the training and the testing loss could be interpreted as the model's ability to perform similarly regardless of training/testing set it uses. In ResNet34's case, the training loss was much higher than the testing loss during the beginning stages of the learning phase. This meant that the model had room for improvement by learning more from the training data. After 100 epochs of fitting, the training and testing loss were both improved (decreased). In addition, the gap between the two loss lines also narrowed down. The training of ResNet34 was successful, since it was able to bring the training and testing loss down and increased the model accuracy.

The cross and/or the divergence of the training and testing loss lines indicates overfitting. This did not occur during the 100 epochs of model fitting. There is potential room for improvement in terms of loss and accuracy scores. However, since 10 epochs take about 8 hours and there was less than 0.005 increase in model validation accuracy during the last 10 epochs, the training process was stopped after 100 epochs to save computation and time costs.

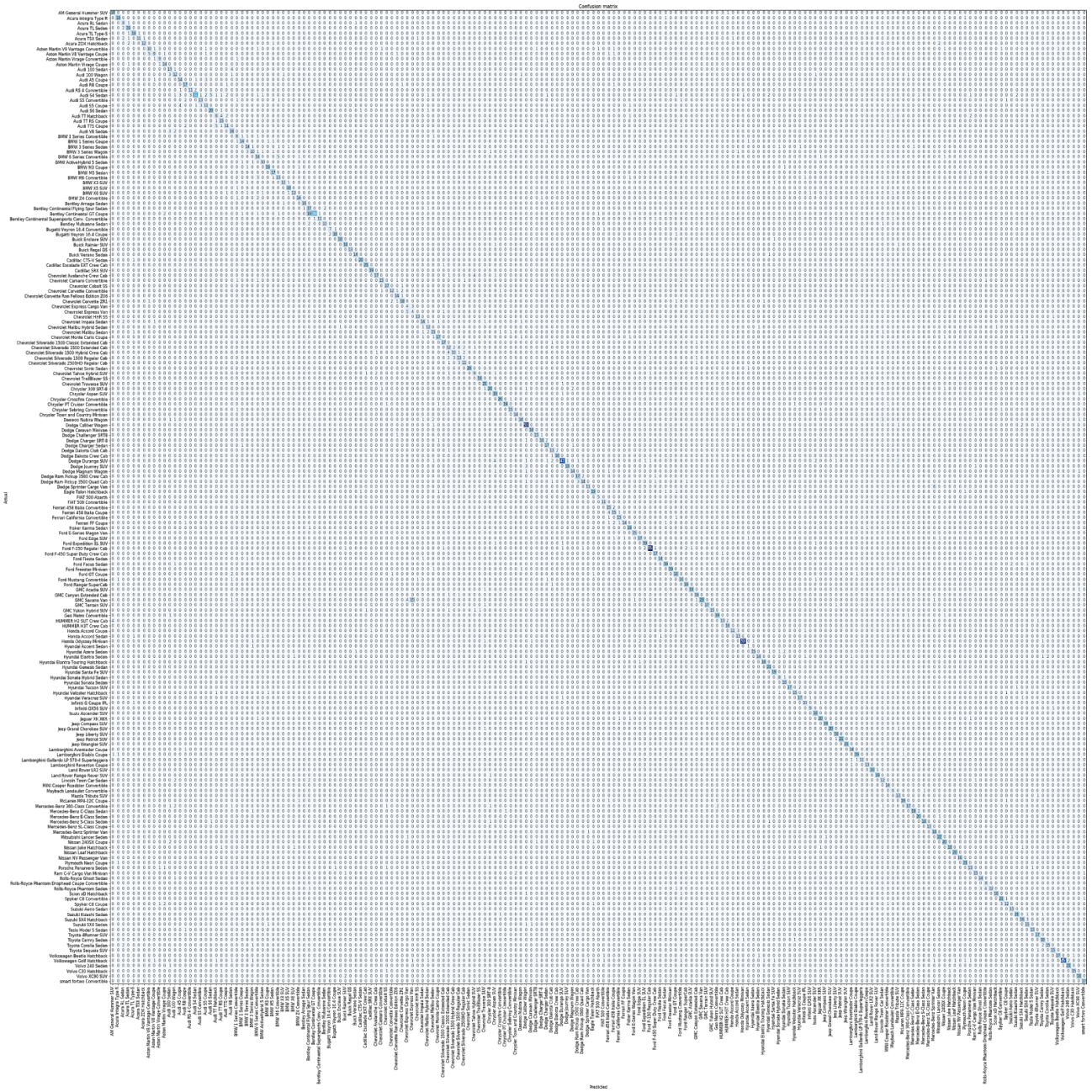


Figure 14 - Actual vs. Predicted Confusion Matrix for the ResNet34 Model.

```
[('Bentley Continental GT Coupe', 'Bentley Continental Flying Spur Sedan', 19),
 ('GMC Savana Van', 'Chevrolet Express Van', 15),
 ('Dodge Sprinter Cargo Van', 'Mercedes-Benz Sprinter Van', 6)]
```

Figure 15 - Most Confused Images for the ResNet34 Model.

Although the confusion matrix above is not very readable, it is clear that the model performance is high based on the diagonal actual vs. predicted line. There are two spots that are not a part of this diagonal line. These two spots are the first two lines of the most confused images summary above. The heatmaps below portray images with high loss scores and highlight areas that the ResNet34 model believes that are important. This could be used to analyze why the model is classifying incorrectly more in depth.

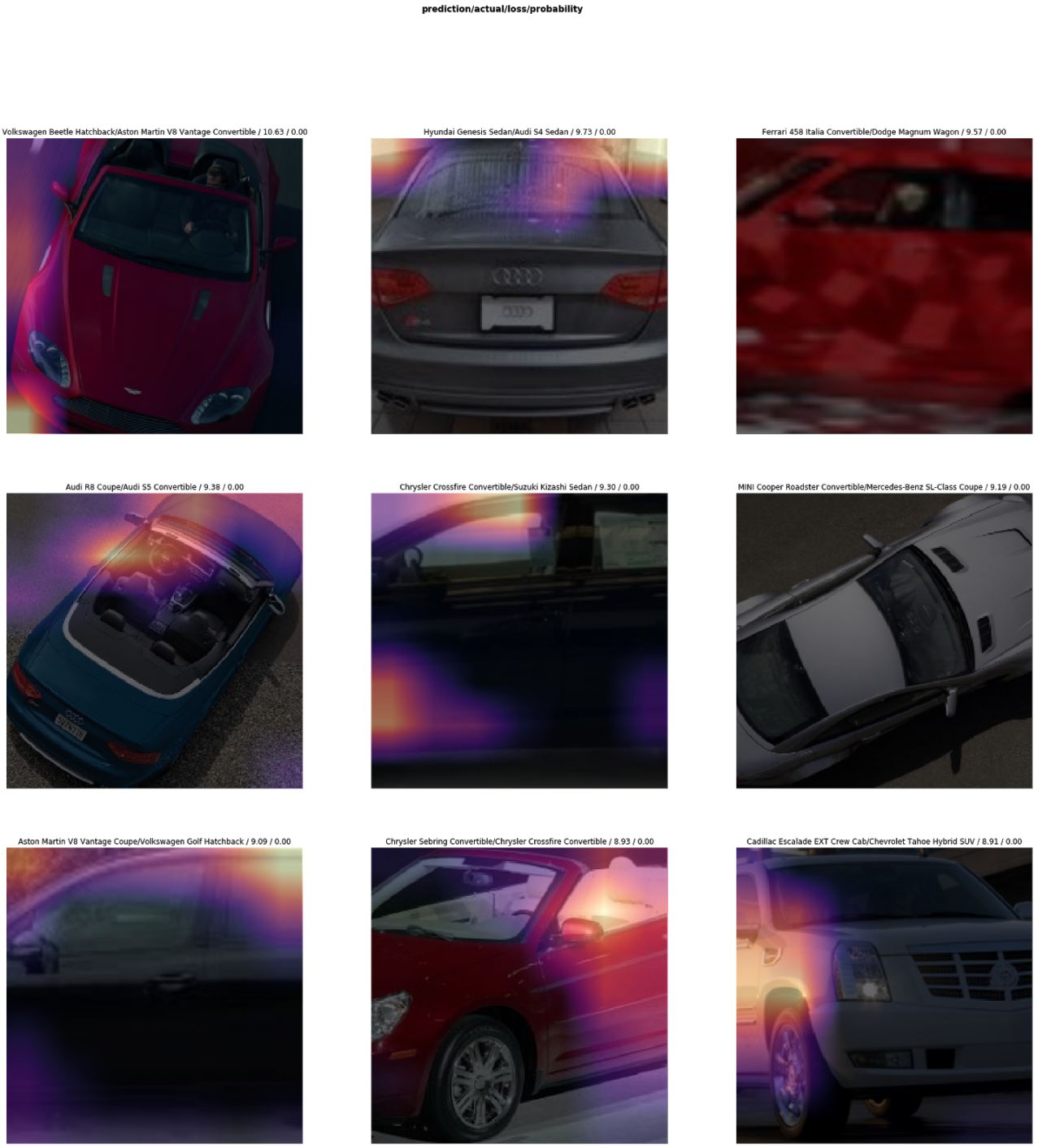


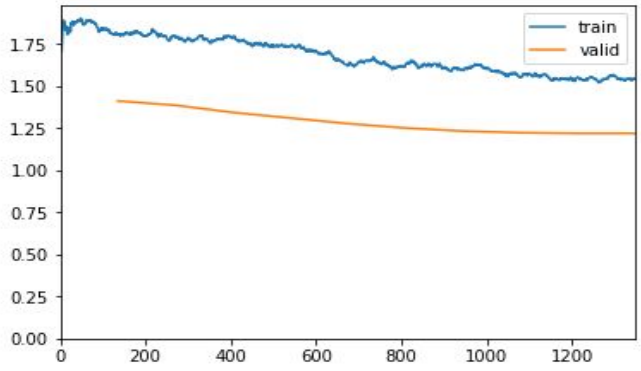
Figure 16 - Areas Focused for the Most Confused Images by the ResNet34 Model.

Conclusion

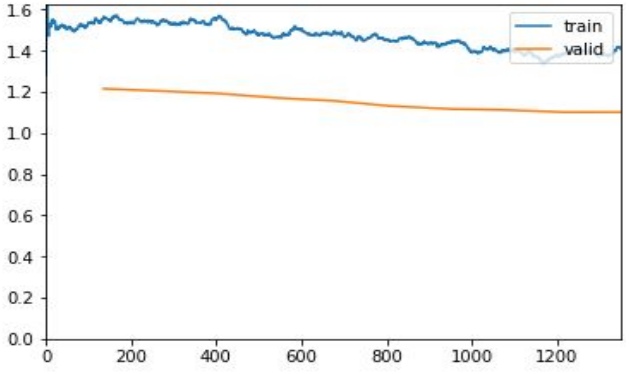
To conclude, although computationally very heavy, the CNN team is achieving a high accuracy in car make/model recognition. These CNN methods showed to be much more accurate than non-Deep Learning methods on this dataset with an important accuracy gap in comparison. In order to work towards Milestone 5, some model training will be achieved to see until where the models' accuracy can go. Lastly, more than 50 additional images will be added to the testing set in order to evaluate the possibility of prediction on more "free-living conditions" data. If the obtained results are successful and the team reach a comparable accuracy using CNN methods, then the team will be able to achieve its business objective initially set.

Miscellaneous Graphs

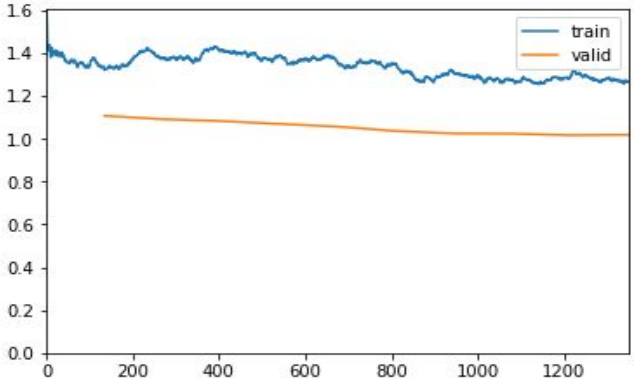
epoch	train_loss	valid_loss	accuracy	time
0	1.808390	1.410663	0.632243	49:07
1	1.790068	1.385608	0.638939	50:36
2	1.781615	1.342545	0.654133	48:59
3	1.734212	1.309546	0.659284	49:18
4	1.619155	1.276190	0.664692	48:52
5	1.619910	1.250240	0.673191	49:28
6	1.594307	1.232061	0.673963	49:26
7	1.550314	1.222225	0.679372	49:52
8	1.552493	1.218435	0.675766	49:52
9	1.542795	1.218609	0.678342	49:56



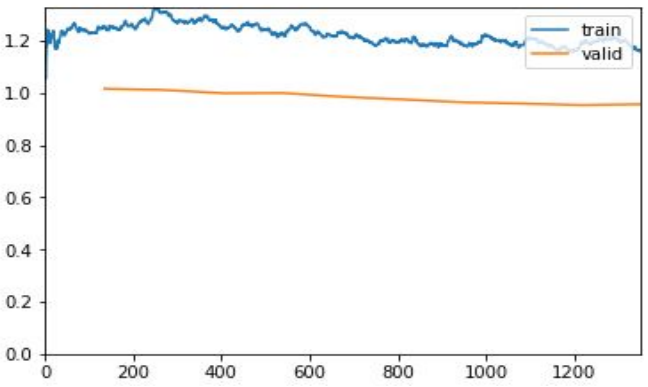
epoch	train_loss	valid_loss	accuracy	time
0	1.557081	1.216002	0.675251	14:58
1	1.545179	1.204483	0.681689	14:48
2	1.564081	1.192634	0.684522	14:51
3	1.465772	1.171620	0.688643	26:22
4	1.483997	1.156955	0.693021	47:19
5	1.449640	1.131565	0.704095	48:21
6	1.440592	1.117790	0.703580	46:59
7	1.400102	1.112542	0.704095	49:10
8	1.384001	1.101945	0.708215	47:01
9	1.410389	1.102118	0.707443	48:00



epoch	train_loss	valid_loss	accuracy	time
0	1.321524	1.107562	0.706155	37:22
1	1.378396	1.091762	0.708215	46:38
2	1.416011	1.083155	0.712078	48:37
3	1.387095	1.069836	0.714654	47:32
4	1.357621	1.057059	0.720577	47:41
5	1.350469	1.036730	0.725728	48:44
6	1.307412	1.025207	0.729591	46:43
7	1.274621	1.024214	0.730363	48:47
8	1.290659	1.016932	0.732423	46:05
9	1.266117	1.018915	0.731393	48:49

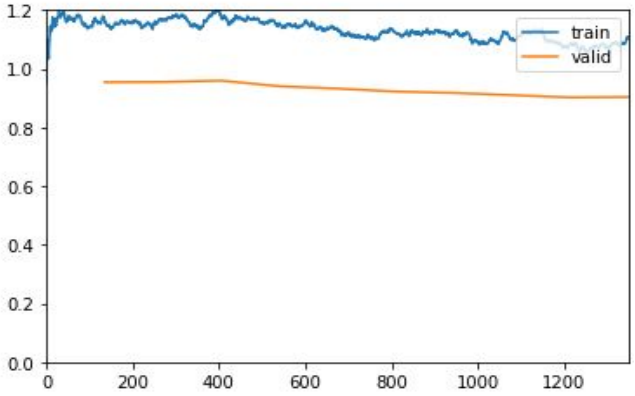


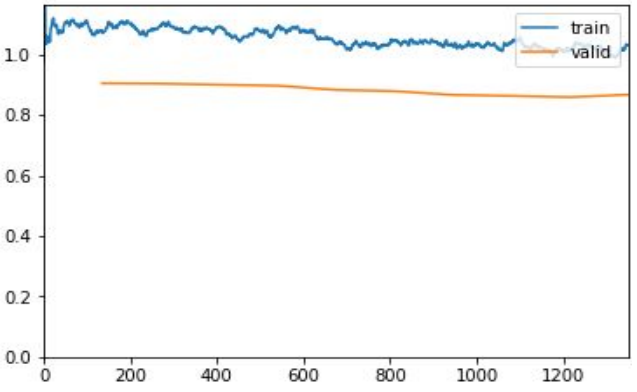
epoch	train_loss	valid_loss	accuracy	time
0	1.250747	1.015860	0.732938	47:46
1	1.307564	1.010971	0.730621	47:41
2	1.249875	0.998471	0.736029	47:18
3	1.230348	0.999026	0.734484	48:15
4	1.232139	0.984812	0.734226	46:46
5	1.207032	0.974829	0.742982	48:10
6	1.189793	0.963933	0.742725	46:56
7	1.181206	0.958936	0.743497	47:41
8	1.183900	0.953023	0.745815	47:05
9	1.159541	0.956321	0.744785	47:41



epoch	train_loss	valid_loss	accuracy	time
0	1.168369	0.954173	0.746073	47:08
1	1.166362	0.954944	0.741952	47:26
2	1.184143	0.959483	0.745043	47:08
3	1.153834	0.940191	0.748648	47:51
4	1.137994	0.932312	0.747360	47:12
5	1.123653	0.922264	0.751481	47:53
6	1.118429	0.917528	0.754829	47:37
7	1.097875	0.909929	0.753799	47:31
8	1.086187	0.902132	0.755086	48:07
9	1.109474	0.903756	0.754314	46:52

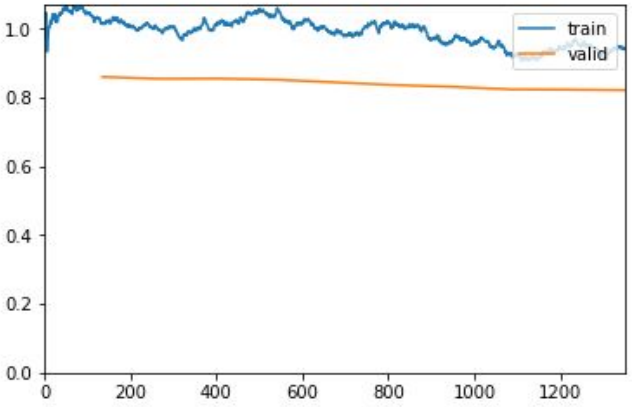
epoch	train_loss	valid_loss	accuracy	time
0	1.074703	0.904523	0.757404	14:52
1	1.097129	0.903359	0.757404	14:50
2	1.092081	0.899894	0.756374	14:58
3	1.088593	0.896544	0.756631	24:34
4	1.035675	0.882772	0.761267	47:48
5	1.048380	0.878450	0.761267	47:56
6	1.029200	0.866074	0.764873	49:15
7	1.038603	0.862865	0.762297	47:42
8	1.012528	0.858699	0.764873	47:58
9	1.029502	0.866741	0.763842	49:20

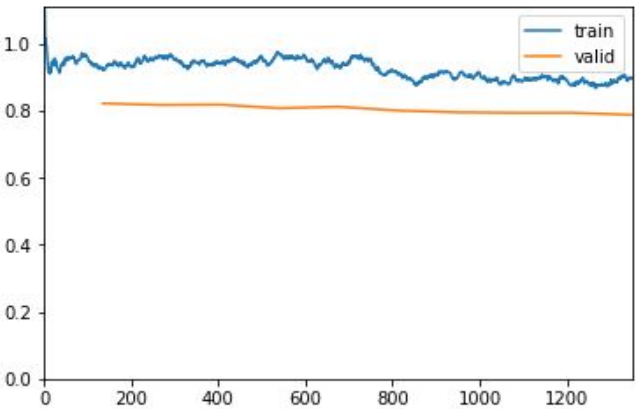




epoch	train_loss	valid_loss	accuracy	time
0	1.014698	0.859986	0.764873	43:32
1	0.995521	0.854635	0.768736	48:41
2	1.018612	0.855251	0.766675	46:24
3	1.044091	0.851986	0.768478	48:38
4	0.986598	0.844310	0.773114	46:36
5	1.006567	0.836668	0.772341	48:20
6	0.961997	0.831541	0.773371	46:45
7	0.927437	0.823986	0.776719	48:14
8	0.946067	0.823031	0.777749	47:13
9	0.940417	0.821491	0.776204	48:06

epoch	train_loss	valid_loss	accuracy	time
0	0.925195	0.822583	0.777492	47:00
1	0.960630	0.818020	0.777749	48:39
2	0.940481	0.818945	0.778779	46:50
3	0.971666	0.808305	0.777234	48:22
4	0.932030	0.812829	0.778264	46:55
5	0.917777	0.801429	0.779294	48:02
6	0.895212	0.796258	0.783672	47:22
7	0.905308	0.794433	0.785733	47:41
8	0.881446	0.794623	0.783157	48:14
9	0.898755	0.788267	0.786505	47:41





epoch	train_loss	valid_loss	accuracy	time
0	0.907949	0.790865	0.783672	42:28
1	0.899129	0.787927	0.781097	47:36
2	0.896608	0.789776	0.780325	48:17
3	0.888053	0.784256	0.786763	46:54
4	0.900395	0.773797	0.784960	48:36
5	0.851556	0.765870	0.787020	46:53
6	0.869491	0.767473	0.788308	48:45
7	0.841771	0.761102	0.789853	46:46
8	0.824650	0.758777	0.788566	48:22
9	0.825978	0.757898	0.789596	47:00

epoch	train_loss	valid_loss	accuracy	time
0	0.828971	0.757601	0.789338	47:57
1	0.823475	0.759355	0.789338	47:17
2	0.834208	0.758517	0.786248	47:51
3	0.851897	0.753168	0.785733	47:02
4	0.821245	0.749253	0.790883	47:52
5	0.804125	0.741682	0.790111	46:54
6	0.818743	0.739885	0.791141	48:05
7	0.801630	0.729439	0.791398	47:00
8	0.798832	0.732448	0.792944	47:57
9	0.780111	0.731119	0.793974	46:53

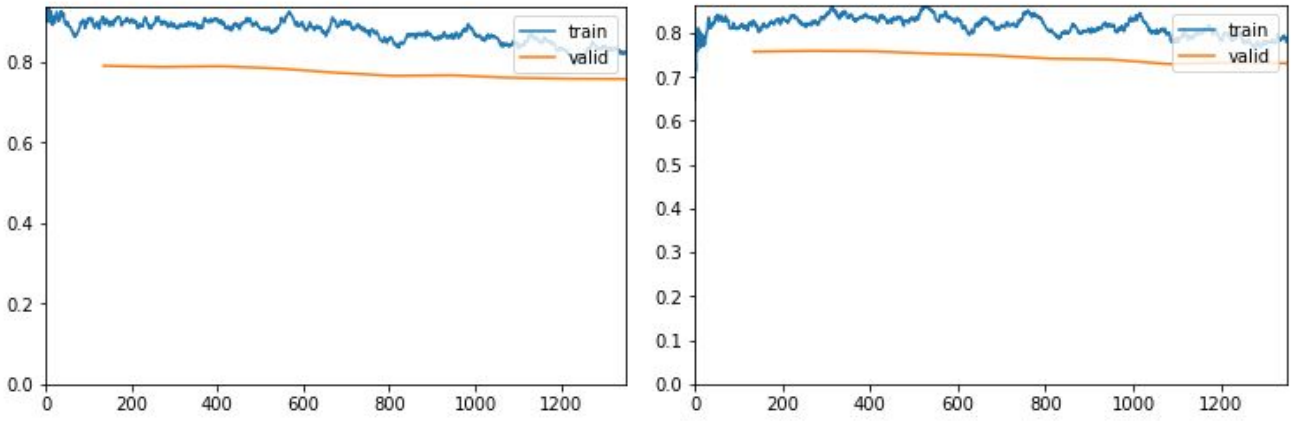


Figure 17 - ResNet34 Training Outputs (10 phases of training, 10 epochs each, excluding the top layer training).