CSE598 Introduction to Deep Learning

Final Project

Cheng-Ming Hsu chsu81@asu.edu

Ninad Bharat Gund | ngund@asu.edu

General project idea:

In this project, we aim to tackle the classic problem of Image Classification for the task of Identifying Dog breeds from the image of the dog. For this purpose, we aim to use Deep learning and train a convolutional neural network which can identify the correct dog breeds with high accuracy. We will evaluate how some of the well known CNN architectures perform this task. We will use Transfer Learning to redirect the pretrained networks by modifying their final layers, and retrain the models for this task.

GitHub repo for all code, notebooks, scripts used:

- https://github.com/ninadgund/CSE598_IntroToDL_Fall21_FinalProject.git

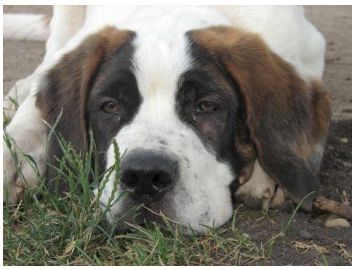
The problem:

In this task, our job is to train a model that inputs a picture of a dog and outputs the breed of this dog.

Input: An image file

Output: string(breed)

e.g.



```
['saint_bernard']
```

Dataset:

Given a training set and a test set of images of dogs. Each image has a filename that is its unique id. The dataset comprises 120 breeds of dogs.

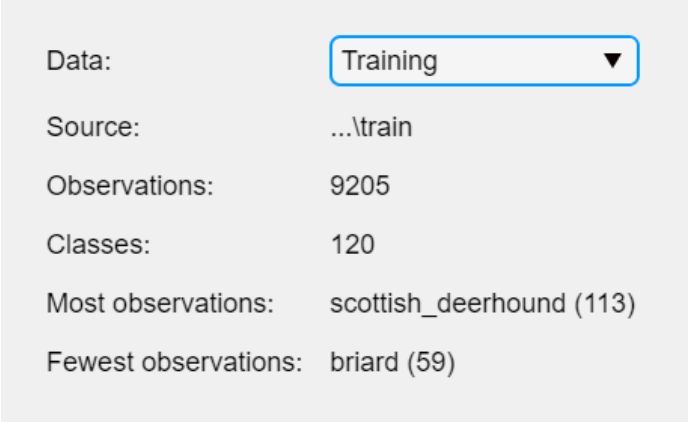
Total number of training data: 10222

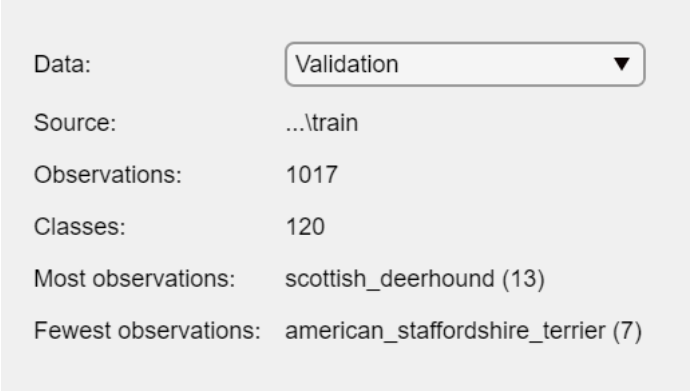
Number of labels: 120

Source link: https://www.kaggle.com/c/dog-breed-identification/data

The dataset only provides the real labels for training data, and the testing data is unlabeled. The problem statement expects a submission on kaggle for test data accuracy. But since we wanted to generate the accuracy on our own, we generated all the training / testing / validation data from the training data itself, ignoring the test data available.

We selected 10% of the training data at random and generated a validation dataset.





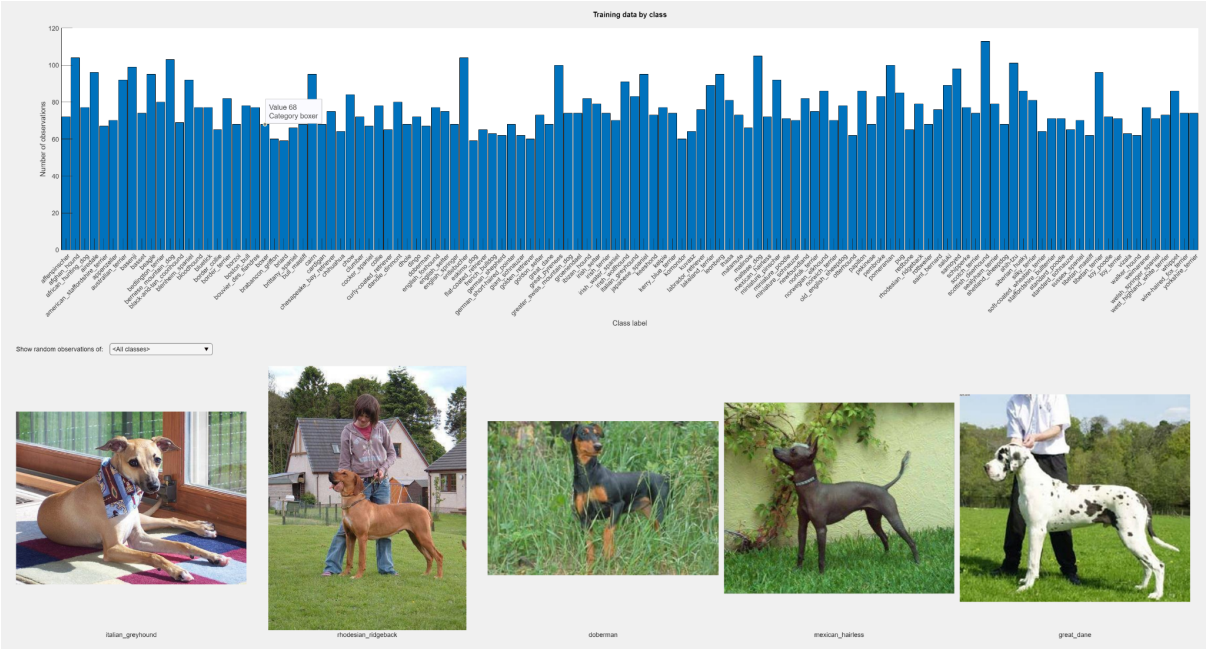
The training data is relatively uniform over 120 labels, so we can use the dataset as it is. But we did modify the structure of the dataset to make it easier for us to feed it to the models.

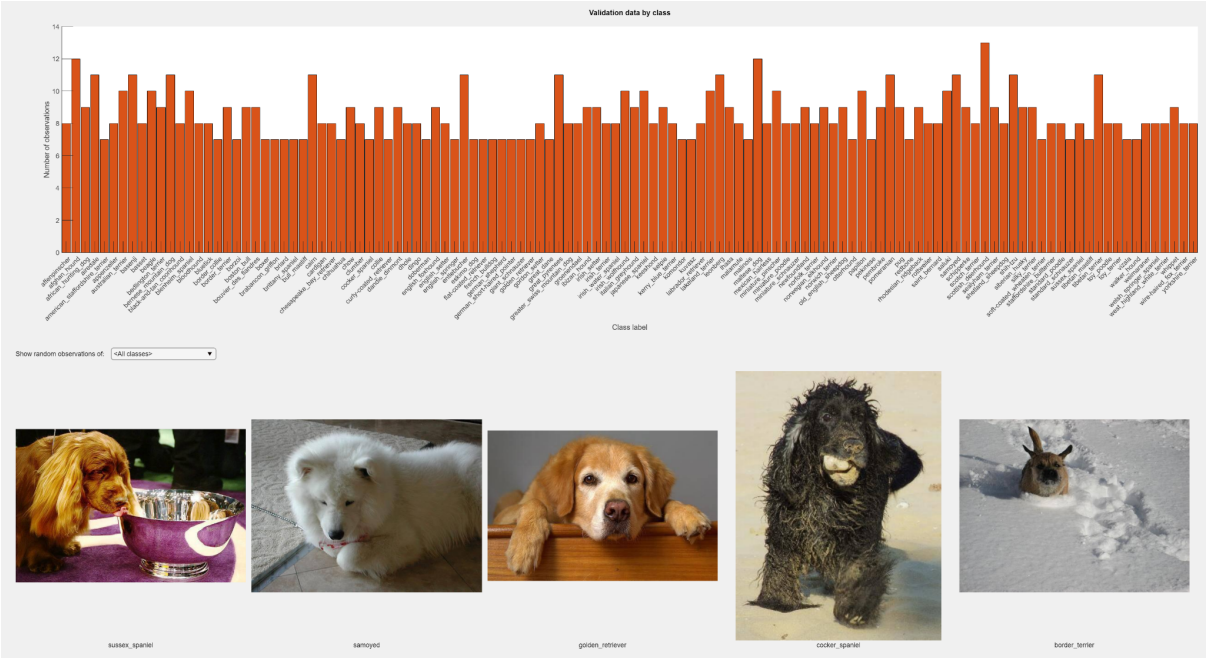
Initial structure

- All datapoints (images) in a single directory, along with a labels.csv file noting the labels for each available datapoint.

Modified structure

- Directories by labels (breeds) and all images placed in their respective label directories. (ref. ModifyDataset.mlx)





Baselines and Evaluation Metrics:

Initial baseline:

We used DummyClassifier from sklearn as our baseline model.

Using all the available strategies like:

1. most frequent:

```
[ ]  clf_mf = DummyClassifier(strategy='most_frequent', random_state=0)
     clf_mf.fit(X_train, y_train)
     clf_mf.score(X_test, y_test)
```

```
0.008998435054773083
```

2. stratified

```
[ ]  clf_str = DummyClassifier(strategy='stratified', random_state=0)
     clf_str.fit(X_train, y_train)
     clf_str.score(X_test, y_test)
```

```
0.007433489827856025
```

3. prior

```
[ ]  clf_pr = DummyClassifier(strategy='prior', random_state=0)
     clf_pr.fit(X_train, y_train)
     clf_pr.score(X_test, y_test)
```

```
0.008998435054773083
```

4. uniform

```
[ ]  clf_un = DummyClassifier(strategy='uniform', random_state=0)
     clf_un.fit(X_train, y_train)
     clf_un.score(X_test, y_test)
```

```
0.007433489827856025
```

5. constant

```
[ ]  clf_con = DummyClassifier(strategy='constant', random_state=0, constant='chihuahua')
     clf_con.fit(X_train, y_train)
     clf_con.score(X_test, y_test)
```

```
0.006259780907668232
```

Since the highest accuracy we got from these dummy models was ~0.009, i.e. 0.9%, we will use this as the baseline to identify that the trained model was able to achieve the minimum amount of success, and is not just identifying breeds at random.

Evaluation:

We decided to compare models by using the accuracy and the loss function values obtained from the validation data. We will also try to infer the difference between training and validation accuracy.

Experiments and Results:

For the Transfer Learning, we selected the following architectures:

1. AlexNet
2. GoogLeNet
3. ResNet18
4. ResNet50V2

All these CNNs were trained to classify between 1000 types of everyday objects and animals. And the input size for them is mostly 224x224 (or very similar, e.g 227x227 for AlexNet). We resize the images to 3 layers (RGB) 224x224 (ref. readFunctionTrain.m). We try to use as similar training hyperparameters as possible, only changing them as if the original values give unsatisfactory results in training.

Method:

Our general plan of using transfer learning was,

1. Trim the final layers of the pretrained networks.
2. Add a few new fully connected layers.
    a. e.g. for AlexNet, we added a 64 point fully connected layer with ReLU, and a 120 point fully connected layer with Softmax, and then a final classification layer.
3. Train the networks with these options
    a. Optimization algorithm - Stochastic Gradient Descent with momentum
    b. Initial Learning Rate - 0.001
        i. Changed this depending on the results of the training run and retrained.
    c. Learning Rate Scheduling - piecewise
    d. Shuffling data every epoch
    e. Total epochs - 30
        i. Did early stops when the network reached convergence and showed no further improvement
    f. MiniBatch size - 128
    g. Regularization - Ridge Regression
    h. Learning Rate drop factor - 0.1 per 10 epochs
    i. Batch normalization - via population

1. AlexNet

Few years ago, when AlexNet was introduced, it was a milestone architecture able to achieve high accuracy for classification datasets. But since then, there have been many other architectures which have been introduced which are proven to be much more powerful. So we wanted to evaluate how the 9 year old model will perform in comparison to the newer models like ResNets for not the original object classification, but Dog Breed identification task.

We use the above described method on the pretrained AlexNet (ref. TransferLearningAlexNet.mlx/TransferLearningAlexNet.pdf/TransferLearningAlexNet.html) -

Access Layers

Visualize the layers of AlexNet, what do we see about this architecture? What do we have to change for this to work for our new data?

```
layers = net.Layers;

layers
```

```
layers =
  25×1 Layer array with layers:

     1   'data'     Image Input                 227×227×3 images with 'zerocenter' normalization
     2   'conv1'    Convolution                 96 11×11×3 convolutions with stride [4  4] and padding [0  0  0  (
     3   'relu1'    ReLU                        ReLU
     4   'norm1'    Cross Channel Normalization cross channel normalization with 5 channels per element
     5   'pool1'    Max Pooling                 3×3 max pooling with stride [2  2] and padding [0  0  0  0]
     6   'conv2'    Grouped Convolution         2 groups of 128 5×5×48 convolutions with stride [1  1] and paddin
     7   'relu2'    ReLU                        ReLU
     8   'norm2'    Cross Channel Normalization cross channel normalization with 5 channels per element
     9   'pool2'    Max Pooling                 3×3 max pooling with stride [2  2] and padding [0  0  0  0]
    10   'conv3'    Convolution                 384 3×3×256 convolutions with stride [1  1] and padding [1  1  1
    11   'relu3'    ReLU                        ReLU
    12   'conv4'    Grouped Convolution         2 groups of 192 3×3×192 convolutions with stride [1  1] and paddi
    13   'relu4'    ReLU                        ReLU
    14   'conv5'    Grouped Convolution         2 groups of 128 3×3×192 convolutions with stride [1  1] and paddi
    15   'relu5'    ReLU                        ReLU
    16   'pool5'    Max Pooling                 3×3 max pooling with stride [2  2] and padding [0  0  0  0]
    17   'fc6'      Fully Connected             4096 fully connected layer
    18   'relu6'    ReLU                        ReLU
    19   'drop6'    Dropout                     50% dropout
    20   'fc7'      Fully Connected             4096 fully connected layer
    21   'relu7'    ReLU                        ReLU
    22   'drop7'    Dropout                     50% dropout
    23   'fc8'      Fully Connected             1000 fully connected layer
    24   'prob'     Softmax                     softmax
    25   'output'   Classification Output       crossentropyex with 'tench' and 999 other classes
```

Set up training data

```
rootFolder = 'train';

LabelData = readtable('.\labels.csv', 'Format', '%C%C');
BreedLabels = string(transpose(table2cell(unique(LabelData(:,'breed')))));

BreedCount = numel(BreedLabels)
```

```
BreedCount = 120
```

```
imds = imageDatastore(fullfile(rootFolder, BreedLabels), 'LabelSource', 'foldernames');

%imds = splitEachLabel(imds, 500, 'randomize') % we only need 500 images per class
imds.ReadFcn = @readFunctionTrain;
```

Take layers from Alex Net, then add our own

```
layers = layers(1:end-3);

layers(end+1) = fullyConnectedLayer(64, 'Name', 'special_2');
layers(end+1) = reluLayer;
layers(end+1) = fullyConnectedLayer(BreedCount, 'Name', 'fc8_2 ');
layers(end+1) = softmaxLayer;
layers(end+1) = classificationLayer()
```

```
layers =
  27×1 Layer array with layers:

     1   'data'        Image Input                   227×227×3 images with 'zerocenter' normalization
     2   'conv1'       Convolution                   96 11×11×3 convolutions with stride [4  4] and padding [0  0  0
     3   'relu1'       ReLU                          ReLU
     4   'norm1'       Cross Channel Normalization   cross channel normalization with 5 channels per element
     5   'pool1'       Max Pooling                   3×3 max pooling with stride [2  2] and padding [0  0  0  0]
     6   'conv2'       Grouped Convolution           2 groups of 128 5×5×48 convolutions with stride [1  1] and pad
     7   'relu2'       ReLU                          ReLU
     8   'norm2'       Cross Channel Normalization   cross channel normalization with 5 channels per element
     9   'pool2'       Max Pooling                   3×3 max pooling with stride [2  2] and padding [0  0  0  0]
    10   'conv3'       Convolution                   384 3×3×256 convolutions with stride [1  1] and padding [1  1
    11   'relu3'       ReLU                          ReLU
    12   'conv4'       Grouped Convolution           2 groups of 192 3×3×192 convolutions with stride [1  1] and pa
    13   'relu4'       ReLU                          ReLU
    14   'conv5'       Grouped Convolution           2 groups of 128 3×3×192 convolutions with stride [1  1] and pa
    15   'relu5'       ReLU                          ReLU
    16   'pool5'       Max Pooling                   3×3 max pooling with stride [2  2] and padding [0  0  0  0]
    17   'fc6'         Fully Connected               4096 fully connected layer
    18   'relu6'       ReLU                          ReLU
    19   'drop6'       Dropout                       50% dropout
    20   'fc7'         Fully Connected               4096 fully connected layer
    21   'relu7'       ReLU                          ReLU
    22   'drop7'       Dropout                       50% dropout
    23   'special_2'   Fully Connected               64 fully connected layer
    24   ''            ReLU                          ReLU
```

Fine-tune learning rates [advanced]

```
layers(end-2).WeightLearnRateFactor = 10;
layers(end-2).WeightL2Factor = 1;
layers(end-2).BiasLearnRateFactor = 20;
layers(end-2).BiasL2Factor = 0;
```

Other training options

```
opts = trainingOptions('sgdm', ...
    'LearnRateSchedule', 'none',...
    'InitialLearnRate', .0001,...
    'MaxEpochs', 20, ...
    'MiniBatchSize', 128);
```

Test GPU before running?

```
gpuDevice()
```

```
ans =
  CUDADevice with properties:

                      Name: 'GeForce GTX 1660 Ti'
                     Index: 1
         ComputeCapability: '7.5'
            SupportsDouble: 1
             DriverVersion: 11.1000
            ToolkitVersion: 11
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 6.4425e+09
           AvailableMemory: 5.2528e+09
       MultiprocessorCount: 24
              ClockRateKHz: 1590000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
           DeviceAvailable: 1
            DeviceSelected: 1
```

Train!

```
convnet = trainNetwork(imds, layers, opts);
```

```
Training on single GPU.
Initializing input data normalization.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:02 |        0.78% |       6.3083 |    1.0000e-04 |
|       1 |          50 |       00:00:55 |        0.78% |       4.8987 |    1.0000e-04 |
|       2 |         100 |       00:01:48 |        0.78% |       4.8134 |    1.0000e-04 |
|       2 |         150 |       00:02:42 |        0.78% |       4.8625 |    1.0000e-04 |
|       3 |         200 |       00:03:35 |        1.56% |       4.7766 |    1.0000e-04 |
|       4 |         250 |       00:04:29 |        2.34% |       4.6281 |    1.0000e-04 |
|       4 |         300 |       00:05:22 |        7.03% |       4.6533 |    1.0000e-04 |
|       5 |         350 |       00:06:15 |        7.03% |       4.4769 |    1.0000e-04 |
|       6 |         400 |       00:07:09 |        4.69% |       4.5620 |    1.0000e-04 |
|       6 |         450 |       00:08:03 |        7.03% |       4.3130 |    1.0000e-04 |
|       7 |         500 |       00:08:56 |       14.84% |       4.2020 |    1.0000e-04 |
|       7 |         550 |       00:09:51 |        9.38% |       4.1246 |    1.0000e-04 |
|       8 |         600 |       00:10:44 |       13.28% |       3.8384 |    1.0000e-04 |
|       9 |         650 |       00:11:38 |       22.66% |       3.7281 |    1.0000e-04 |
|       9 |         700 |       00:12:32 |       25.78% |       3.3433 |    1.0000e-04 |
|      10 |         750 |       00:13:25 |       28.12% |       3.4208 |    1.0000e-04 |
|      11 |         800 |       00:14:19 |       29.69% |       3.0885 |    1.0000e-04 |
|      11 |         850 |       00:15:14 |       32.03% |       2.8115 |    1.0000e-04 |
|      12 |         900 |       00:16:09 |       32.03% |       2.7894 |    1.0000e-04 |
|      13 |         950 |       00:17:04 |       29.69% |       2.6385 |    1.0000e-04 |
|      13 |        1000 |       00:17:58 |       40.62% |       2.1943 |    1.0000e-04 |
|      14 |        1050 |       00:18:52 |       43.75% |       2.2778 |    1.0000e-04 |
|      14 |        1100 |       00:19:47 |       40.62% |       2.3786 |    1.0000e-04 |
|      15 |        1150 |       00:20:41 |       34.38% |       2.2446 |    1.0000e-04 |
|      16 |        1200 |       00:21:35 |       37.50% |       2.3385 |    1.0000e-04 |
|      16 |        1250 |       00:22:29 |       44.53% |       2.0821 |    1.0000e-04 |
|      17 |        1300 |       00:23:22 |       39.06% |       2.1643 |    1.0000e-04 |
|      18 |        1350 |       00:24:16 |       46.09% |       2.0995 |    1.0000e-04 |
|      18 |        1400 |       00:25:10 |       47.66% |       2.0064 |    1.0000e-04 |
|      19 |        1450 |       00:26:05 |       46.09% |       1.7603 |    1.0000e-04 |
|      19 |        1500 |       00:27:01 |       51.56% |       1.9441 |    1.0000e-04 |
|      20 |        1550 |       00:27:55 |       43.75% |       2.3045 |    1.0000e-04 |
|      20 |        1580 |       00:28:28 |       51.56% |       1.6720 |    1.0000e-04 |
|=========================================================================================|
Training finished: Max epochs completed.
```

Determine overall accuracy

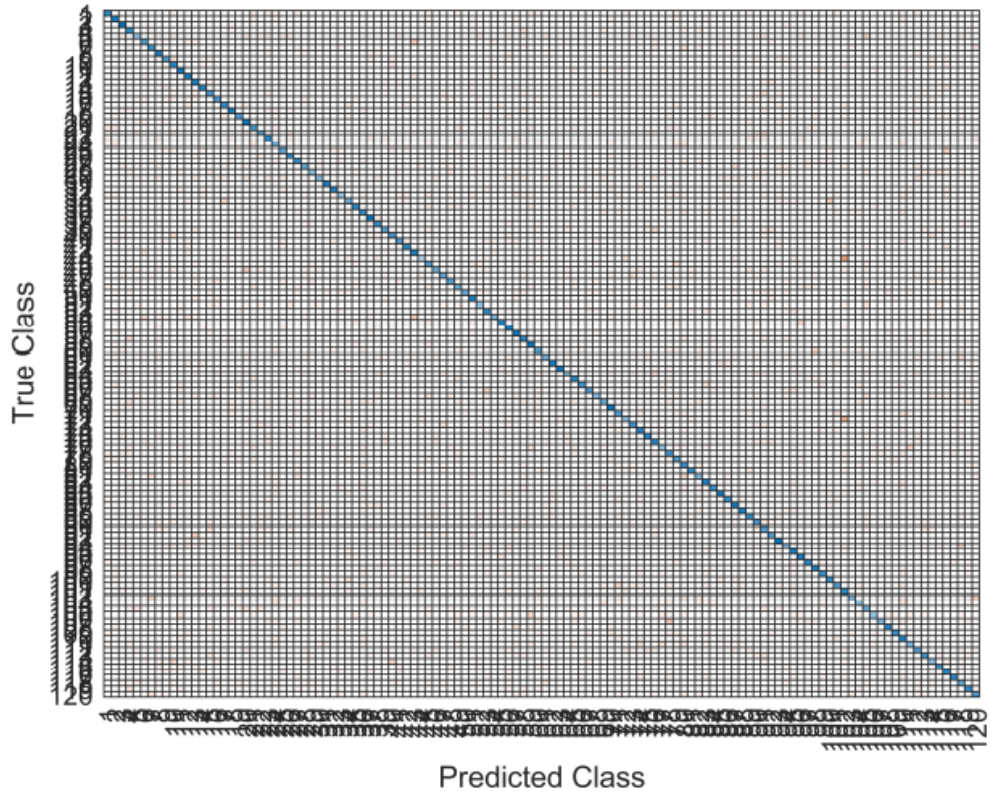
```
confMat = confusionmat(testDS.Labels, labels);
confMat = confMat./sum(confMat,2);
OverallAccuracy = mean(diag(confMat))
```

```
OverallAccuracy = 0.6869
```

```
BreedAcc = diag(confMat).';
int_confMat = int64(confMat .* 10000)
```

```
int_confMat = 120×120 int64 matrix
   7500      0      0      0      0      0      0      0      0    125      0 ···
      0   8707      0      0      0      0      0      0      0      0      0
      0      0   9302      0      0      0      0      0      0      0      0
      0      0      0   7570      0      0    187      0      0      0      0
      0      0    135      0   3919      0      0    135      0    270      0
      0      0      0      0      0   6154      0      0    128      0      0
      0      0      0    196      0      0   7059      0      0      0      0
      0      0      0      0      0      0      0   8364      0      0      0
      0      0      0      0      0    122      0    122   6341    976      0
      0      0      0      0      0      0      0      0    381   7905      0
      ⋮
```

```
confusionchart(int_confMat )
```



So the initial training accuracy we achieved was **~0.69**, which was definitely better than the baseline that we had set, but we wanted to see if the model can perform better with modified hyperparams and layers.

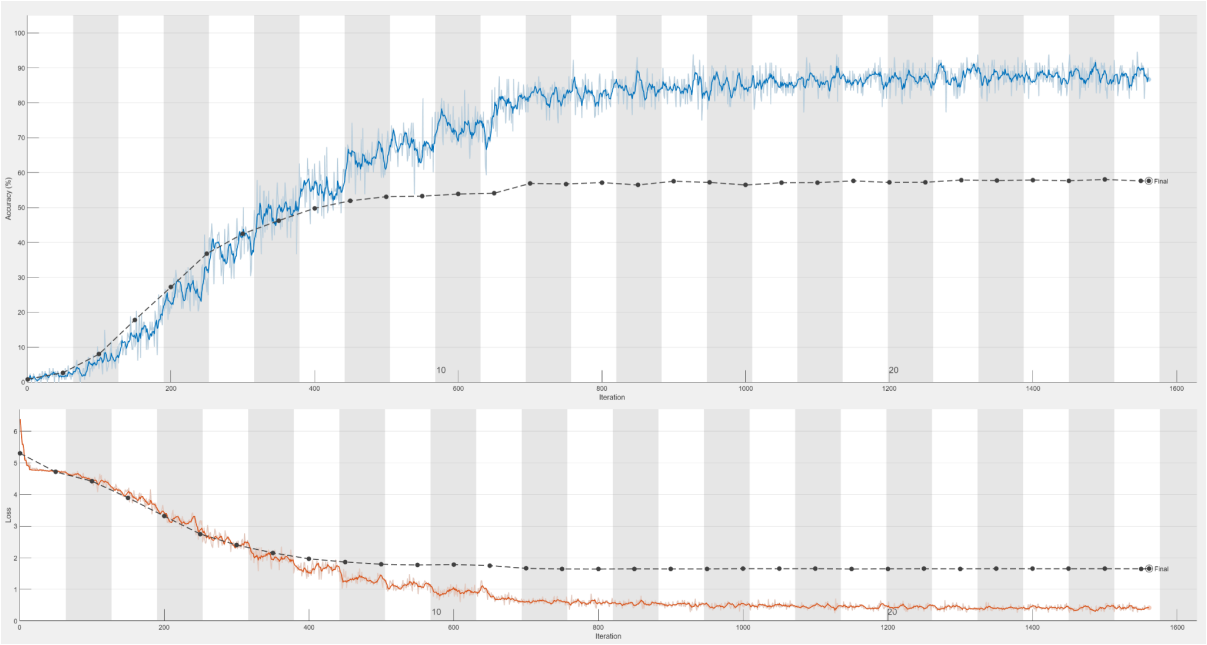
Retraining with tuned hyperparameters

Then, using the Deep learning designer in MATLAB, we were able to modify the layers visually and tune hyperparameters while observing the validation accuracy during training in real time (every 50 iterations).

The final training of AlexNet yielded much better results. Reaching -

Training Accuracy: ~0.87

Validation Accuracy: ~0.58



Results

Validation accuracy:	57.63%
Training finished:	Stopped manually

Training Time

Start time:	
Elapsed time:	24 min 51 sec

Training Cycle

Epoch:	25 of 30
Iteration:	1561 of 1890
Iterations per epoch:	63
Maximum iterations:	1890

Validation

Frequency:	50 iterations

Other Information

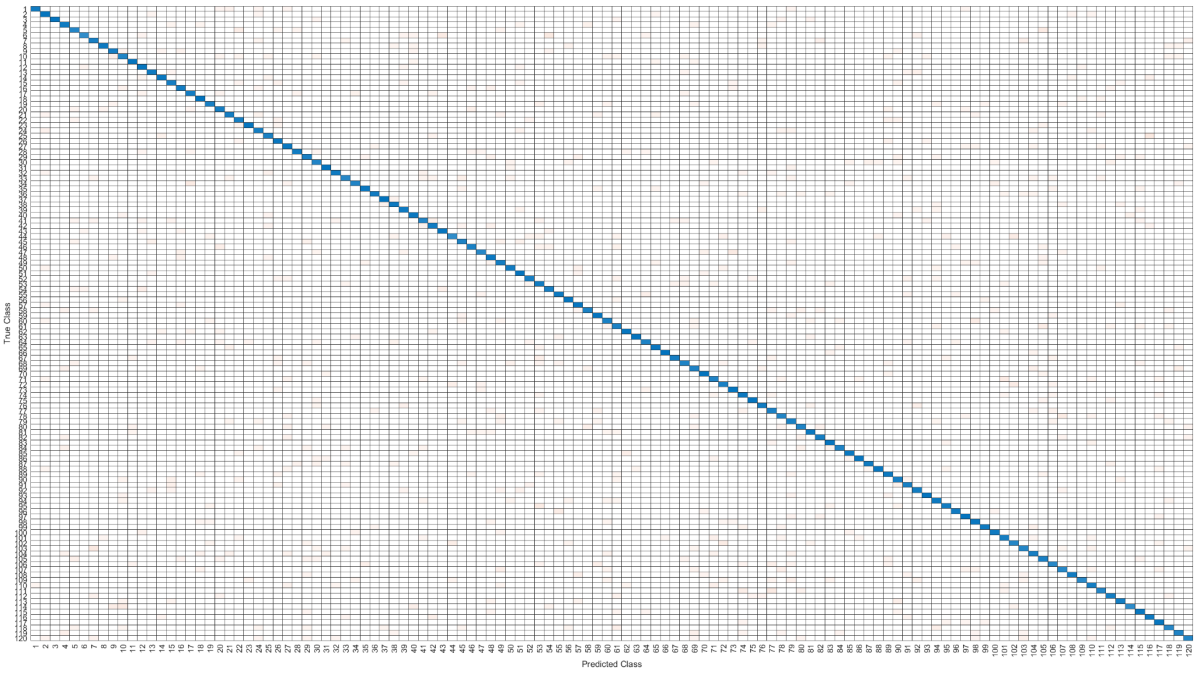
Hardware resource:	Single GPU
Learning rate schedule:	Piecewise
Learning rate:	1e-05

Accuracy

Training (smoothed)
Training
Validation

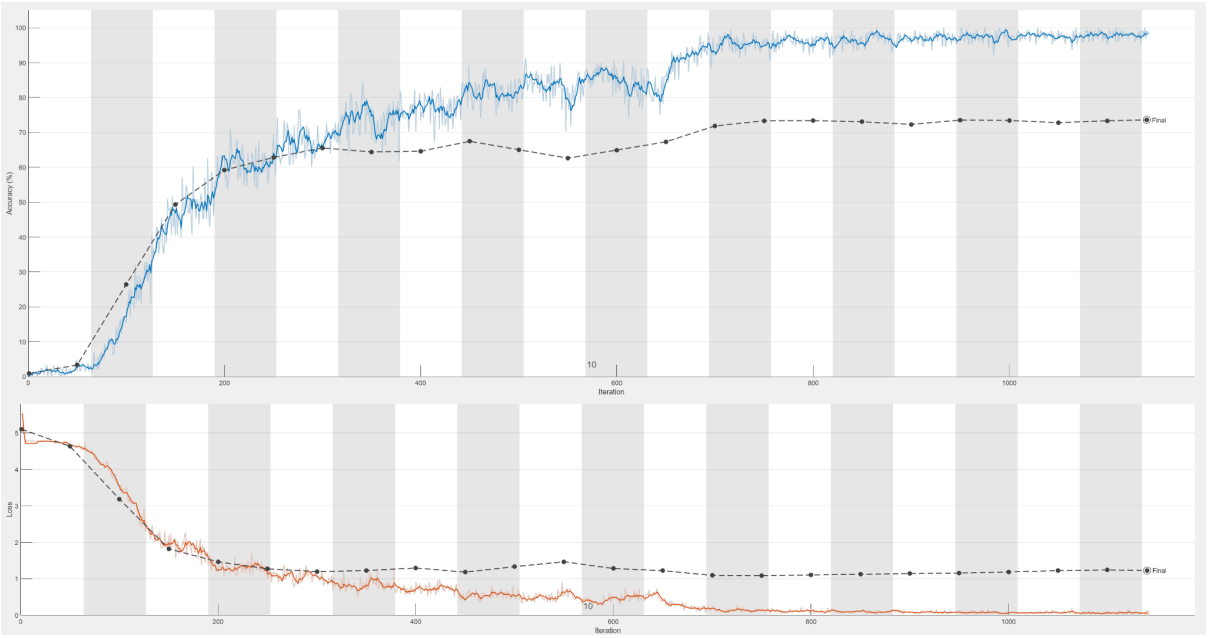
Loss

Training (smoothed)
Training
Validation



2. GoogLeNet

Similarly, we obtained following results by retraining the pretrained GooLeNet model -



Results

Validation accuracy:	73.59%
Training finished:	Stopped manually

Training Time

Start time:	
Elapsed time:	135 min 37 sec

Training Cycle

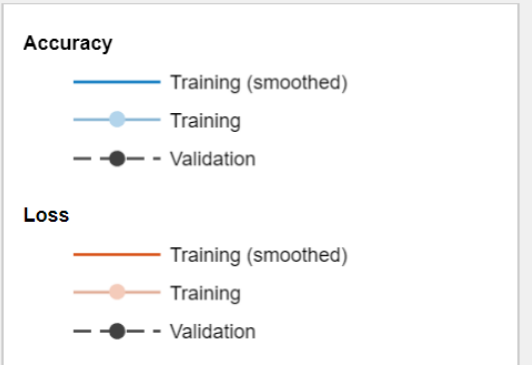
Epoch:	19 of 30
Iteration:	1140 of 1890
Iterations per epoch:	63
Maximum iterations:	1890

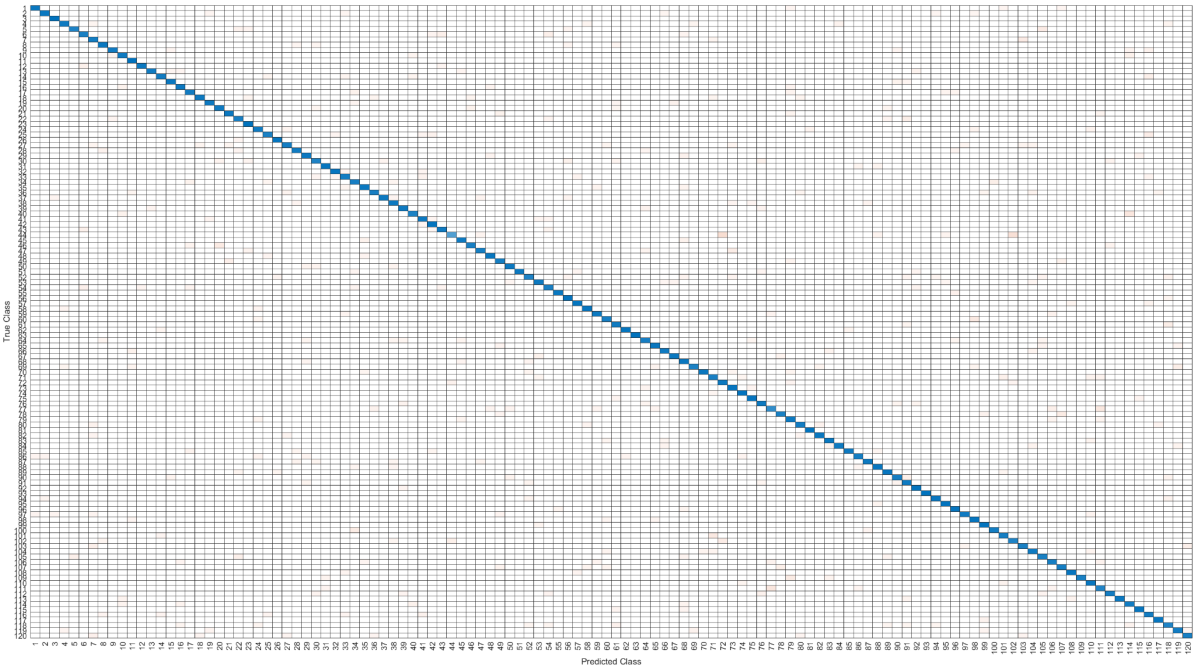
Validation

Frequency:	50 iterations

Other Information

Hardware resource:	Single GPU
Learning rate schedule:	Piecewise
Learning rate:	0.001



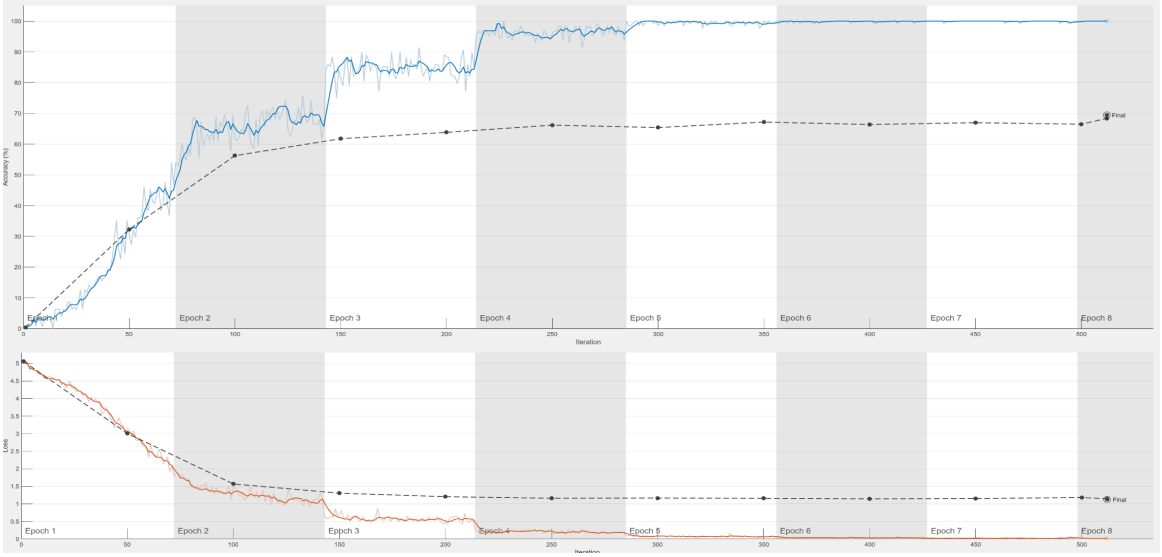


Training Accuracy: ~0.97

Validation Accuracy: ~0.73

3. ResNet18

Similarly, we obtained following results by retraining the pretrained ResNet18 model -



Results

Validation accuracy:	69.42%
Training finished:	Stopped manually

Training Time

Start time:	
Elapsed time:	12 min 15 sec

Training Cycle

Epoch:	8 of 30
Iteration:	512 of 2130
Iterations per epoch:	71
Maximum iterations:	2130

Validation

Frequency:	50 iterations

Other Information

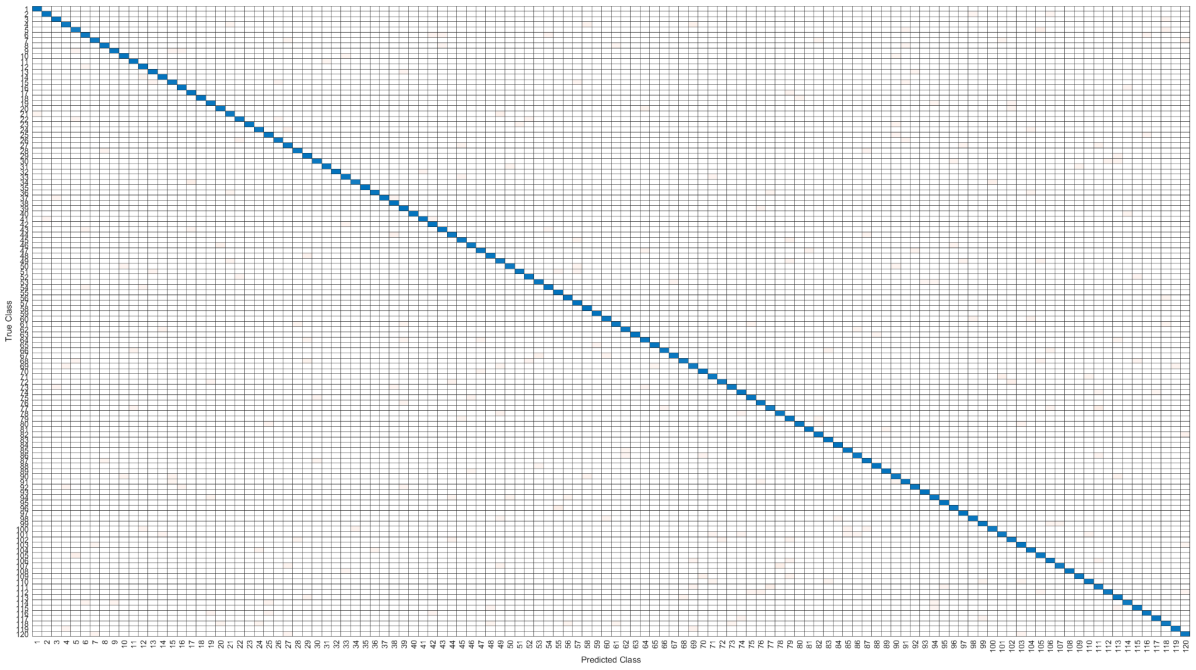
Hardware resource:	Single GPU
Learning rate schedule:	Piecewise
Learning rate:	0.01

Accuracy

Training (smoothed)
Training
Validation

Loss

Training (smoothed)
Training
Validation



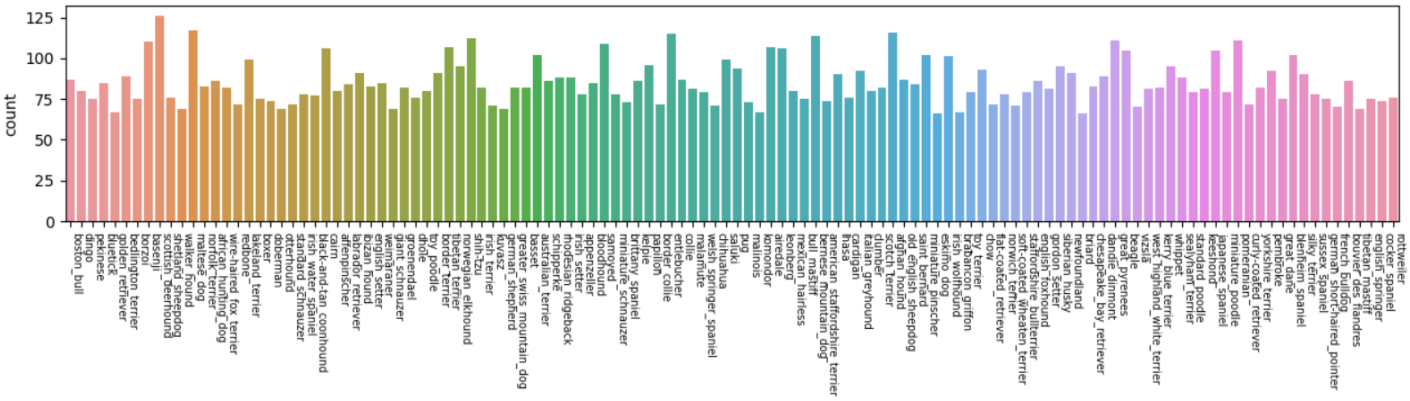
Training Accuracy: ~1.00

Validation Accuracy: ~0.69

4. ResNet50V2

For ResNet50V2, we tried a slightly different approach and training environment.

First we show the numbers of each breed in the dataset on a plot.



Then we set up training options. (learning rate = 0.00001, max epoch number = 30)

```python
#set learning rate as 0.00001
opt = tf.keras.optimizers.Adam(learning_rate = 1e-5)
model.compile(optimizer=opt,loss='categorical_crossentropy',metrics=['accuracy'])

#defining train step and validate step (total value/batch size)
train_step = train_set.n//train_set.batch_size
validate_step = validate_set.n//validate_set.batch_size

#training model
resnet50 = model.fit(train_set,validation_data = validate_set,epochs = 30,steps_per_epoch = train_step, validation_steps = validate_step)
```

Next, we started to train the model.

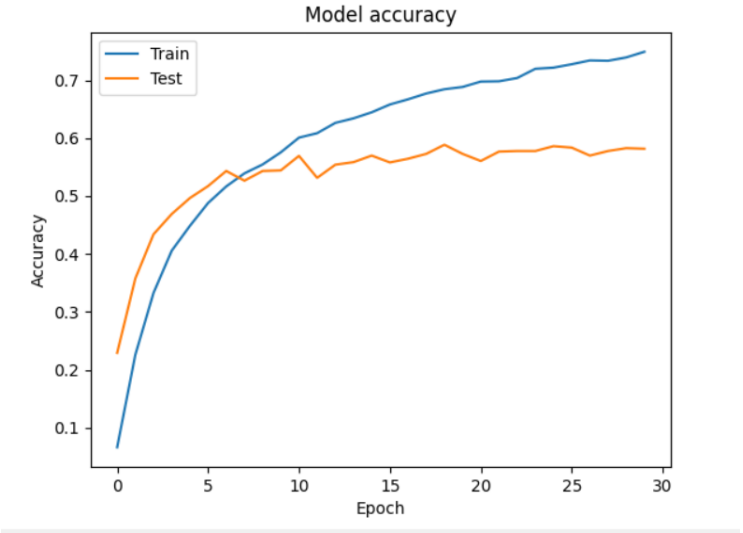
```
Epoch 1/30
511/511 [==============================] - 381s 743ms/step - loss: 7.9738 - accuracy: 0.0664 - val_loss: 4.0821 - val_accuracy: 0.2293
Epoch 2/30
511/511 [==============================] - 384s 751ms/step - loss: 4.9197 - accuracy: 0.2260 - val_loss: 3.0617 - val_accuracy: 0.3578
Epoch 3/30
511/511 [==============================] - 388s 759ms/step - loss: 3.8942 - accuracy: 0.3329 - val_loss: 2.7324 - val_accuracy: 0.4341
Epoch 4/30
511/511 [==============================] - 393s 768ms/step - loss: 3.2953 - accuracy: 0.4057 - val_loss: 2.5169 - val_accuracy: 0.4690
Epoch 5/30
511/511 [==============================] - 389s 762ms/step - loss: 2.9709 - accuracy: 0.4485 - val_loss: 2.4250 - val_accuracy: 0.4966
Epoch 6/30
511/511 [==============================] - 389s 761ms/step - loss: 2.7118 - accuracy: 0.4880 - val_loss: 2.3153 - val_accuracy: 0.5172
Epoch 7/30
511/511 [==============================] - 385s 753ms/step - loss: 2.5407 - accuracy: 0.5167 - val_loss: 2.3123 - val_accuracy: 0.5433
Epoch 8/30
511/511 [==============================] - 372s 728ms/step - loss: 2.3878 - accuracy: 0.5392 - val_loss: 2.3506 - val_accuracy: 0.5266
Epoch 9/30
511/511 [==============================] - 354s 693ms/step - loss: 2.2469 - accuracy: 0.5545 - val_loss: 2.3438 - val_accuracy: 0.5433
Epoch 10/30
511/511 [==============================] - 363s 710ms/step - loss: 2.1445 - accuracy: 0.5755 - val_loss: 2.3369 - val_accuracy: 0.5443
Epoch 11/30
511/511 [==============================] - 374s 732ms/step - loss: 1.9786 - accuracy: 0.6008 - val_loss: 2.2688 - val_accuracy: 0.5694
Epoch 12/30
511/511 [==============================] - 391s 765ms/step - loss: 1.9319 - accuracy: 0.6086 - val_loss: 2.3956 - val_accuracy: 0.5315
Epoch 13/30
511/511 [==============================] - 390s 763ms/step - loss: 1.8264 - accuracy: 0.6263 - val_loss: 2.3420 - val_accuracy: 0.5541
Epoch 14/30
511/511 [==============================] - 386s 756ms/step - loss: 1.7722 - accuracy: 0.6342 - val_loss: 2.3142 - val_accuracy: 0.5586
Epoch 15/30
511/511 [==============================] - 387s 758ms/step - loss: 1.7016 - accuracy: 0.6446 - val_loss: 2.2541 - val_accuracy: 0.5699
```

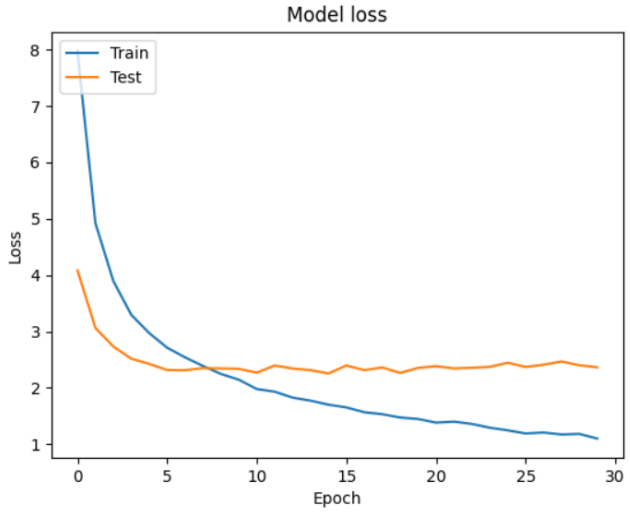
```
Epoch 16/30
511/511 [==============================] - 382s 747ms/step - loss: 1.6529 - accuracy: 0.6580 - val_loss: 2.3956 - val_accuracy: 0.5581
Epoch 17/30
511/511 [==============================] - 352s 689ms/step - loss: 1.5670 - accuracy: 0.6670 - val_loss: 2.3143 - val_accuracy: 0.5645
Epoch 18/30
511/511 [==============================] - 363s 710ms/step - loss: 1.5325 - accuracy: 0.6770 - val_loss: 2.3609 - val_accuracy: 0.5728
Epoch 19/30
511/511 [==============================] - 381s 745ms/step - loss: 1.4739 - accuracy: 0.6844 - val_loss: 2.2652 - val_accuracy: 0.5886
Epoch 20/30
511/511 [==============================] - 386s 754ms/step - loss: 1.4473 - accuracy: 0.6883 - val_loss: 2.3533 - val_accuracy: 0.5728
Epoch 21/30
511/511 [==============================] - 390s 764ms/step - loss: 1.3831 - accuracy: 0.6976 - val_loss: 2.3839 - val_accuracy: 0.5605
Epoch 22/30
511/511 [==============================] - 370s 725ms/step - loss: 1.4004 - accuracy: 0.6981 - val_loss: 2.3441 - val_accuracy: 0.5768
Epoch 23/30
511/511 [==============================] - 372s 729ms/step - loss: 1.3597 - accuracy: 0.7037 - val_loss: 2.3556 - val_accuracy: 0.5778
Epoch 24/30
511/511 [==============================] - 380s 744ms/step - loss: 1.2931 - accuracy: 0.7197 - val_loss: 2.3716 - val_accuracy: 0.5778
Epoch 25/30
511/511 [==============================] - 364s 712ms/step - loss: 1.2464 - accuracy: 0.7216 - val_loss: 2.4446 - val_accuracy: 0.5861
Epoch 26/30
511/511 [==============================] - 351s 688ms/step - loss: 1.1908 - accuracy: 0.7275 - val_loss: 2.3700 - val_accuracy: 0.5837
Epoch 27/30
511/511 [==============================] - 343s 672ms/step - loss: 1.2072 - accuracy: 0.7341 - val_loss: 2.4100 - val_accuracy: 0.5699
Epoch 28/30
511/511 [==============================] - 343s 671ms/step - loss: 1.1726 - accuracy: 0.7335 - val_loss: 2.4671 - val_accuracy: 0.5778
Epoch 29/30
511/511 [==============================] - 344s 672ms/step - loss: 1.1829 - accuracy: 0.7393 - val_loss: 2.4007 - val_accuracy: 0.5827
Epoch 30/30
511/511 [==============================] - 344s 674ms/step - loss: 1.1001 - accuracy: 0.7491 - val_loss: 2.3637 - val_accuracy: 0.5817
```

After that, we evaluated the model to get the total accuracy and loss, then started to predict the test set.

```
128/128 [==============================] - 65s 505ms/step - loss: 2.3641 - accuracy: 0.5910
647/647 [==============================] - 319s 493ms/step
```

Finally, we got the model accuracy and model loss plots of the train set and test set.





Training Accuracy: ~0.75

Validation Accuracy: ~0.59

We can also let the user enter an image's path, and output the prediction of what breed of dog is in the image.

e.g.

```
Enter an image path to predict: D:\\Chris\\我的文件\\CSE598\\Project\\dog-breed-identification\\test\\00c14d34a725db12068402e4ce714d4c.jpg
['saint_bernard']
```

00c14d34a725db12068402e4ce714d4c.jpg

Conclusion:

	AlexNet	GoogLeNet	ResNet18	ResNet50V2
Training acc	0.87	0.97	1.00	0.75
Validation acc	0.58	0.73	0.69	0.59

As the results shown above, we discovered that ResNet18 got the best accuracy in the training section, it got 100% accuracy which is incredible. And GoogLeNet got the best accuracy with 73% in the validation section. On the other hand, GoogLeNet and ResNet18 both performed well in reducing the losses. In general, we think that GoogLeNet is the best model between these models.

By doing this project, we understand that transfer learning models are way more powerful than the DummyClassifier baseline model. Although they often take a long time in training, the results still satisfied us.

References:

1. Dataset source: Kaggle - Dog Breed Identification
    a. https://www.kaggle.com/c/dog-breed-identification/data
2. MATLAB - Transfer Learning
    a. https://www.mathworks.com/matlabcentral/fileexchange/62990-deep-learning-tutorial-series
    b. https://www.mathworks.com/help/deeplearning/gs/get-started-with-transfer-learning.html
    c. https://www.mathworks.com/help/deeplearning/ug/transfer-learning-using-alexnet.html
    d. https://www.mathworks.com/help/deeplearning/ug/transfer-learning-with-deep-network-designer.html