Assignment 2

Team 8: Manoj Karru, Nishant Gandhi, Emily Strong
INFO 7374 Spring 2019

GitHub:
https://github.com/erstrong/DeepNeuralNetworksandAI/tree/master/Assignment2/part2

Summary Table

Experiment	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
1: Keras Best Model	2.3651	0.432	2.6525	0.3916
2: Autokeras Best Model	10.9597	0.4212	n/a	0.4124
3: AlexNet	5.1745	0.0135	5.1179	0.0150
4: MobileNetV2	2.33	0.76	2.85	0.353
5: Transfer Learning	n/a	n/a	2.304441494	0.0921
6: Transfer Learning Fine Tuning	0.6798	0.7668	0.7480	0.7628

Experiment 1

Colab notebook:
https://drive.google.com/file/d/1Spp451nPJbFc3wsXUMdi3RQNEBK_s82B/view?usp=sharing

For this experiment our data set was the imagenet 200 set which we scaled to 32x32. For selecting a model, we started with the CIFAR10 CNN architecture used in assignment 1. This had a 2.9% validation accuracy and in testing each of the optimizers available the highest validation accuracy we were able to get was 18%. We concluded that the architecture was too simple for 200 classes. We then found an architecture designed for the CIFAR100. The default settings had a 23% accuracy, and we tuned the model to achieve 39% validation accuracy in 50 epochs with a batch size of 32 and using image augmentation. We found the best optimizer was SGD with a learning rate of 0.01, and we added batch normalization. The full scores for the training and validation data for the best model are in the summary table. Our final architecture is shown in Figure 1. Plots comparing the starting and final architecture performance are in Figure 2.

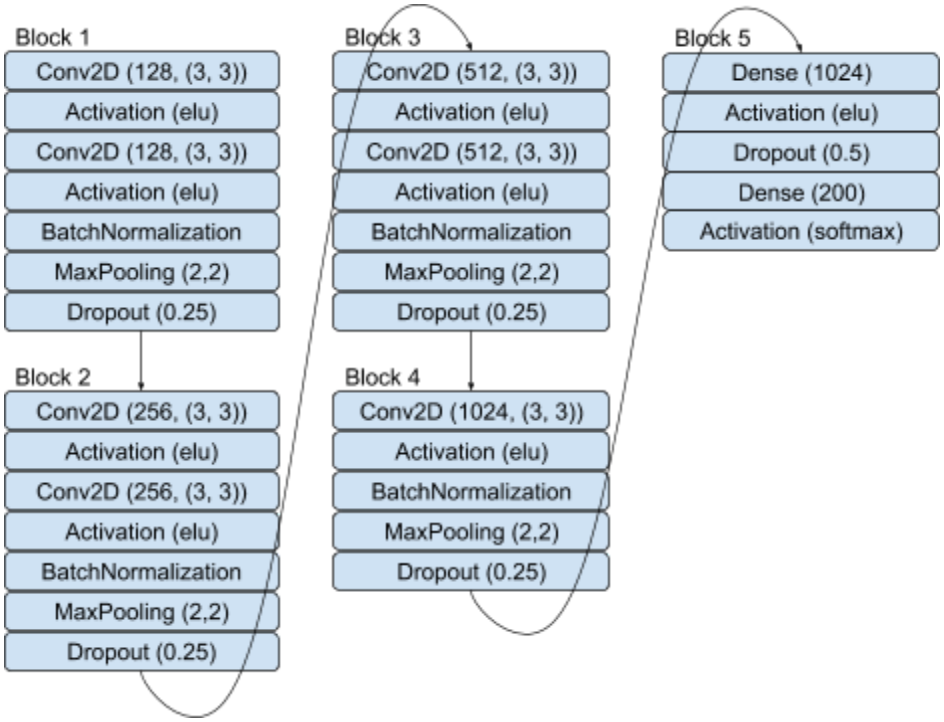


Figure 1: Final Architecture for Imagenet-200

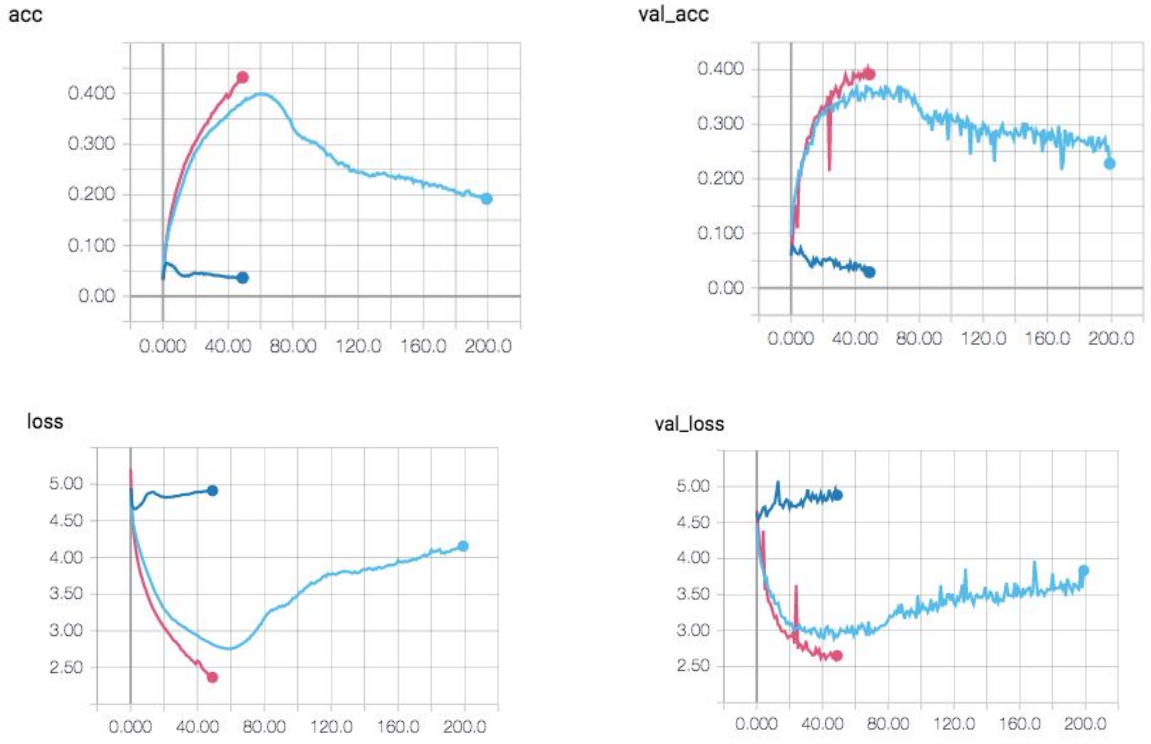


Figure 2: Train and validation loss and accuracy of the models. Pink is the final model, dark blue is the starting CIFAR10 model, light blue is the starting CIFAR100 model.

Experiment 2

Colab notebook:
https://drive.google.com/file/d/1dIQXMsdlpAQN9BknlDVJvOrBoZ90qE9W/view?usp=sharing

For this experiment we used the 32x32 imagenet 200 set with autokeras. Autokeras uses efficient neural architecture search by using Bayesian optimization to change the architecture over a series of training runs. The process works by morphing the architecture through a balance of exploration and exploitation of previous successful architecture, training the model, generating a computational graph and updating the underlying gaussian process with the graph. The model generated from each run gets saved in a model storage pool and at the end of the designated training duration, the best one is selected.

We ran autokeras twice, once for 12 hours and once for 24 hours. In the 12 hour run the library tested 63 models, with the best model having a validation accuracy of 31%. In the 24 hour run the library tested 42 models, with the best model having a validation accuracy of 41%, a 2% improvement over our manually tuned model from experiment 1. However the model had a training loss higher than 10, whereas the manual model was 2.4. Autokeras does not calculate validation loss, so we cannot compare the models on that metric. The best model has 27 convolutional layers with only one dense layer as the output layer. Interestingly, it does not use pooling and instead has 17 batch normalization layers.

Final architecture:

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 32, 32, 3)	0	
conv2d_1 (Conv2D)	(None, 32, 32, 64)	1792	input_1[0][0]
batch_normalization_1	(BatchNor (None, 32, 32, 64)	256	conv2d_1[0][0]
batch_normalization_2	(BatchNor (None, 32, 32, 64)	256	batch_normalization_1[0][0]
activation_25 (Activation)	(None, 32, 32, 64)	0	batch_normalization_2[0][0]
conv2d_26 (Conv2D)	(None, 32, 32, 64)	4160	activation_25[0][0]
activation_1 (Activation)	(None, 32, 32, 64)	0	conv2d_26[0][0]
conv2d_2 (Conv2D)	(None, 32, 32, 64)	36928	activation_1[0][0]
batch_normalization_3	(BatchNor (None, 32, 32, 64)	256	conv2d_2[0][0]
activation_2 (Activation)	(None, 32, 32, 64)	0	batch_normalization_3[0][0]
conv2d_27 (Conv2D)	(None, 32, 32, 64)	4160	activation_2[0][0]

activation_3 (Activation)	(None, 32, 32, 64)	0	activation_1[0][0]
conv2d_3 (Conv2D)	(None, 32, 32, 64)	36928	conv2d_27[0][0]
conv2d_4 (Conv2D)	(None, 32, 32, 64)	4160	activation_3[0][0]
add_1 (Add)	(None, 32, 32, 64)	0	conv2d_3[0][0] conv2d_4[0][0]
batch_normalization_4	(BatchNor (None, 32, 32, 64)	256	add_1[0][0]
activation_4 (Activation)	(None, 32, 32, 64)	0	batch_normalization_4[0][0]
conv2d_5 (Conv2D)	(None, 32, 32, 64)	36928	activation_4[0][0]
batch_normalization_5	(BatchNor (None, 32, 32, 64)	256	conv2d_5[0][0]
activation_5 (Activation)	(None, 32, 32, 64)	0	batch_normalization_5[0][0]
activation_6 (Activation)	(None, 32, 32, 64)	0	activation_4[0][0]
conv2d_6 (Conv2D)	(None, 32, 32, 64)	36928	activation_5[0][0]
conv2d_7 (Conv2D)	(None, 32, 32, 64)	4160	activation_6[0][0]
add_2 (Add)	(None, 32, 32, 64)	0	conv2d_6[0][0] conv2d_7[0][0]
batch_normalization_6	(BatchNor (None, 32, 32, 64)	256	add_2[0][0]
activation_7 (Activation)	(None, 32, 32, 64)	0	batch_normalization_6[0][0]
conv2d_8 (Conv2D)	(None, 32, 32, 128)	73856	activation_7[0][0]
batch_normalization_7	(BatchNor (None, 32, 32, 128)	512	conv2d_8[0][0]
activation_8 (Activation)	(None, 32, 32, 128)	0	batch_normalization_7[0][0]
activation_9 (Activation)	(None, 32, 32, 64)	0	activation_7[0][0]
conv2d_9 (Conv2D)	(None, 32, 32, 2048)	2361344	activation_8[0][0]
conv2d_10 (Conv2D)	(None, 32, 32, 2048)	133120	activation_9[0][0]
add_3 (Add)	(None, 32, 32, 2048)	0	conv2d_9[0][0] conv2d_10[0][0]
batch_normalization_8	(BatchNor (None, 32, 32, 2048)	8192	add_3[0][0]
activation_10 (Activation)	(None, 32, 32, 2048)	0	batch_normalization_8[0][0]
conv2d_11 (Conv2D)	(None, 32, 32, 128)	2359424	activation_10[0][0]
batch_normalization_9	(BatchNor (None, 32, 32, 128)	512	conv2d_11[0][0]
activation_11 (Activation)	(None, 32, 32, 128)	0	batch_normalization_9[0][0]

activation_12 (Activation)	(None, 32, 32, 2048)	0	activation_10[0][0]
conv2d_12 (Conv2D)	(None, 32, 32, 128)	147584	activation_11[0][0]
conv2d_13 (Conv2D)	(None, 32, 32, 128)	262272	activation_12[0][0]
add_4 (Add)	(None, 32, 32, 128)	0	conv2d_12[0][0] conv2d_13[0][0]
batch_normalization_10	(BatchNo (None, 32, 32, 128)	512	add_4[0][0]
activation_13 (Activation)	(None, 32, 32, 128)	0	batch_normalization_10[0][0]
conv2d_14 (Conv2D)	(None, 32, 32, 256)	295168	activation_13[0][0]
batch_normalization_11	(BatchNo (None, 32, 32, 256)	1024	conv2d_14[0][0]
activation_14 (Activation)	(None, 32, 32, 256)	0	batch_normalization_11[0][0]
activation_15 (Activation)	(None, 32, 32, 128)	0	activation_13[0][0]
conv2d_15 (Conv2D)	(None, 32, 32, 256)	590080	activation_14[0][0]
conv2d_16 (Conv2D)	(None, 32, 32, 256)	33024	activation_15[0][0]
add_5 (Add)	(None, 32, 32, 256)	0	conv2d_15[0][0] conv2d_16[0][0]
batch_normalization_12	(BatchNo (None, 32, 32, 256)	1024	add_5[0][0]
activation_16 (Activation)	(None, 32, 32, 256)	0	batch_normalization_12[0][0]
conv2d_17 (Conv2D)	(None, 32, 32, 256)	590080	activation_16[0][0]
batch_normalization_13	(BatchNo (None, 32, 32, 256)	1024	conv2d_17[0][0]
activation_17 (Activation)	(None, 32, 32, 256)	0	batch_normalization_13[0][0]
activation_18 (Activation)	(None, 32, 32, 256)	0	activation_16[0][0]
conv2d_18 (Conv2D)	(None, 32, 32, 256)	590080	activation_17[0][0]
conv2d_19 (Conv2D)	(None, 32, 32, 256)	65792	activation_18[0][0]
add_6 (Add)	(None, 32, 32, 256)	0	conv2d_18[0][0] conv2d_19[0][0]
batch_normalization_14	(BatchNo (None, 32, 32, 256)	1024	add_6[0][0]
activation_19 (Activation)	(None, 32, 32, 256)	0	batch_normalization_14[0][0]
conv2d_20 (Conv2D)	(None, 32, 32, 512)	1180160	activation_19[0][0]
batch_normalization_15	(BatchNo (None, 32, 32, 512)	2048	conv2d_20[0][0]
activation_20 (Activation)	(None, 32, 32, 512)	0	batch_normalization_15[0][0]

activation_21 (Activation)	(None, 32, 32, 256)	0	activation_19[0][0]
conv2d_21 (Conv2D)	(None, 32, 32, 512)	2359808	activation_20[0][0]
conv2d_22 (Conv2D)	(None, 32, 32, 512)	131584	activation_21[0][0]
add_7 (Add)	(None, 32, 32, 512)	0	conv2d_21[0][0] conv2d_22[0][0]
batch_normalization_16	(BatchNo (None, 32, 32, 512)	2048	add_7[0][0]
activation_22 (Activation)	(None, 32, 32, 512)	0	batch_normalization_16[0][0]
conv2d_23 (Conv2D)	(None, 32, 32, 512)	2359808	activation_22[0][0]
batch_normalization_17	(BatchNo (None, 32, 32, 512)	2048	conv2d_23[0][0]
activation_23 (Activation)	(None, 32, 32, 512)	0	batch_normalization_17[0][0]
activation_24 (Activation)	(None, 32, 32, 512)	0	activation_22[0][0]
conv2d_24 (Conv2D)	(None, 32, 32, 512)	2359808	activation_23[0][0]
conv2d_25 (Conv2D)	(None, 32, 32, 512)	262656	activation_24[0][0]
add_8 (Add)	(None, 32, 32, 512)	0	conv2d_24[0][0] conv2d_25[0][0]
global_average_pooling2d_1	(Glo (None, 512)	0	add_8[0][0]
dense_1 (Dense)	(None, 200)	102600	global_average_pooling2d_1[0][0]

Total params: 16,445,896
Trainable params: 16,435,144
Non-trainable params: 10,752

Experiment 3

Colab notebook:
https://drive.google.com/file/d/1r8Fx23si6wtpNPi_VhDkTOhuH0BdRzs4/view?usp=sharing

For this experiment, we used 32x32 imagenet 200 set. We referenced the architecture of AlexNet being shared by professor. We changed the input layer to accommodate 32x32 size image which different from AlexNet's 227x227 input size. We considered change image dimension to match AlexNet but we did not use it with concern about missing the pattern in image as well as computation resource constraints. The AlexNet has very high number of tunable parameters, which means more data and more epochs required to reduce validation loss.

Using ‘adam’ optimizer was not providing good performance and switching to ‘sgd’ was significant in observing the model performance improvement on validation loss.

Experiment 4

Colab notebook:
https://colab.research.google.com/drive/1AgjwfxHyE9pe7OHO5zya1GNonHICkxk6

MobileNetV2: Inverted residuals and Linear Bottlenecks

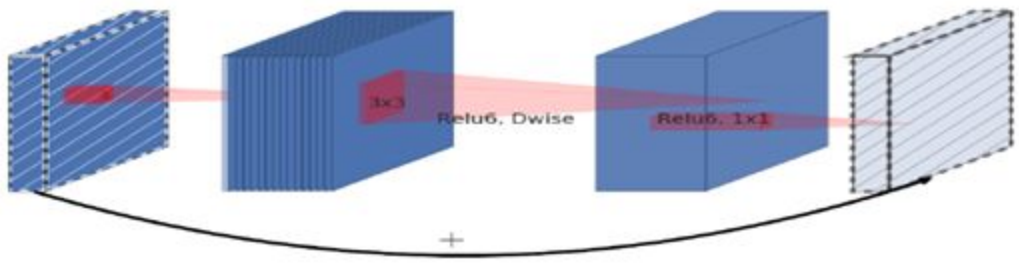
This Neural Network architecture that was optimized for mobile devices. This model strives to get high accuracy while keeping the parameters and mathematical operations as low as possible.

The architecture dubbed MobileNet revolves around the idea of using depth-wise separable convolutions, which consist of a depthwise and a pointwise convolution after one another. MobileNetV2 extends its predecessor with 2 main ideas. 1) Inverted Residuals 2) Linear Bottlenecks.

Residuals.
In keras residual approach looks like this:

```
def residual_block(x, squeeze=16, expand=64):
  m = Conv2D(squeeze, (1,1), activation='relu')(x)
  m = Conv2D(squeeze, (3,3), activation='relu')(m)
  m = Conv2D(expand, (1,1), activation='relu')(m)
  return Add()([m, x])
```

Inverted Residuals
MobileNetV2 follows a narrow->wide->narrow approach. The first step widens the network using a 1x1 convolution because the following 3x3 depthwise convolution already greatly reduces the number of parameters. Afterwards another 1x1 convolution squeezes the network in order to match the initial number of channels.



An inverted residual block connects narrow layers with a skip connection while layers in between are wide

In Keras it would look like this:

```
def inverted_residual_block(x, expand=64, squeeze=16):
  m = Conv2D(expand, (1,1), activation='relu')(x)
  m = DepthwiseConv2D((3,3), activation='relu')(m)
  m = Conv2D(squeeze, (1,1), activation='relu')(m)
  return Add()([m, x])
```

Linear Bottleneck

The reason we use non-linear activation functions in neural networks is that multiple matrix multiplications cannot be reduced to a single numerical operation. It allows us to build neural networks that have multiple layers. At the same time the activation function ReLU, which is commonly used in neural networks, discards values that are smaller than 0. This loss of information can be tackled by increasing the number of channels in order to increase the capacity of the network.

With inverted residual blocks we do the opposite and squeeze the layers where the skip connections are linked. This hurts the performance of the network. The authors introduced the idea of a linear bottleneck where the last convolution of a residual block has a linear output before it's added to the initial activations. Putting this into code is super simple as we simply discard the last activation function of the convolutional block:

```
def inverted_linear_residual_block(x, expand=64, squeeze=16):
  m = Conv2D(expand, (1,1), activation='relu')(x)
  m = DepthwiseConv2D((3,3),  activation='relu')(m)
  m = Conv2D(squeeze, (1,1))(m)
  return Add()([m, x])
```

Using Relu6 as Activation Function:

The snippet above shows the structure of a convolutional block that incorporates inverted residuals and linear bottlenecks. If you want to match MobileNetV2 as closely as possible there are two other pieces you need. The first aspect simply adds Batch Normalization behind every convolutional layer as you're probably used to by now anyways.
The second addition is not quite as common. The authors use ReLU6 instead of ReLU, which limits the value of activations to a maximum of 6. The activation is linear as long as it's between 0 and 6.

```
def relu(x):
  return max(0, x)

def relu6(x):
  return min(max(0, x), 6)
```

The final building block looks like this:

```
def bottleneck_block(x, expand=64, squeeze=16):
  m = Conv2D(expand, (1,1))(x)
  m = BatchNormalization()(m)
  m = Activation('relu6')(m)
  m = DepthwiseConv2D((3,3))(m)
  m = BatchNormalization()(m)
  m = Activation('relu6')(m)
  m = Conv2D(squeeze, (1,1))(m)
  m = BatchNormalization()(m)
  return Add()([m, x])
```

Final Architecture:

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 MobileNetV2 architecture

We understand the building block of MobileNetV2 we can take a look at the entire architecture. In the table you can see how the bottleneck blocks are arranged. t stands for expansion rate of the channels. As you can see they used a factor of 6 opposed to the 4 in our example. c

represents the number of input channels and n how often the block is repeated. Lastly s tells us whether the first repetition of a block used a stride of 2 for the downsampling process. All in all it's a very simple and common assembly of convolutional blocks.

Results of MobilenetV2

MobileNetV2 model accepts one of the following formats: *(96, 96), (128, 128), (160, 160),(192, 192), or (224, 224).* In addition, the image has to be 3 channel (RGB) format. Therefore, We need to resize & convert our images. from (32 X 32 X 3) to (96 X 96 X 3).
Since converting 32 X 32 image to 96 X 96 consume lot of memory. So we assign 32 X 32 to Input Tensor.

Example code:

```
input_tensor = Input(shape=(32, 32, 3))
base_model = MobileNetV2( include_top=False, weights='imagenet',
input_tensor=input_tensor,pooling='avg')
#for layer in base_model.layers:
 #        layer.trainable = True
x = Dense(4096, activation='relu')(base_model.output)
x = Dropout(0.5)(x)
x = Dense(4096, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(200, activation = 'softmax')(x)
model = Model(inputs = base_model.input, outputs = x)
model. Summary()
```

Epochs	Batch_size	Drop_out	Train_Accuracy	Test_Accuracy	Train_Loss
10	256	0	39	25	3.42
25	512	0.5	73	35	3.55
25	256	0.4	82	30	3.22

The models are heavily overfitted.

Experiment 5

Colab notebook:
https://drive.google.com/file/d/1dTpUt3TUE-JpnGjOrnv0zijX_5201R9O/view?usp=sharing

For this experiment we used the MobileNetV2 model with the CIFAR10 data in transfer learning. This is when a pre-trained model is used with a new data set. MobileNetV2 does not have a weights available for an architecture compatible with 32x32 images, so we had to add an input layer to use the data set. The model had very poor performance, with only 9% accuracy as shown in the summary table.

Experiment 6

Colab notebook:
https://drive.google.com/file/d/1dTpUt3TUE-JpnGjOrnv0zijX_5201R9O/view?usp=sharing

For this experiment we fine tuned the MobileNetV2 model to work with the CIFAR10 data. We first tried tuning only the dense layer we added to connect the final convolutional layer to our output layer. This did not give any improvement in performance. Next we tuned the final convolution block, which yielded a 13% validation accuracy in 9 epochs. We then set all of the layers to be trainable and after 18 epochs achieved 77% validation accuracy as shown in the table below and Figure 3. The difference in loss indicates there is some overfitting occurring, but the difference in accuracy between train and validation is only 0.004. The fact that we had to tune all of the layers is likely because the images are smaller than the ones the original weights were trained from, so even the early layers that are detecting edges did not have appropriate weights.

Experiment	**Train Loss**	**Train Accuracy**	**Validation Loss**	**Validation Accuracy**
Dense layer	1.9695	0.2987	2.3012	0.0981
Final block	1.611	0.4306	2.9861	0.1268
All layers	0.6798	0.7668	0.748	0.7628

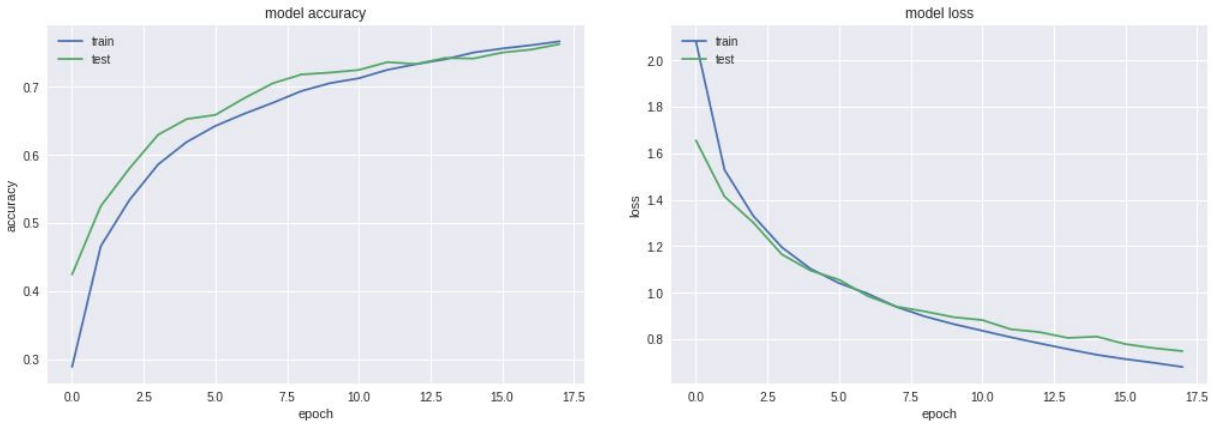


Figure 3: Accuracy and loss curves for tuning MobileNetV2 to CIFAR10