Report

Model Architecture

The difficult part of this task is to reduce parameter numbers while maintaining the model accuracy. As we know some model architecture and techniques save parameters. For an instance, convolutional layer uses fewer parameters than fully connected layer. Pooling layer does not use any parameters. Inception block saves parameters very well. The input to the convolutional layer can vary, only the last fully connect layer will be impacted for most CNN model architecture. Some models have good trade-off between performance and accuracy, such as MobileNet, and Xception, EfficientNet, etc.

MobileNet uses depthwise and pointwise convolutions convolution to reduce the model size and improve efficiency largely. This technique has also been used in Xception as well(Howard et al.). As we know, the inception block used much less parameters than traditional convolution. Also people tried factorization and bottleneck to reduce the number of parameters, such as SqueezeNet. a standard convolution is factorized into a depthwise convolution and a 1 × 1 pointwise convolution to reduce the number of parameters used in the model. MobileNets use both batchnorm and ReLU nonlinearities for both layers.

The first layer of MobileNetV1 is a full convolution. Then it used the factorized layer with depthwise convolution, 1 × 1 pointwise convolution as well as batchnorm and ReLU for each convolutional layer. Down sampling is handled with strided convolution in the depthwise convolutions. A final average pooling reduces the spatial resolution to 1 before the fully connected layer. Counting depthwise and pointwise convolutions as separate layers, MobileNet has 28 layers.

We can use optimizer - RMSprop (Tieleman and Hinton) to train the model according the original paper. We can decide whether to do data augmentation and regularization based on the model performance. Small model may not overfit that much. According to the paper, we implicitly applied the resolution multiplier by reducing the input size. Also based on the model architecture

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	3 × 3 × 3 × 32	224 × 224 × 3
Conv dw / s1	3 × 3 × 32 dw	112 × 112 × 32
Conv / s1	1 × 1 × 32 × 64	112 × 112 × 32
Conv dw / s2	3 × 3 × 64 dw	112 × 112 × 64
Conv / s1	1 × 1 × 64 × 128	56 × 56 × 64
Conv dw / s1	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 128	56 × 56 × 128
Conv dw / s2	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 256	28 × 28 × 128
Conv dw / s1	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 256	28 × 28 × 256
Conv dw / s2	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 512	14 × 14 × 256
5× Conv dw / s1	3 × 3 × 512 dw	14 × 14 × 512
5× Conv / s1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw / s2	3 × 3 × 512 dw	14 × 14 × 512
Conv / s1	1 × 1 × 512 × 1024	7 × 7 × 512
Conv dw / s2	3 × 3 × 1024 dw	7 × 7 × 1024
Conv / s1	1 × 1 × 1024 × 1024	7 × 7 × 1024
Avg Pool / s1	Pool 7 × 7	7 × 7 × 1024
FC / s1	1024 × 1000	1 × 1 × 1024
Softmax / s1	Classifier	1 × 1 × 1000

It only has one fully connected layer output from the average pooling. It is not fussy with input shape. Smaller input size may not get 7*7*1024 before average pooling. But it only affects width and height. It will be less than 7. The channel is still 1024 however. After average pooling, it will still get 1024 input nodes for dense layer. For our case, we only need to change the number of classes at the final step.

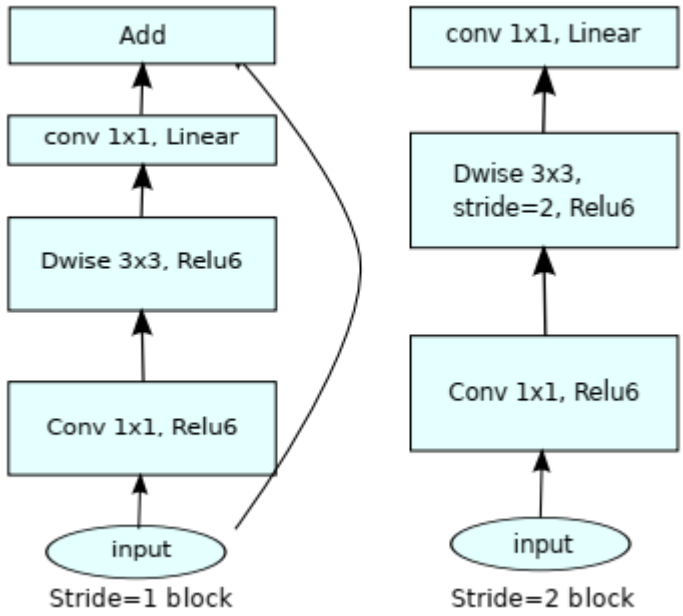
From experiments, I found the problem that output shape dropped to 1 by 1 quickly with small input image size after the 12^{th} depthwise covolution which makes convolution in the deep layers meaningless. It goes the same with MobileNetV2 and MobileNetV3. We could either use transposed convolution to up sample or reduce the stride to keep the dimensions. Alternatively, we could even chop the deep layers to make the model shallow. If we remove some layers to make the network less deep, we could save some parameters.

For level1 task, I decided to use MobileNetV2 since it is very similar to the original MobileNetV1, except that it uses inverted residual blocks with bottleneck layers. It has a much lower parameter count than the original MobileNet. MobileNets support any input size greater than 32 x 32, with larger image sizes offering better performance(Sandler et al.).

Inspired by the manifold interest, mobileNetV2 introduced the Linear Bottlenecks which was also used as inverted residuals to propagate gradients. According to the paper, the real-world images on a manifold can be represented using low dimensional layers (thin layers). Relu however caused the information loss in thin layers. The channel was expanded using pointwise convolution for Relu6 activation, then depthwise covolution for Relu6, finally shrunk using pointwise convolution to Linear bottleneck. When the bottleneck dimensions are the same, the previous bottleneck is added to the current one as residuals. The block is described as below,

Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 1: *Bottleneck residual block* transforming from k to k' channels, with stride s, and expansion factor t.



Batch normalization and dropout were used. V2 is faster than V1 mainly due to less channels for each block after bottleneck thin layers introduced. It is more accurate primarily because of residual layers added. It is mentioned the expansion rate depends on the network size. Resolution and width multiplier as tuneable hyper parameters to trade off performance and accuracy. Again, resolution has already dropped since our input size is 32 by 32 which means we do not need to apply resolution multiplier at all. When we used input size 32 by 32, from block 12 the shape dropped to 1 by 1.

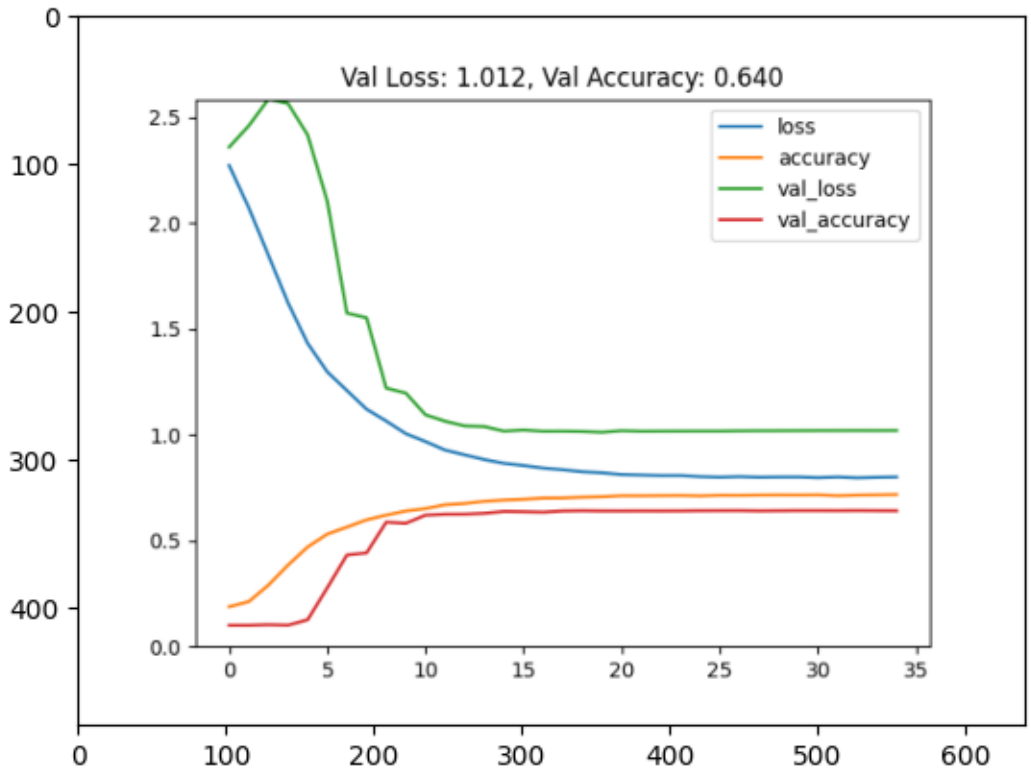
Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Experiments

In both task1 and task2 models, mobilenetv2 preprocess is imbedded in the model itself so that we can import test data directly to the model.

In level1 task, I trained the base model from scratch by adding L2 norm for weight decay and reducing alpha – width mulptiplier. I tuned alpha, learning rate, momentum, weight decay. The best setting found to be 0.9, 0.0005, 0.9, 0.

The training graph for the best model hyper parameter tuned is as below,



The model converged well. And there is no overfitting issue since the validation accuracy is like training.

The original model has nearly 2 million parameters.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_103 (InputLayer)      [(None, 32, 32, 3)]       0

 tf.math.truediv_50 (TFOpLa  (None, 32, 32, 3)         0
 mbda)

 tf.math.subtract_50 (TFOpL  (None, 32, 32, 3)         0
 ambda)

 mobilenetv2_0.90_32 (Funct  (None, 10)                1904954
 ional)

=================================================================
Total params: 1904954 (7.27 MB)
Trainable params: 1873610 (7.15 MB)
Non-trainable params: 31344 (122.44 KB)
_________________________________________________________________
```

The speed is low. The inference time of the model in milliseconds per image per CPU is around 1470 milliseconds per image.

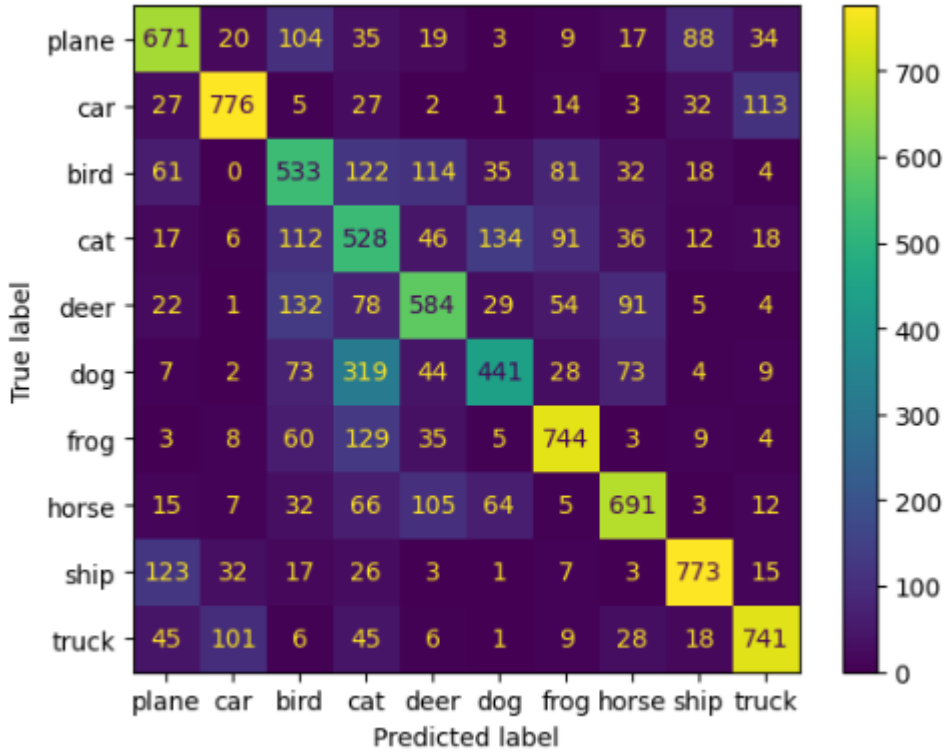
Applied on test data, model achieves 0.65 accuracy. Detailed classification report for each class is as below,

```
The inference time of the model in milliseconds per image per CPU is 1469.974303246
              precision    recall  f1-score   support

       plane       0.68      0.67      0.67      1000
         car       0.81      0.78      0.79      1000
        bird       0.50      0.53      0.51      1000
         cat       0.38      0.53      0.44      1000
        deer       0.61      0.58      0.60      1000
         dog       0.62      0.44      0.51      1000
        frog       0.71      0.74      0.73      1000
       horse       0.71      0.69      0.70      1000
        ship       0.80      0.77      0.79      1000
       truck       0.78      0.74      0.76      1000

    accuracy                           0.65     10000
   macro avg       0.66      0.65      0.65     10000
weighted avg       0.66      0.65      0.65     10000
```

The accuracy is fine in general. The training data is class balanced. However, the model trained performs bad for some categories such as bird, cat, deer, dog. They are confused with each other. We need to do further work to improve this, such as data augmentation to increase the sample sizes.



In level2 task, I removed bottleneck layers after block_12_add layer which output 2*2*96 shape. The bottlenecks removed are highlighted as below,

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

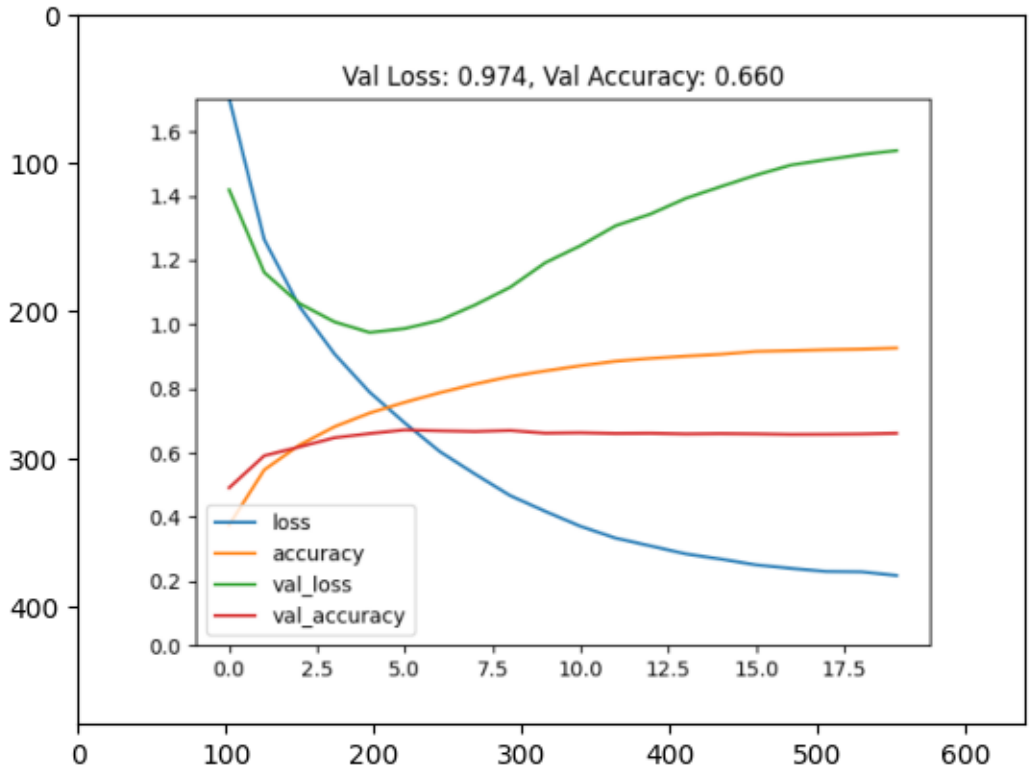
The final convolution was replaced with a dense layer to output 10 classes. The number of neurons input for dense layers is 1280 in the original base model. This number was tuned by adjusting the channels in the final convolutions. I also used dropout for them in the final dense layer. The new model architecture is shown as below with alpha at 1, dropout at 0,

Layer (type)	Output Shape	Param #
input_27 (InputLayer)	[(None, 32, 32, 3)]	0
tf.math.truediv_23 (TFOpLambda)	(None, 32, 32, 3)	0
tf.math.subtract_23 (TFOpLambda)	(None, 32, 32, 3)	0
model_29 (Functional)	(None, 2, 2, 96)	558656
conv2d_13 (Conv2D)	(None, 2, 2, 1280)	124160
batch_normalization_9 (BatchNormalization)	(None, 2, 2, 1280)	5120
re_lu_40 (ReLU)	(None, 2, 2, 1280)	0
global_average_pooling2d_10 (GlobalAveragePooling2D)	(None, 1280)	0
dropout_5 (Dropout)	(None, 1280)	0
dense_4 (Dense)	(None, 10)	12810

Total params: 700746 (2.67 MB)
Trainable params: 682058 (2.60 MB)
Non-trainable params: 18688 (73.00 KB)

After removing the 2 bottlenecks, the total parameter dropped to 0.7 million. We could either increase the default alpha – depth multiplier to make more complex models or reduce it for regularization depending on model performance and accuracy trade-off.

I tuned the hyperparameters of alpha – depth multiplier, learning rate, momentum, weight decay, fully connect layer input unit number, fully connected layer input unit dropout rate. The best model used 0.9 as depth multiplier, 0.0001 as learning rate, 0.9 as momentum, 1e-05 as weight decay, 1000 as fully connect layer input unit number, 0.2 as fully connected layer input unit dropout rate with early stopping. The training graph for the best model hyper parameter tuned is as below,



There is an overfitting problem with this model. The early stopping has helped regularization. We could increase the weight decay, dropout rate to further regularize the model. We could also further reduce the learning rate during training to approach the minimum point better. Need to experiment in detailed level considering the settings we have already tuned.

This model fine-tuned has less than 0.6 million parameters.

Layer (type)	Output Shape	Param #
input_135 (InputLayer)	[(None, 32, 32, 3)]	0
tf.math.truediv_67 (TFOpLambda)	(None, 32, 32, 3)	0
tf.math.subtract_67 (TFOpLambda)	(None, 32, 32, 3)	0
model_134 (Functional)	(None, 2, 2, 88)	469056
conv2d_67 (Conv2D)	(None, 2, 2, 1000)	89000
batch_normalization_67 (BatchNormalization)	(None, 2, 2, 1000)	4000
re_lu_67 (ReLU)	(None, 2, 2, 1000)	0
global_average_pooling2d_135 (GlobalAveragePooling2D)	(None, 1000)	0
dropout_67 (Dropout)	(None, 1000)	0
dense_67 (Dense)	(None, 10)	10010

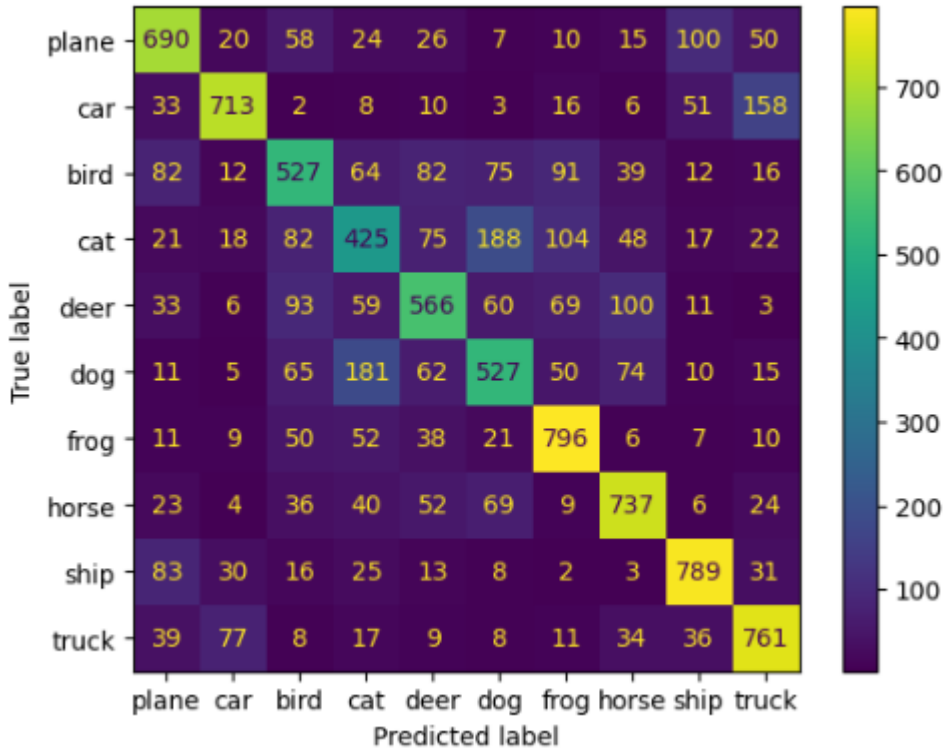
Total params: 572066 (2.18 MB)
Trainable params: 555202 (2.12 MB)
Non-trainable params: 16864 (65.88 KB)

With smaller model size, it only cost 1419 milliseconds per test image on CPU around. Applied on the test data, it achieves 0.653 accuracy overall. Detailed classification report is as below,

The inference time of the model in milliseconds per image per CPU is 1418.926715851

	precision	recall	f1-score	support
plane	0.67	0.69	0.68	1000
car	0.80	0.71	0.75	1000
bird	0.56	0.53	0.54	1000
cat	0.47	0.42	0.45	1000
deer	0.61	0.57	0.59	1000
dog	0.55	0.53	0.54	1000
frog	0.69	0.80	0.74	1000
horse	0.69	0.74	0.71	1000
ship	0.76	0.79	0.77	1000
truck	0.70	0.76	0.73	1000
accuracy			0.65	10000
macro avg	0.65	0.65	0.65	10000
weighted avg	0.65	0.65	0.65	10000

The bird, cat, dog, deer categories are still confused with each other since we haven't done anything to improve them in task2.



Future work

To find the best hyper parameters, it is best to use cross validation. In my experiment, I only split 20% of training data to fix validation data. The hyper parameter chosen is prone to overfit the validation dataset. In the future, we can use the cross validation which takes more time and computation for hyper parameter tuning but achieves best result.

Instead of removing bottlenecks, we could adjust the stride to 1 for the first normal conv2d to keep dimensions. Or even we could up sample the original image size by transposed convolution. Unfortunately, I did not have enough time to implement this idea.

This idea of changing strides can be implemented using the resources as below,

https://github.com/chenhang98/mobileNet-v2_cifar10/tree/master

https://stackoverflow.com/questions/67176547/how-to-remove-first-n-layers-from-a-keras-model/67180315#67180315

https://stackoverflow.com/questions/67453434/replace-stride-layers-in-mobilenet-application-in-keras

Then we tune the hyper parameter alpha - width multiplier to control the model complexity for accuracy and performance trade-off. Also, mobileNetV3small is very promising as well.

We could also use variational auto encoder to find the features represented in latent space of these images, then input the low dimensional features for training. In this way, those confusing categories – like dog, cat - may be identified better by the model.

Reference

Howard, Andrew G, et al. "Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications." *arXiv preprint arXiv:1704.04861* (2017). Print.

Mobilenetv2: Inverted Residuals and Linear Bottlenecks. Proceedings of the IEEE conference on computer vision and pattern recognition. 2018. Print.

Tieleman, Tijmen, and Geoffrey Hinton. "Rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude. Coursera: Neural Networks for Machine Learning." *COURSERA Neural Networks Mach. Learn* 17 (2012). Print.