Part 1:
Build the best network you can using Keras Summarize Accuracy measures for Train, Test and Validation for the best model

Architecture:

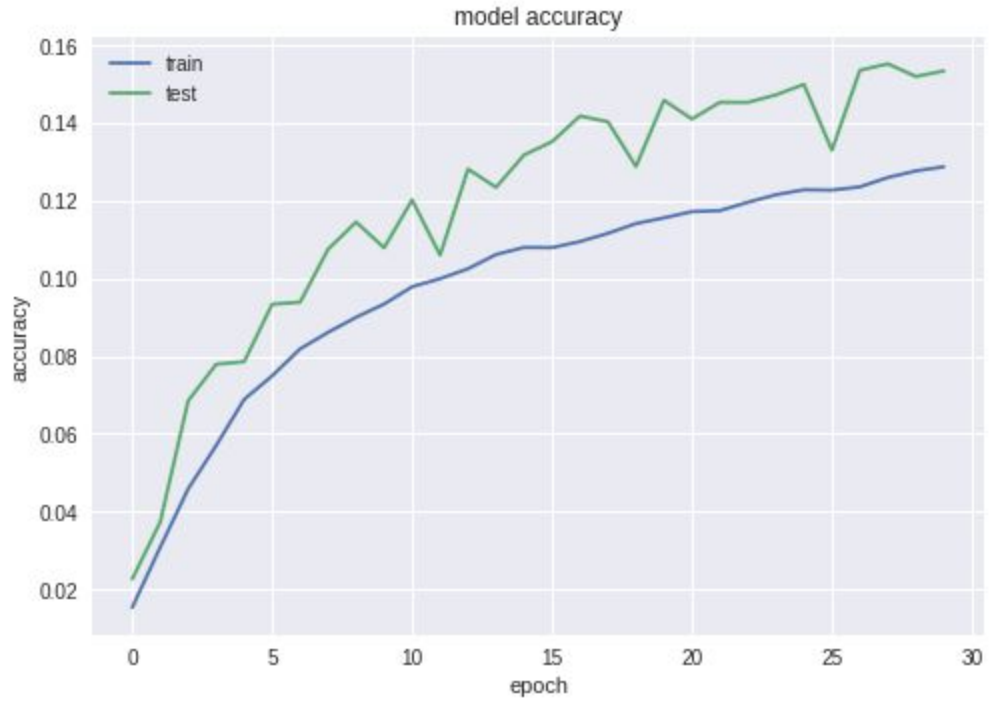
Layer (type)	Output Shape	Param #
conv2d_19 (Conv2D)	(None, 32, 32, 16)	448
activation_25 (Activation)	(None, 32, 32, 16)	0
batch_normalization_19 (Batc	(None, 32, 32, 16)	64
conv2d_20 (Conv2D)	(None, 30, 30, 16)	2320
activation_26 (Activation)	(None, 30, 30, 16)	0
batch_normalization_20 (Batc	(None, 30, 30, 16)	64
max_pooling2d_10 (MaxPooling	(None, 15, 15, 16)	0
dropout_13 (Dropout)	(None, 15, 15, 16)	0
conv2d_21 (Conv2D)	(None, 15, 15, 32)	4640
activation_27 (Activation)	(None, 15, 15, 32)	0
batch_normalization_21 (Batc	(None, 15, 15, 32)	128
conv2d_22 (Conv2D)	(None, 13, 13, 32)	9248
activation_28 (Activation)	(None, 13, 13, 32)	0
batch_normalization_22 (Batc	(None, 13, 13, 32)	128
max_pooling2d_11 (MaxPooling	(None, 6, 6, 32)	0
dropout_14 (Dropout)	(None, 6, 6, 32)	0
conv2d_23 (Conv2D)	(None, 6, 6, 64)	18496

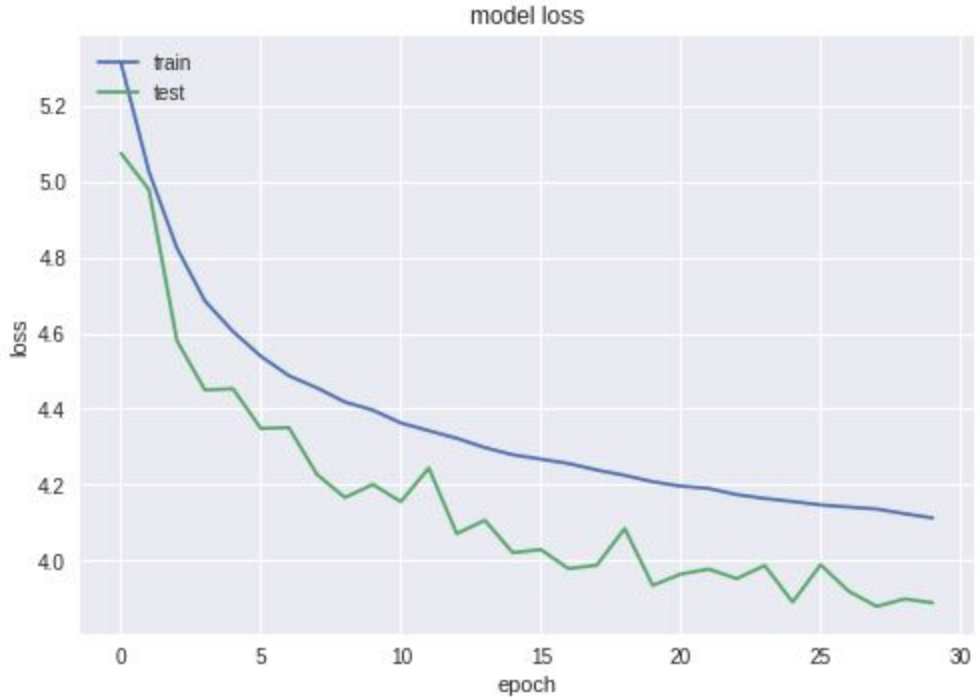
activation_29 (Activation)	(None, 6, 6, 64)	0
batch_normalization_23 (Batc	(None, 6, 6, 64)	256
conv2d_24 (Conv2D)	(None, 4, 4, 64)	36928
activation_30 (Activation)	(None, 4, 4, 64)	0
batch_normalization_24 (Batc	(None, 4, 4, 64)	256
max_pooling2d_12 (MaxPooling	(None, 2, 2, 64)	0
dropout_15 (Dropout)	(None, 2, 2, 64)	0
conv2d_25 (Conv2D)	(None, 2, 2, 64)	36928
activation_31 (Activation)	(None, 2, 2, 64)	0
batch_normalization_25 (Batc	(None, 2, 2, 64)	256
max_pooling2d_13 (MaxPooling	(None, 1, 1, 64)	0
dropout_16 (Dropout)	(None, 1, 1, 64)	0
global_average_pooling2d_4 (	(None, 64)	0
activation_32 (Activation)	(None, 64)	0
dropout_17 (Dropout)	(None, 64)	0
dense_4 (Dense)	(None, 200)	13000
activation_33 (Activation)	(None, 200)	0

This is a general keras model that worked better than the others. We tried differents parameter values and layer structure, though they actually did not make a pretty obvious difference.

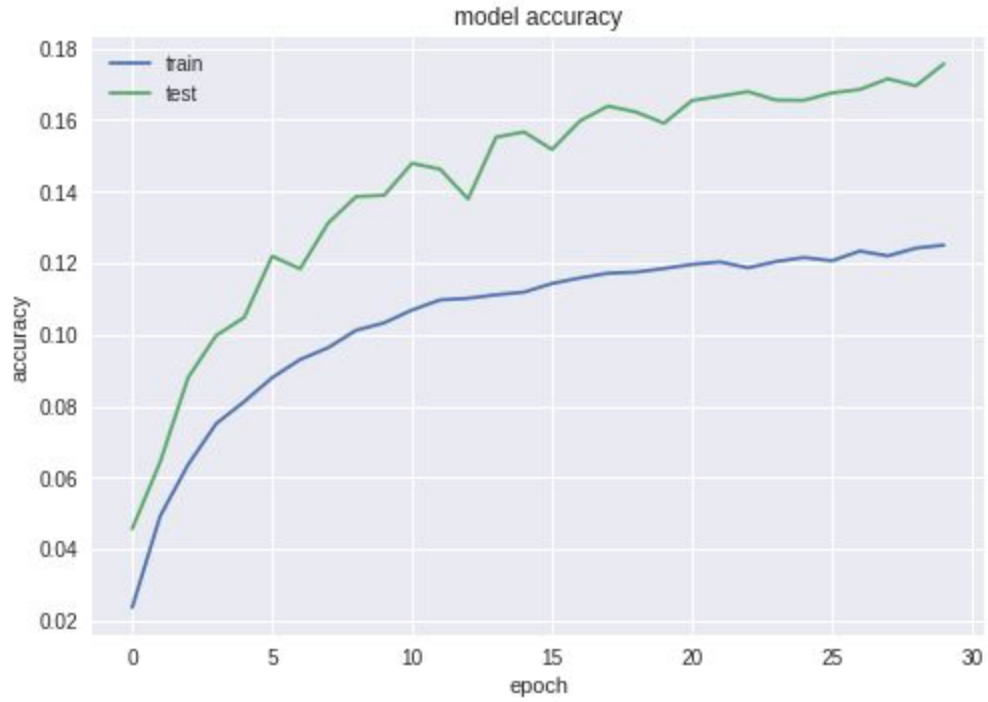
We reformatted the dataset which can be delivered to the model, and set the validation split to 0.3.

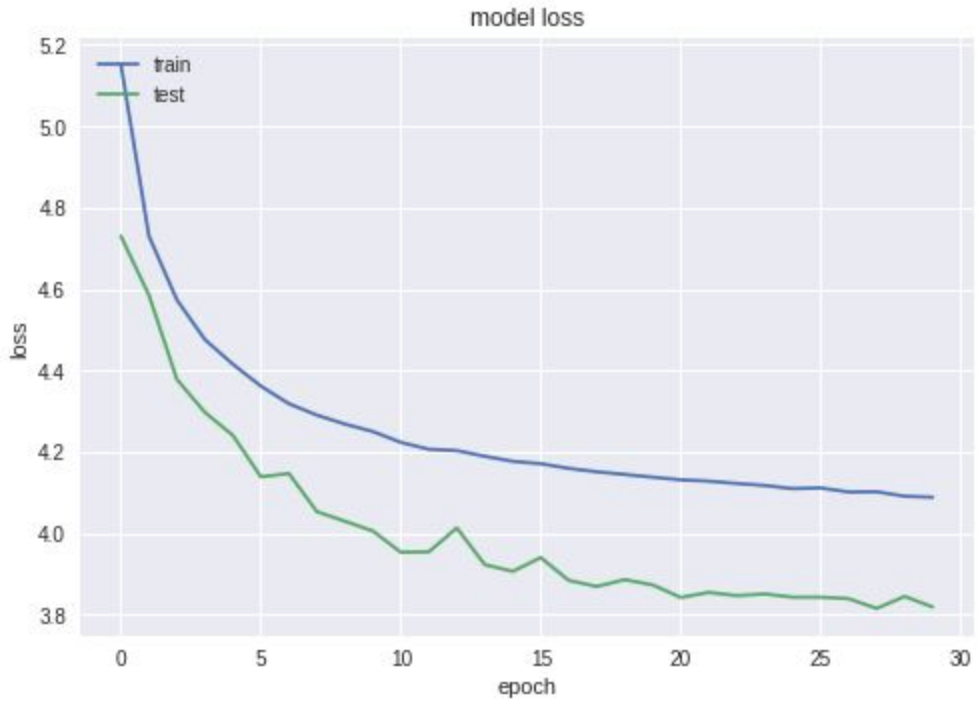
Training Result:
Test loss: 3.8896696652730305
Test accuracy: 0.15346666666666667





Here is another example of our trainings:
Test loss: 3.8188165491739907
Test accuracy: 0.17563333333333334





Though the test accuracy is higher, but this model is too underfitting. So we changed the dropout and adjusted the layers counts and the result got a bit better.

Part 2:
Use Autokeras to tune hyperparameters. Summarize Accuracy measures for Train, Test and Validation for the best model.

Configuration:
We split the validation data as 30% of the whole dataset.
The training_times is 3 hours. The whole training progress spent around 12 hours.

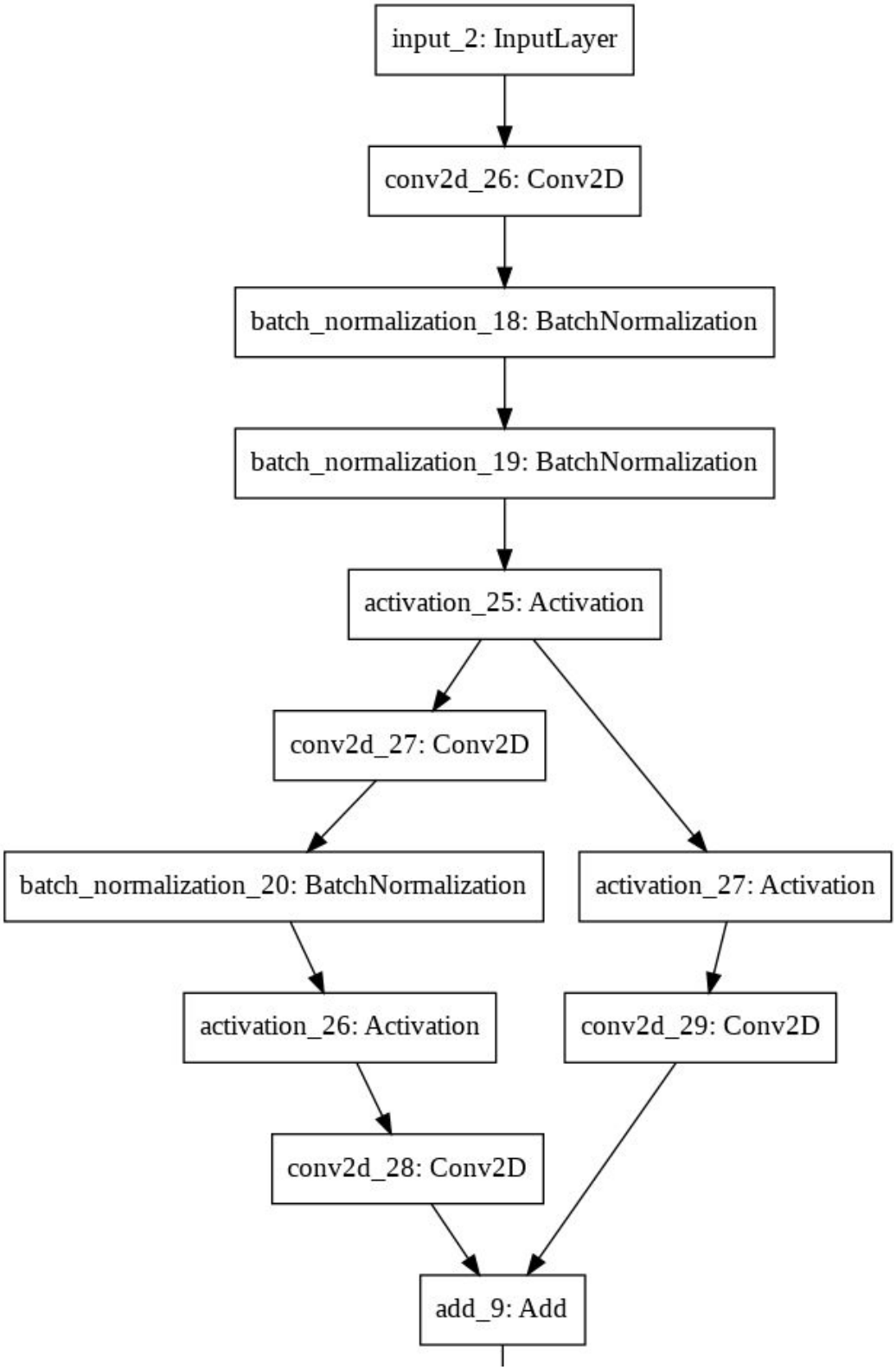
Training Result:
Autokeras trained three models totally and gave us a big model containing 26 conv2D layers.
Here is the Autokeras report:

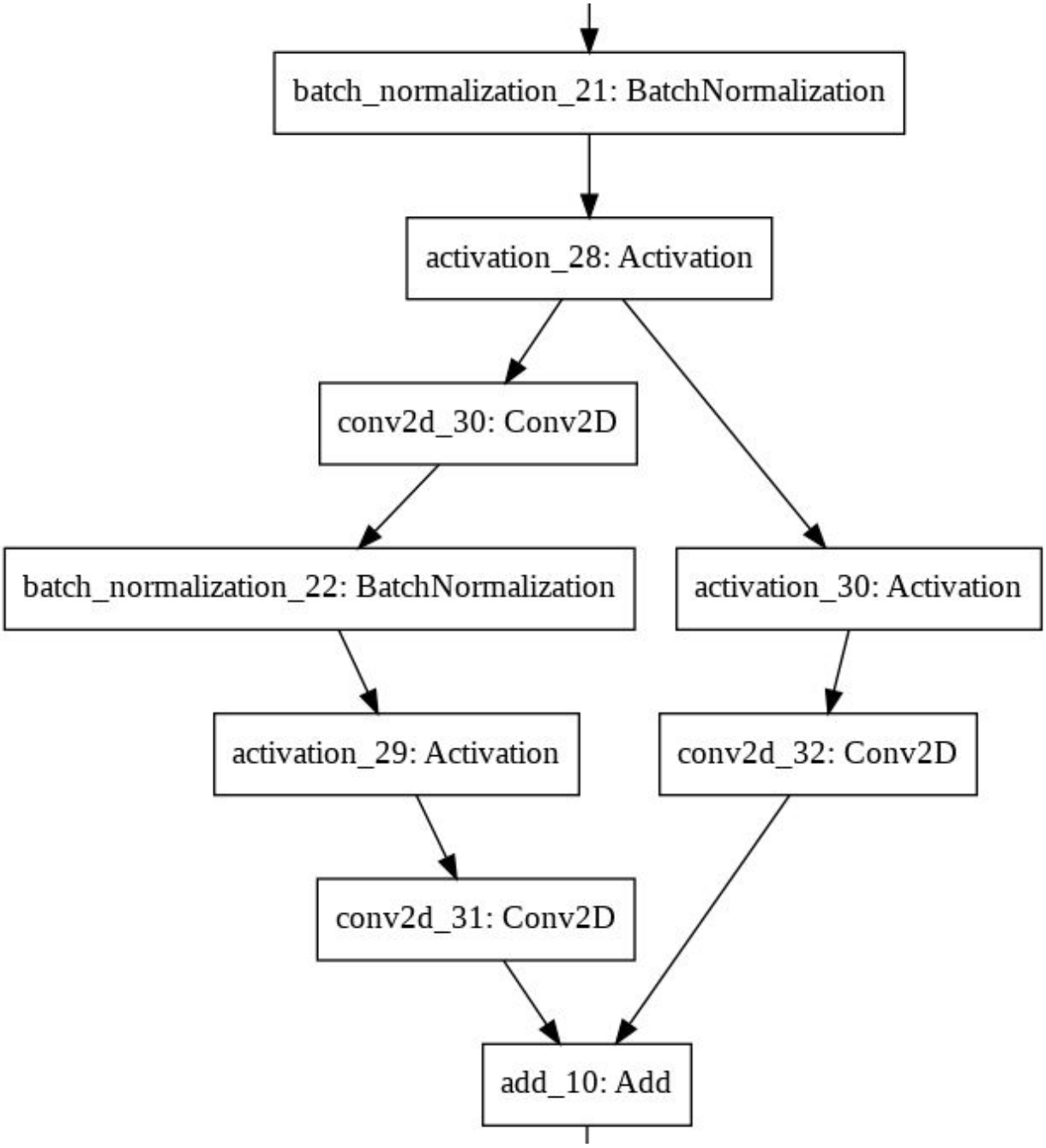
	precision	recall	f1-score	support
micro avg	0.30	0.30	0.30	30000
macro avg	0.34	0.30	0.29	30000
weighted avg	0.34	0.30	0.29	30000

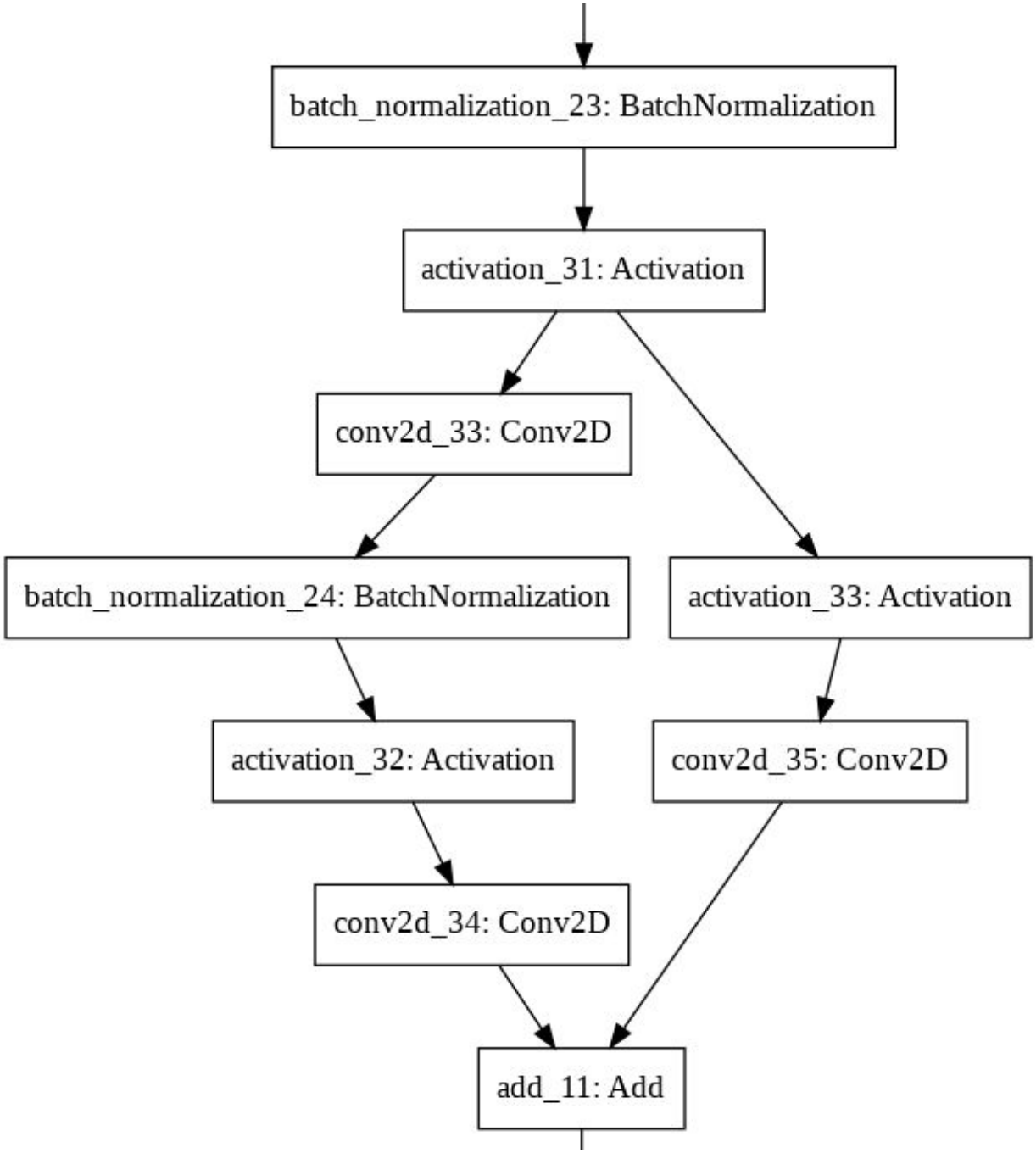
score: 0.30046666666666666

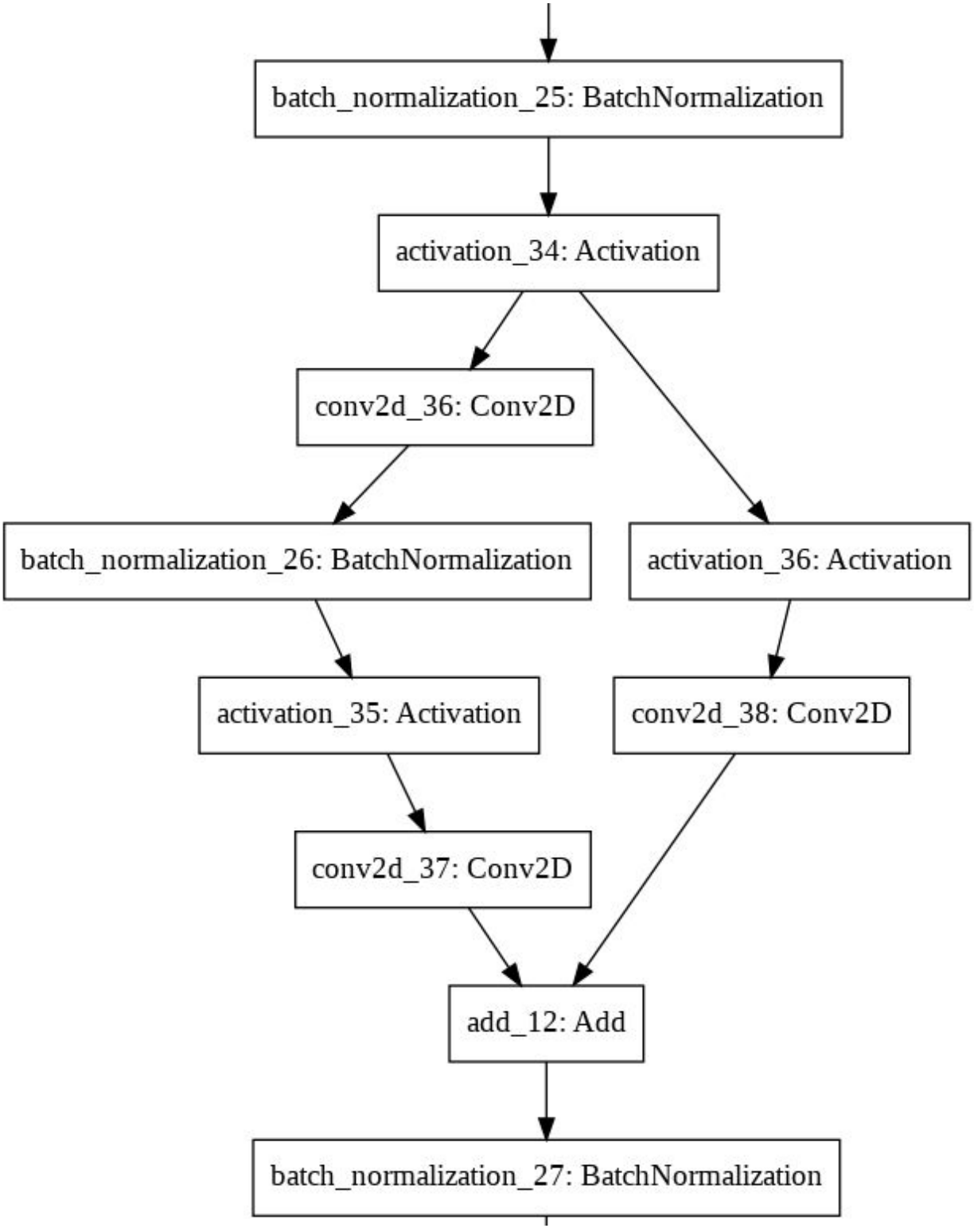
The final score is 0.3, and average accuracy is 0.34.

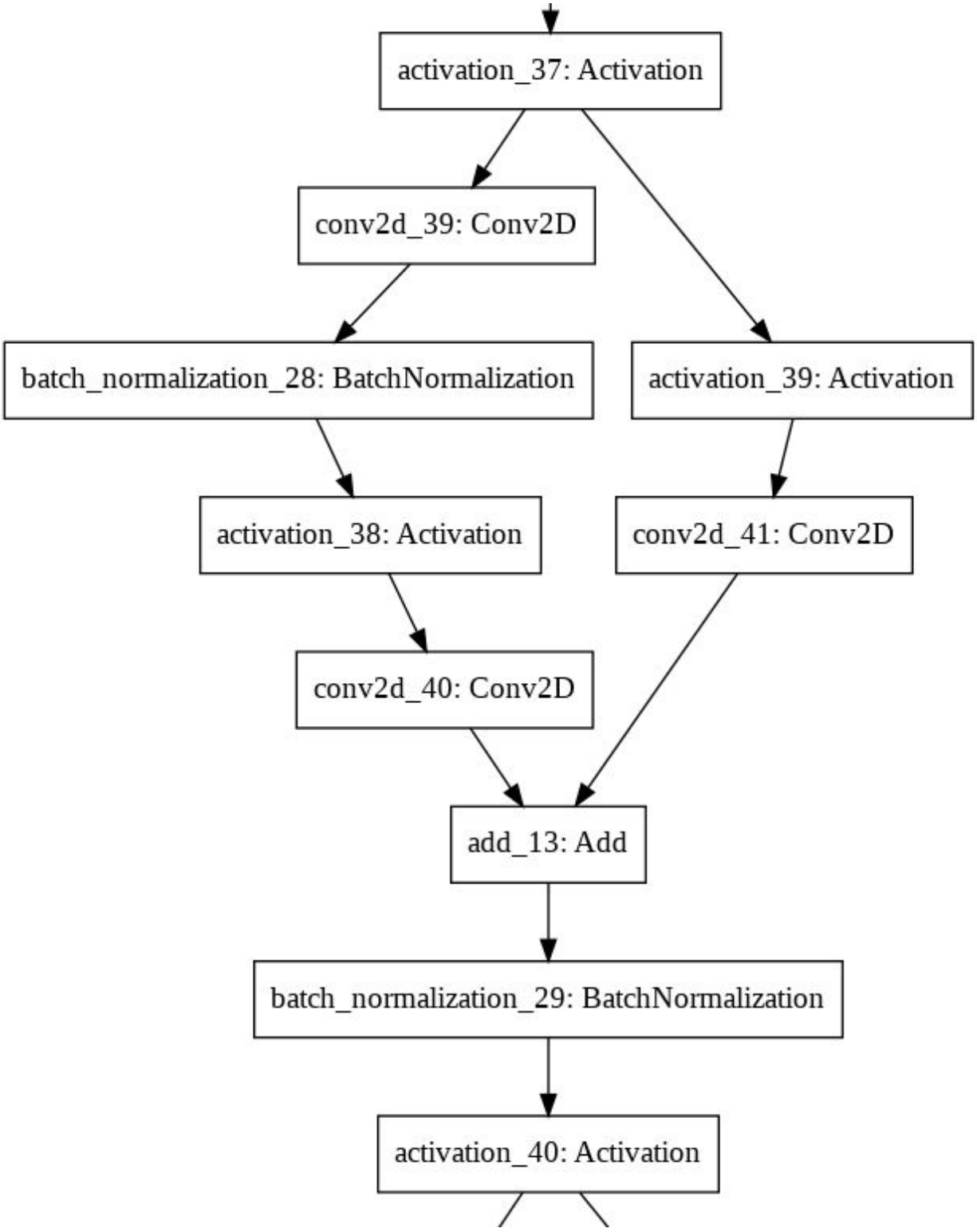
The following is the Autokeras model plot:

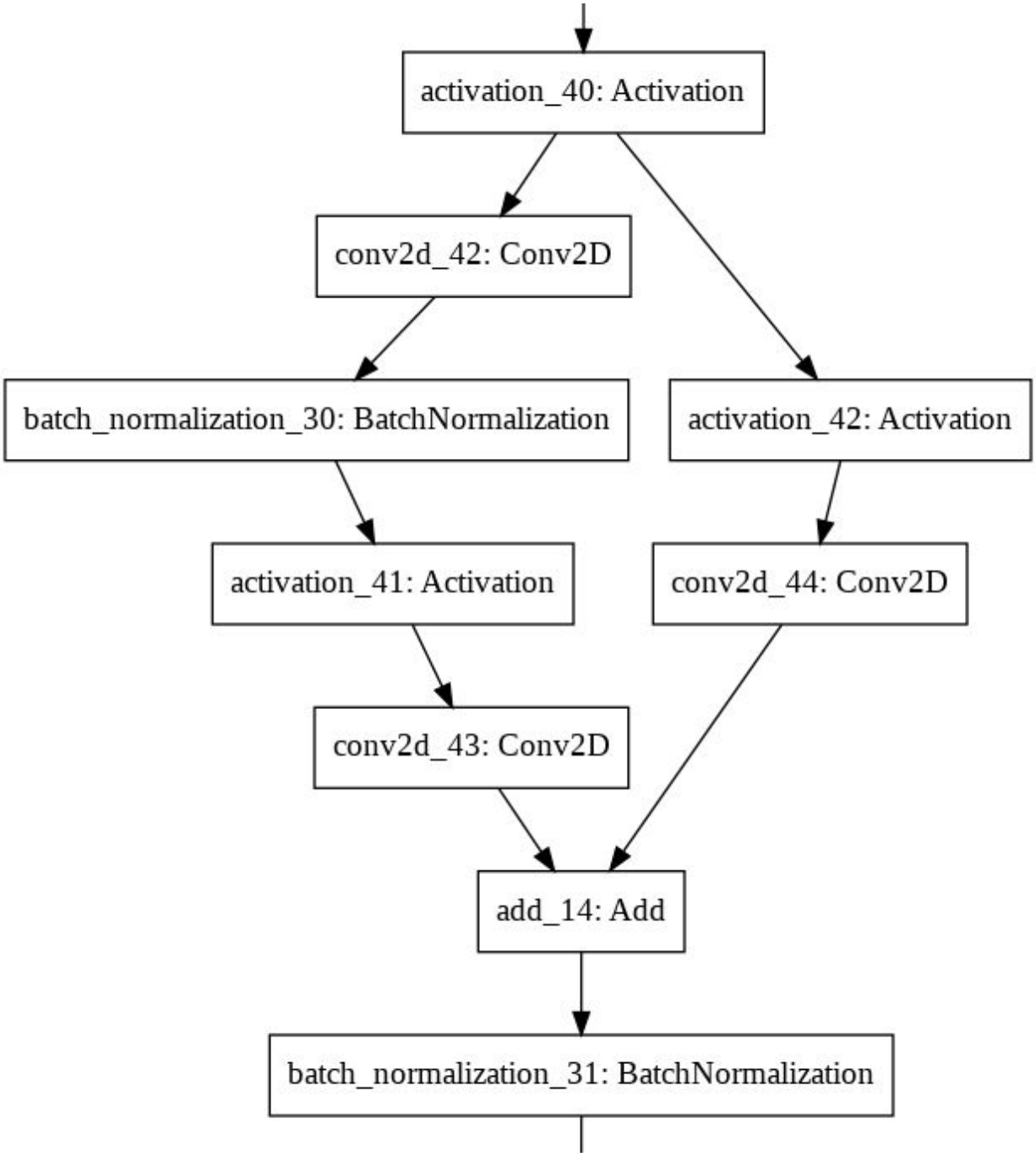

input_2: InputLayer
conv2d_26: Conv2D
batch_normalization_18: BatchNormalization
batch_normalization_19: BatchNormalization
activation_25: Activation
conv2d_27: Conv2D
batch_normalization_20: BatchNormalization
activation_27: Activation
activation_26: Activation
conv2d_29: Conv2D
conv2d_28: Conv2D
add_9: Add

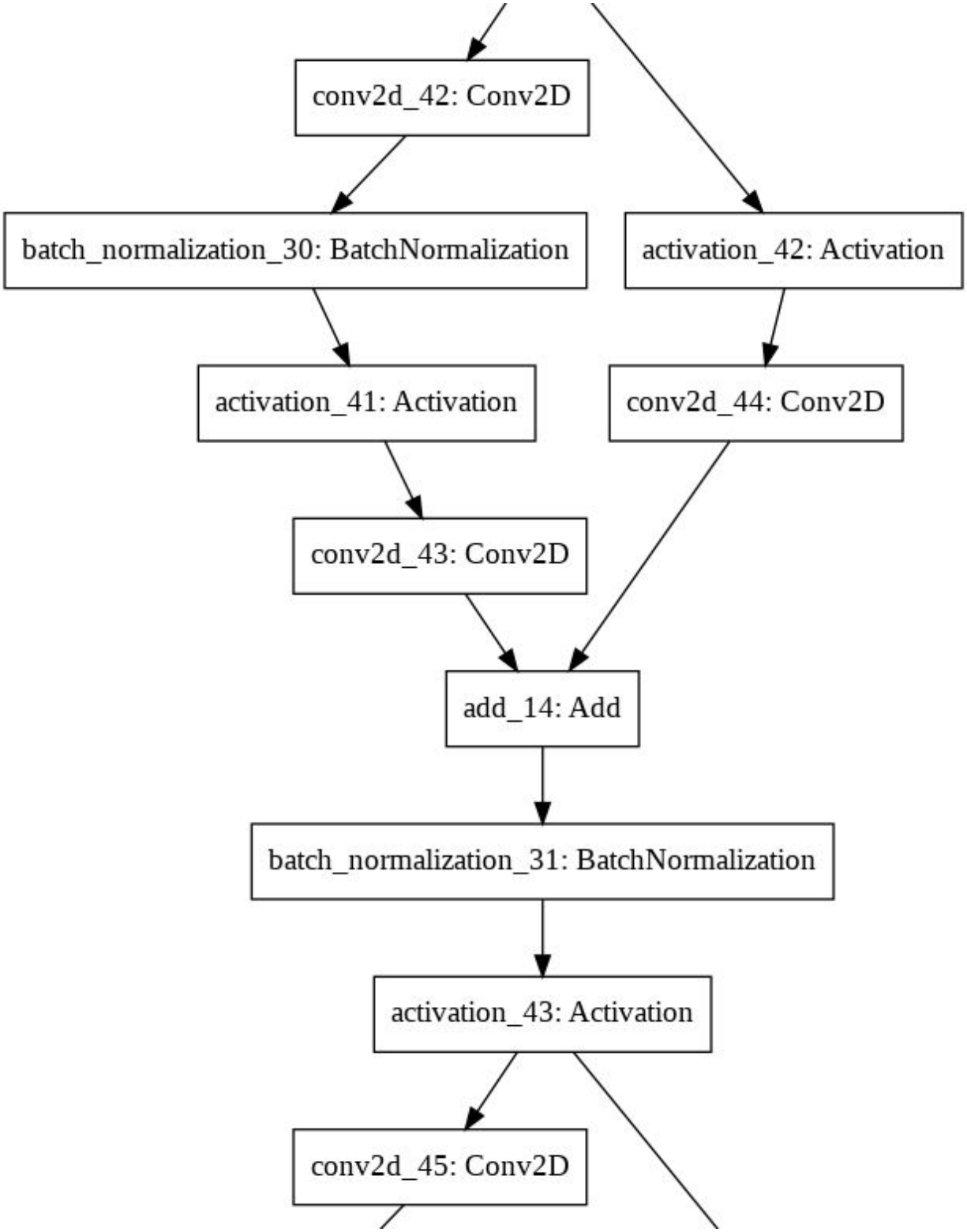

batch_normalization_21: BatchNormalization
activation_28: Activation
conv2d_30: Conv2D
batch_normalization_22: BatchNormalization
activation_30: Activation
activation_29: Activation
conv2d_32: Conv2D
conv2d_31: Conv2D
add_10: Add

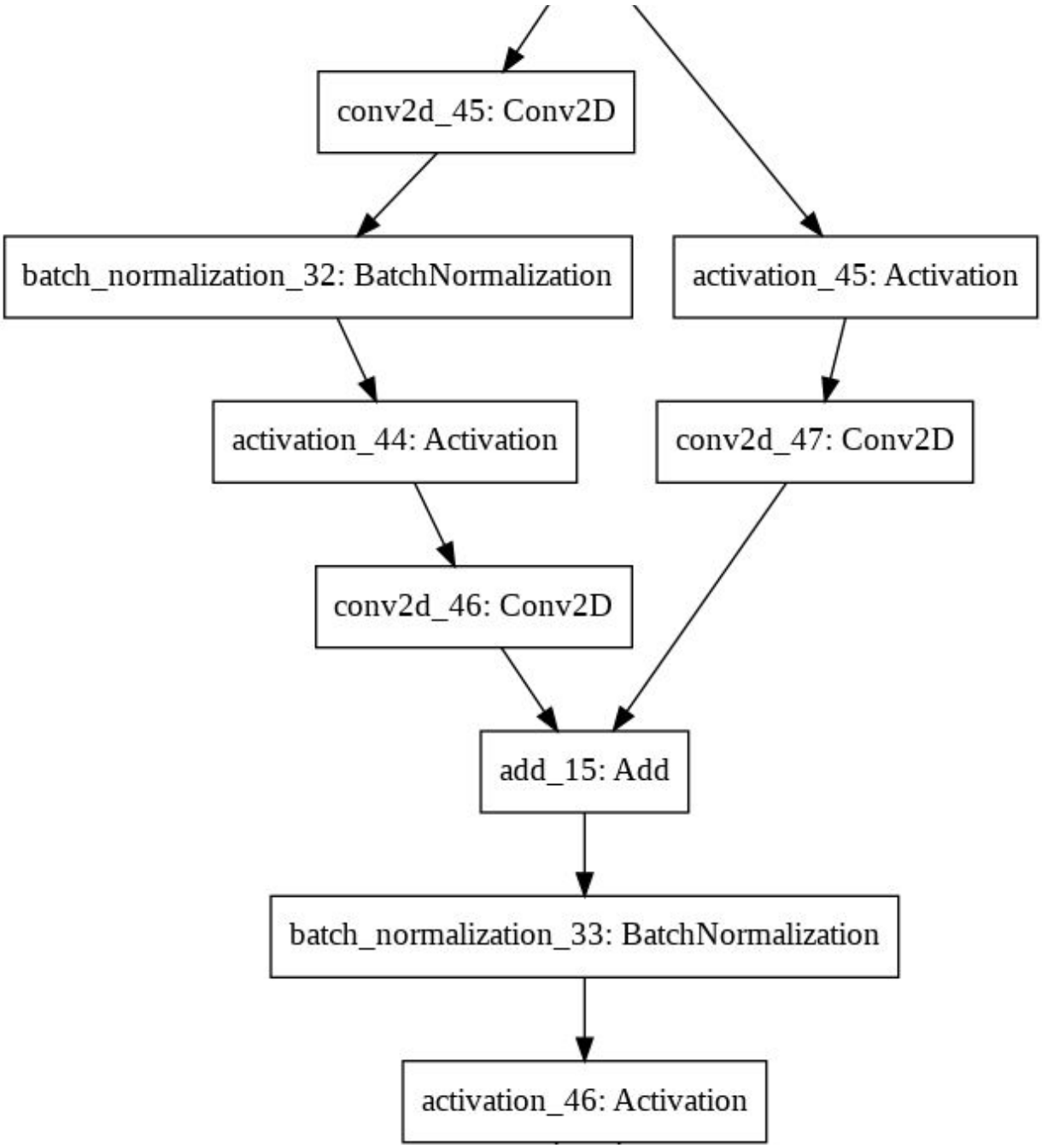

batch_normalization_23: BatchNormalization
activation_31: Activation
conv2d_33: Conv2D
batch_normalization_24: BatchNormalization
activation_33: Activation
activation_32: Activation
conv2d_35: Conv2D
conv2d_34: Conv2D
add_11: Add

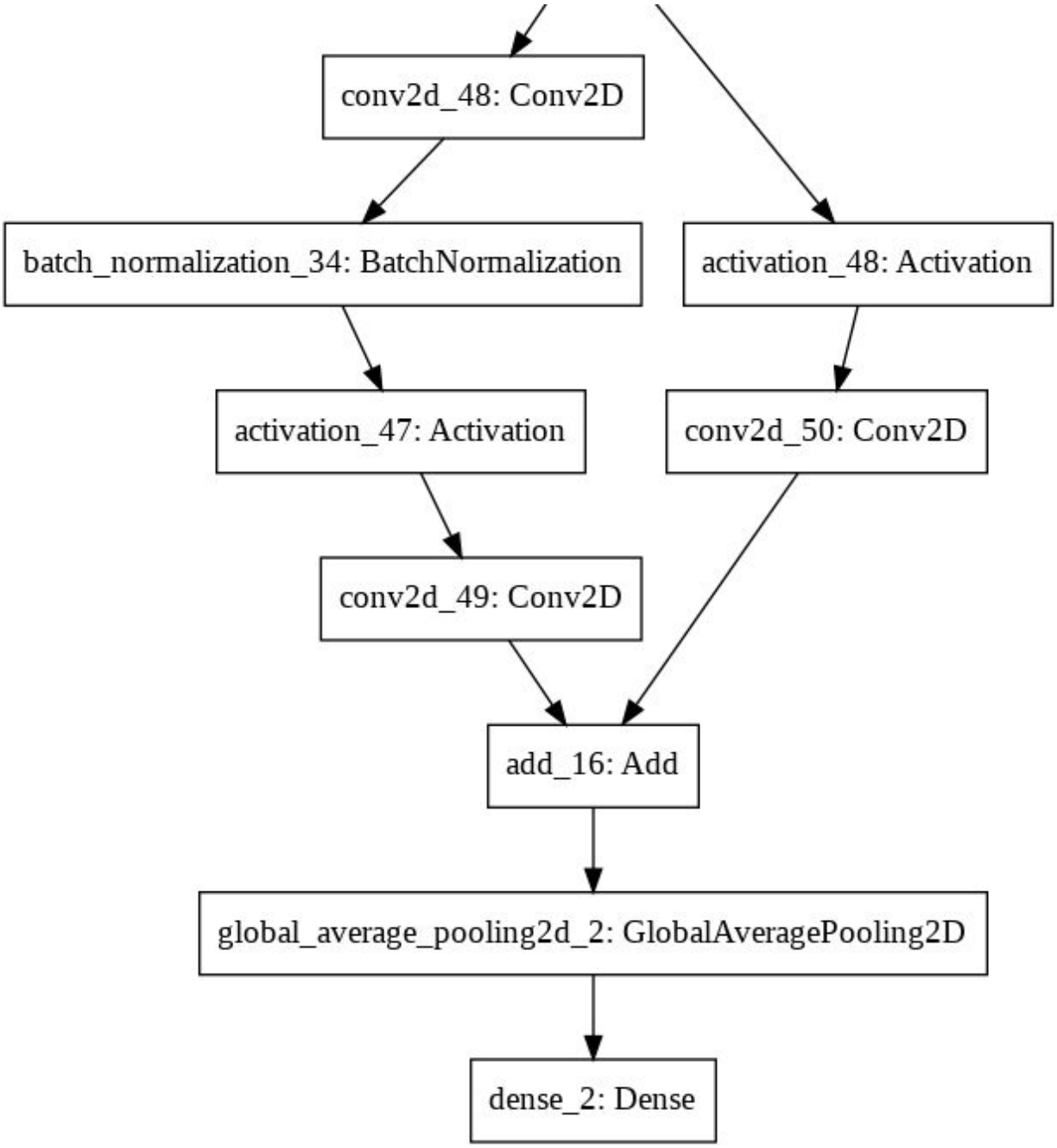

batch_normalization_25: BatchNormalization
activation_34: Activation
conv2d_36: Conv2D
batch_normalization_26: BatchNormalization
activation_36: Activation
activation_35: Activation
conv2d_38: Conv2D
conv2d_37: Conv2D
add_12: Add
batch_normalization_27: BatchNormalization


activation_37: Activation
conv2d_39: Conv2D
batch_normalization_28: BatchNormalization
activation_39: Activation
activation_38: Activation
conv2d_41: Conv2D
conv2d_40: Conv2D
add_13: Add
batch_normalization_29: BatchNormalization
activation_40: Activation


activation_40: Activation
conv2d_42: Conv2D
batch_normalization_30: BatchNormalization
activation_42: Activation
activation_41: Activation
conv2d_44: Conv2D
conv2d_43: Conv2D
add_14: Add
batch_normalization_31: BatchNormalization


conv2d_42: Conv2D
batch_normalization_30: BatchNormalization
activation_42: Activation
activation_41: Activation
conv2d_44: Conv2D
conv2d_43: Conv2D
add_14: Add
batch_normalization_31: BatchNormalization
activation_43: Activation
conv2d_45: Conv2D


conv2d_45: Conv2D
batch_normalization_32: BatchNormalization
activation_45: Activation
activation_44: Activation
conv2d_47: Conv2D
conv2d_46: Conv2D
add_15: Add
batch_normalization_33: BatchNormalization
activation_46: Activation


conv2d_48: Conv2D
batch_normalization_34: BatchNormalization
activation_48: Activation
activation_47: Activation
conv2d_50: Conv2D
conv2d_49: Conv2D
add_16: Add
global_average_pooling2d_2: GlobalAveragePooling2D
dense_2: Dense

Experiment 3:

Implement the Alexnet models in Keras. Summarize Accuracy measures for Train, Test and Validation for the best model

```
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

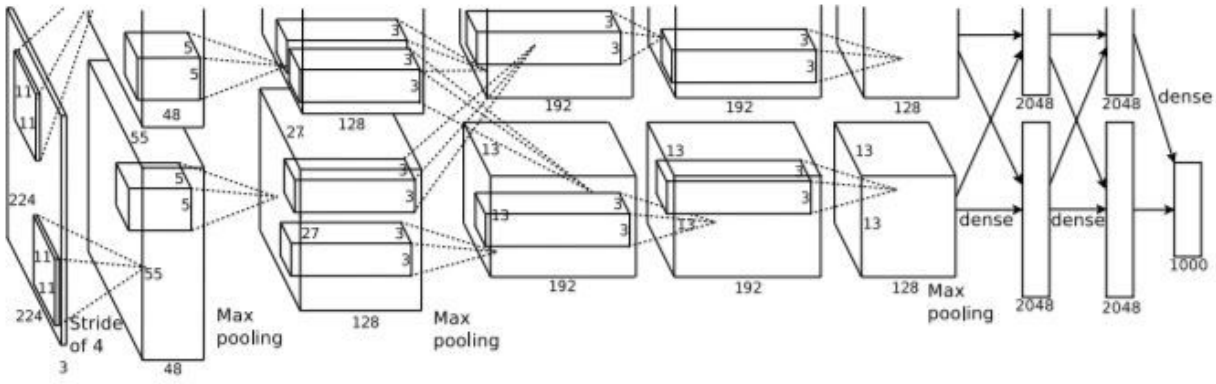
```
(70000, 32, 32, 3)
(70000, 200)
(30000, 32, 32, 3)
(30000, 200)
```

Data:

We implement the Alexnet model in Keras with some modification on our "Tiny-Imagenet/train" data set. The portion of train and test data set is 7 : 3 which are 70000 and 30000 pictures correspondingly.

Architecture:

The AlexNet architecture consists of five convolutional layers, some of which are followed by maximum pooling layers and then three fully-connected layers and finally a 1000-way softmax classifier.



In the original paper, all the layers are divided into two to train them on separate GPUs. Since it is a complex arrangement and difficult to understand, we will implement AlexNet model in one layer concept.

First Layer:

The input for AlexNet is a 227x227x3 RGB image which passes through the first convolutional layer with 96 feature maps or filters having size 11×11 and a stride of 4. The image dimensions changes to 55x55x96.

(227-11)/4+1=55 (add one more pixel from the 11x11)

Then the AlexNet applies maximum pooling layer or sub-sampling layer with a filter size 3×3 and a stride of two. The resulting image dimensions will be reduced to 27x27x96. (55-3)/2+1=27

Our First Layer:

```
model.add(Conv2D(96, (3,3), strides=(2,2), activation='relu', padding='same', input_shape=(x_train.shape[1],x_train.shape[2],x_t
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2,2)))
# Local Response normalization for Original Alexnet
model.add(BatchNormalization())
```

```
conv2d_1 (Conv2D)            (None, 16, 16, 96)        2688
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 96)          0
_________________________________________________________________
batch_normalization_1 (Batch (None, 8, 8, 96)          384
_________________________________________________________________
```

Our input shape is 32x32x3 after reshape the “Tiny-Imagenet” image size, and with a stride of 2, the dimensions change to 16x16x96. Finally, we add one more batch_normalization layer after the pooling layer.

Second Layer:

Next, there is a second convolutional layer with 256 feature maps having size 5×5 and a stride of 1.

Then there is again a maximum pooling layer with filter size 3×3 and a stride of 2. This layer is same as the second layer except it has 256 feature maps so the output will be reduced to 13x13x256.

Our Second Layer:

```
model.add(Conv2D(256, (5,5), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2,2)))
# Local Response normalization for Original Alexnet
model.add(BatchNormalization())
```

```
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 256)         614656
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 3, 3, 256)         0
_________________________________________________________________
batch_normalization_2 (Batch (None, 3, 3, 256)         1024
```

Third, Fourth and Fifth Layers:

The third, fourth and fifth layers are convolutional layers with filter size 3×3 and a stride of one. The first two used 384 feature maps where the third used 256 filters.

The three convolutional layers are followed by a maximum pooling layer with filter size 3×3, a stride of 2 and have 256 feature maps.

Our Third, Fourth and Fifth Layers:

```
model.add(Conv2D(384, (3,3), activation='relu', padding='same'))
model.add(Conv2D(384, (3,3), activation='relu', padding='same'))
model.add(Conv2D(256, (3,3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2,2)))
# Local Response normalization for Original Alexnet
model.add(BatchNormalization())
```

```
conv2d_3 (Conv2D)            (None, 3, 3, 384)         885120
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 3, 3, 384)         1327488
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 3, 3, 256)         884992
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 1, 1, 256)         0
_________________________________________________________________
batch_normalization_3 (Batch (None, 1, 1, 256)         1024
_________________________________________________________________
```

Sixth Layer:

The convolutional layer output is flatten through a fully connected layer with 9216 feature maps each of size 1×1.

```
model.add(Flatten())
```

```
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
```

Seventh and Eighth Layers:

Next is again two fully connected layers with 4096 units. And two dropout layer with rate as 0.5.

```
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='tanh'))
model.add(Dropout(0.5))
```

```
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              1052672
_________________________________________________________________
dropout_1 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_2 (Dropout)          (None, 4096)              0
_________________________________________________________________
```

Output Layer:

Finally, there is a softmax output layer ŷ with 200 possible values.

```
model.add(Dense(num_classes, activation='softmax'))
```

```
_________________________________________________________________
dense_3 (Dense)              (None, 200)               819400
=================================================================
Total params: 22,370,760
Trainable params: 22,369,544
Non-trainable params: 1,216
```

Deploy Model:

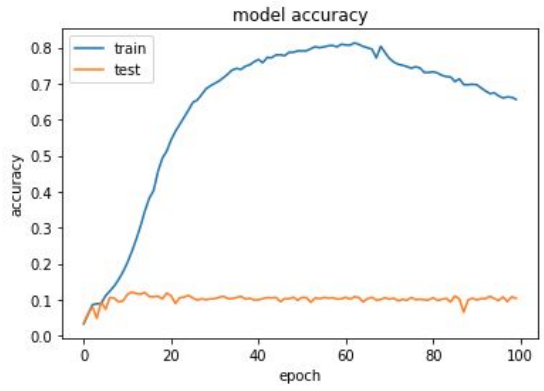
```
hist_1 = model.fit(x_train, y_train,
                   batch_size=64,
                   epochs=100,
                   shuffle=True,
                  validation_data=(x_test, y_test))
```

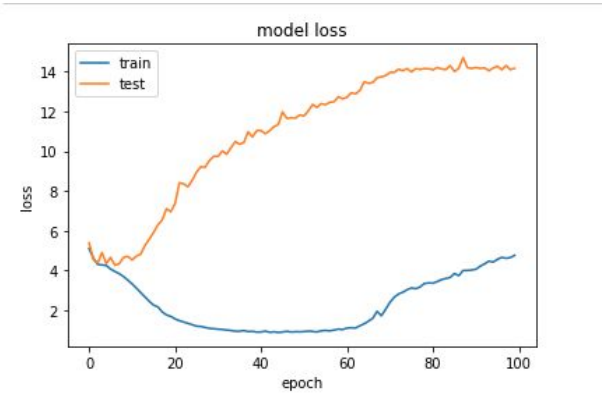
Results:

Peak training accuracy at epoch 63 as 0.8134 with 1.1061 training loss;

Test loss: 14.159581217447917

Test accuracy: 0.10366666666666667

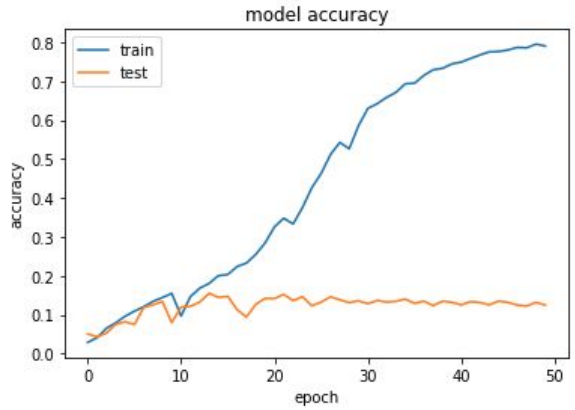


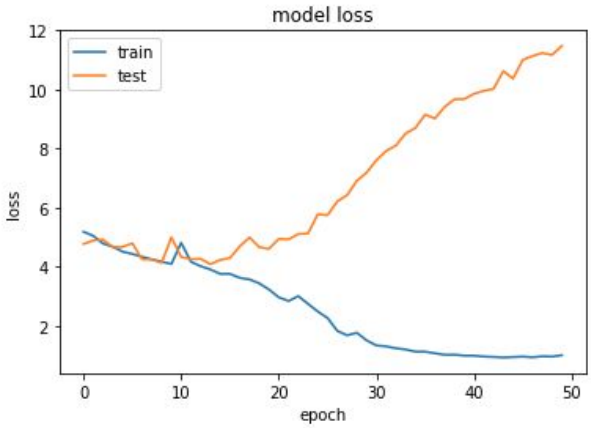


Epoch = 50:

Test loss: 11.464195709228516

Test accuracy: 0.1251





Comparing with 100 epochs, the training result looks great! However, it is still over-fitting because the test loss is way higher than the training loss.

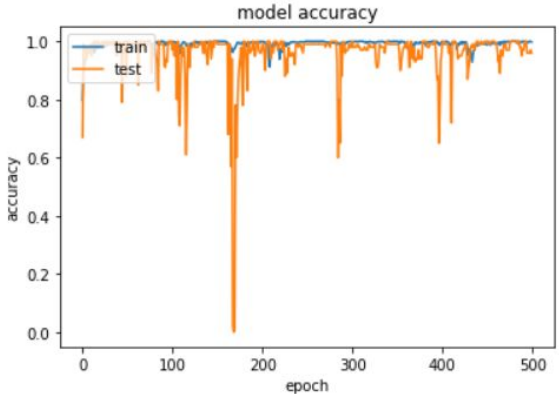
Epoch = 500 with only 2 classes data:

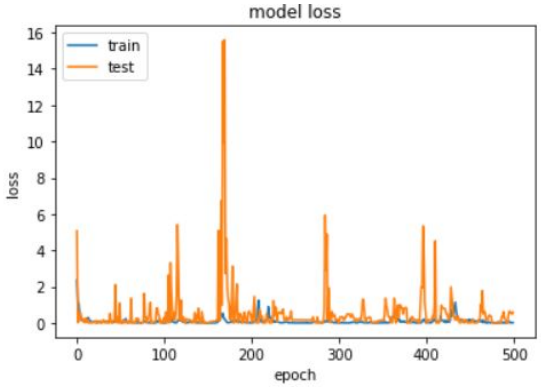
Not only shrink the data set to 2 classes but split the validation data from the training data set after finishing an epoch.

```
hist = model.fit(x_train, y_train,
                 batch_size=46,
                 epochs=500,
                 validation_split=0.1,
                 shuffle=True)
```

Test loss: 0.0582424342563

Test accuracy: 0.996





Analysis:

Epoch = 100:

The training accuracy and loss are reasonably tuned before epoch 60 but become weird after that. So we run another training for only 50 epoch below to see if this is because of large number of epoch.
And validation accuracy and loss do not make sense at all. This might because the validation data set is too large which the model does not learn very well.

Epoch = 50:

From the training plot, the convergence is not obvious which means that the epoch number is too low, the curve might be more stable if we run more epoches, like 1000+ epoches. This make perfect sense because our data set is huge and the image complexity is high. And the validation data set is too large comparing with training data set, we should decrease the portion of validation data set and run more epoches. However, because our computer's capacity, we better use AWS or more powerful GPU.

Epoch = 500 with only 2 classes data:

The result reaches our expectation, we get high accuracy and low loss for both train and validation, which proves our assumption that if the number of epoch is high enough, we could get a good result by AlexNet model.

Experiment 4:

Use the pre-trained model as per your team assignment -- InceptionV3

Data:

The same data set as above but the pictures are reshaped with size 150x150 because the minimum of Inception V3 input image's width and length is 75. However, the default input size is 299x299.

Architecture:

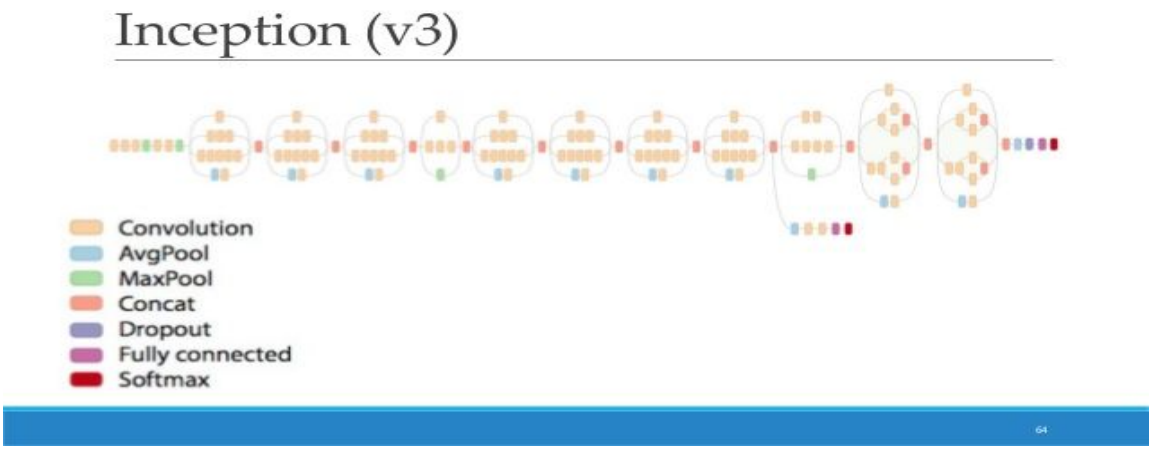
Inception V3 model, with weights pre-trained on ImageNet.

```
num_classes = 200

base_model = inception_v3.InceptionV3

base_model = base_model(weights='imagenet', include_top=False)
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)
model = Model(inputs=base_model.input, outputs=predictions)
for layer in base_model.layers:
    layer.trainable = False
model.summary()
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=Adam(lr=0.0001),
              metrics=['acc'])
```

```
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.5/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5
87916544/87910968 [==============================] - 3s 0us/step
```



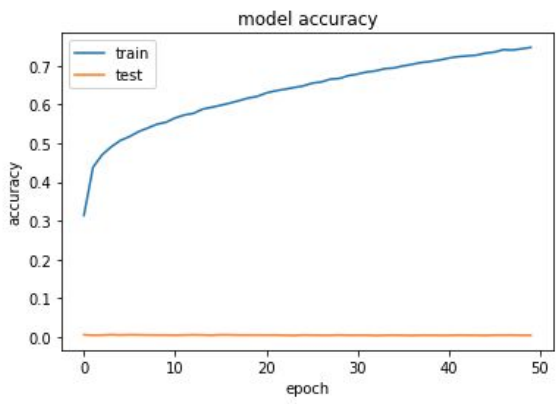
After the base model of inception v3, we add one globalaveragepooling layer, one 1024 dense layer with ‘relu’ and a final dense layer of 200, which is the class number, with ‘softmax’.

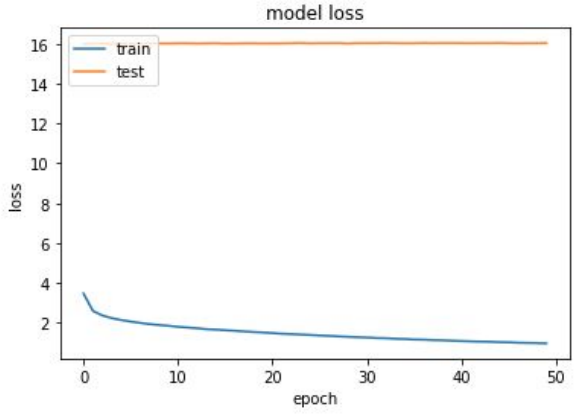
```
global_average_pooling2d_1 (Glo (None, 2048)         0           mixed10[0][0]
_____________________________________________________________________________________
dense_1 (Dense)                 (None, 1024)         2098176     global_average_pooling2d_1[0][0]
_____________________________________________________________________________________
dense_2 (Dense)                 (None, 200)          205000      dense_1[0][0]
=====================================================================================
Total params: 24,105,960
Trainable params: 2,303,176
Non-trainable params: 21,802,784
_____________________________________________________________________________________
```

Result:

Test loss: 16.04263661295573

Test accuracy: 0.004233333333333334



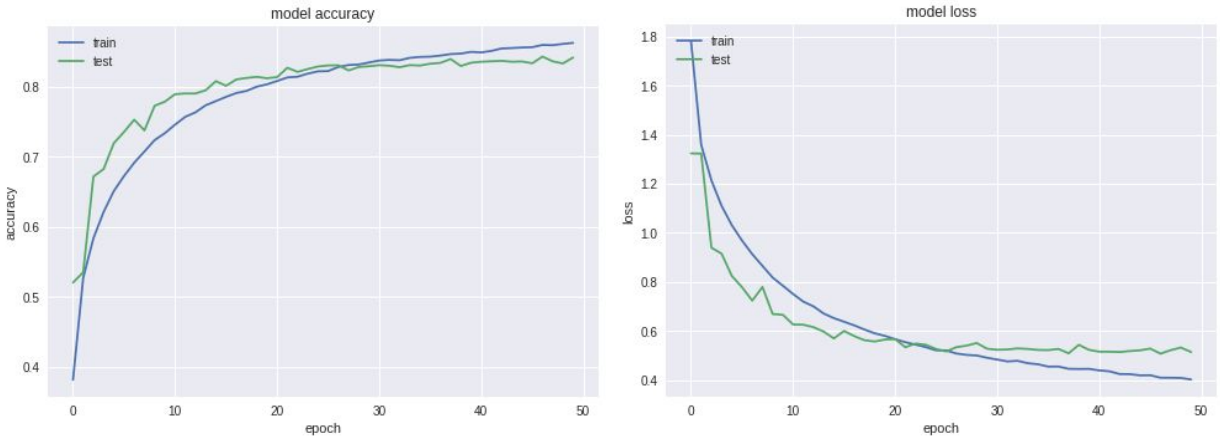


Analysis:
Our training goes well but the validation is not learning at all. Same as part3, the data set is too large and the image is too complex. We need to run more epoches or decrease our data set.

Experiment 5: Transfer Learning
Apply Transfer learning for your team's network to the CIFAR 10 dataset.

Data: Tiny Imagenet 200 dataset as above.

Step 1:
To apply transfer learning to this dataset, it is first of all needed that, to train and save a model with it weights. Accordingly the program uses the CIFAR 10 pre-trained neural network as a base model.
(Please refer to:
https://colab.research.google.com/drive/19OGj3N-_Yateh16qL5seqU6UCaA3F3nN)
It's metrics shows:




```
10000/10000 [==============================] - 2s 173us/step
Test loss: 0.5143789597988129
Test accuracy: 0.8423
```

Using checkpoints during the training may help the further experiments. The programme saved the model at 50 epochs, which is slightly overfitting.

Step 2:
The programme loaded and cut the model in to required shape, by doing:

```
from keras.models import model_from_json
# load json and create model
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")
```

And after doing so, use pop() to get rid of the last layers which makes predictions. This results in a model like:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation_1 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
activation_2 (Activation)    (None, 30, 30, 32)        0
_________________________________________________________________
batch_normalization_2 (Batch (None, 30, 30, 32)        128
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
spatial_dropout2d_1 (Spatial (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 64)        18496
_________________________________________________________________
activation_3 (Activation)    (None, 15, 15, 64)        0
_________________________________________________________________
batch_normalization_3 (Batch (None, 15, 15, 64)        256
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 64)        36928
_________________________________________________________________
activation_4 (Activation)    (None, 13, 13, 64)        0
_________________________________________________________________
batch_normalization_4 (Batch (None, 13, 13, 64)        256
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
spatial_dropout2d_2 (Spatial (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 6, 6, 128)         73856
_________________________________________________________________
activation_5 (Activation)    (None, 6, 6, 128)         0
_________________________________________________________________
batch_normalization_5 (Batch (None, 6, 6, 128)         512
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 4, 4, 128)         147584
_________________________________________________________________
activation_6 (Activation)    (None, 4, 4, 128)         0
```

```
_________________________________________________________________
batch_normalization_6 (Batch (None, 4, 4, 128)         512
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 2, 2, 128)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 2, 2, 128)         0
=================================================================
Total params: 288,800
Trainable params: 287,904
Non-trainable params: 896
```

Step 3:
Now that the programme gets a model with certain knowledges about patterns learnt from CIFAR 10 dataset, let us see how these knowledges works for imagenet data.

```
[ ]  extracted_features = loaded_model.predict(x_train)
     print(extracted_features.shape)
```

```
(70000, 2, 2, 128)
```

```
validation_features = loaded_model.predict(x_test)
print(validation_features.shape)
```

```
(30000, 2, 2, 128)
```

By simply walking the imagenet data through the pretrained network, the programme extracts features who are useful for prediction -- CIFAR 10 prediction.

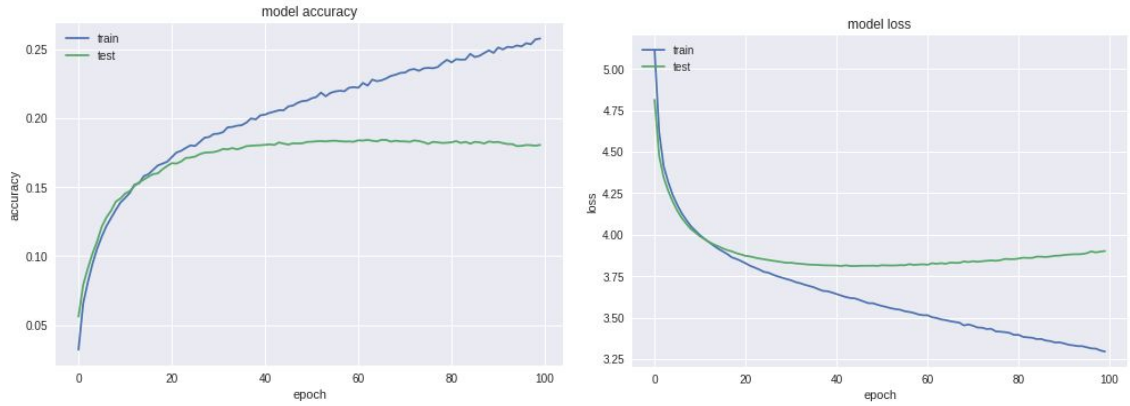
Step 4:
Apply those features into a neural network, and make predictions.

The network is as simple as:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_5 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_9 (Dense)              (None, 512)               1573376
_________________________________________________________________
activation_9 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_5 (Dropout)          (None, 512)               0
```

```
_________________________________________________________________
dense_10 (Dense)             (None, 200)               102600
_________________________________________________________________
activation_10 (Activation)   (None, 200)               0
=================================================================
Total params: 1,675,976
Trainable params: 1,675,976
Non-trainable params: 0
```

It takes the input of exactly the same shape as the features have been extracted.
The performance shows:



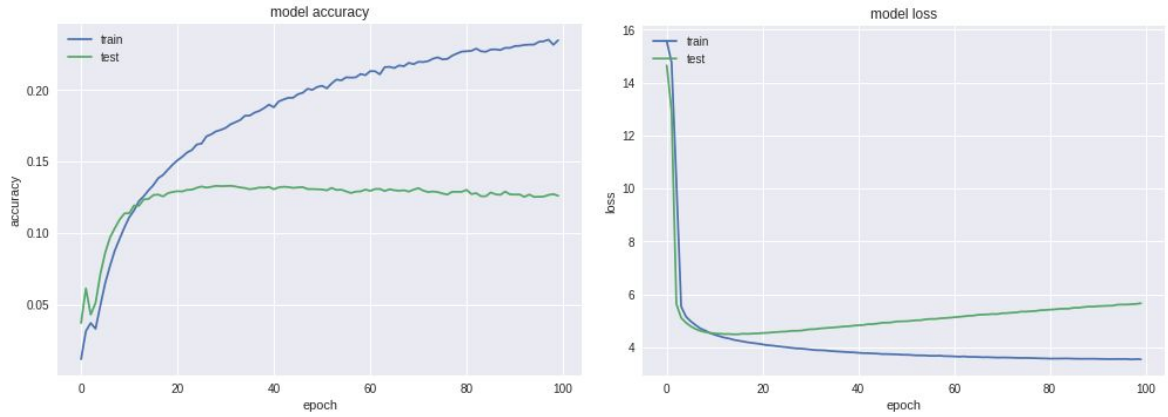
Last epoch: `loss: 3.2950 - acc: 0.2576 - val_loss: 3.9016 - val_acc: 0.1806`
The team observed a lack of learning capability in this model. So what might be the problem?

Hypothesis 1: Model. This pretrained model is not able to learn enough features in new data. This can be either because of a low similarity between two datasets, or a low learning capability of model itself.

Hypothesis 2: Data. The data has lost too much information before being applied to the model.
To demonstrate against these hypothesis, this experiment includes the following step.

Step 5:
Hypothesis 1:
The programme introduced a pre-trained vgg-16 model to perform the same steps above. Vgg-16 is trained upon the dataset of imagenet and proved to be a well performed one.

And so gives a result of:

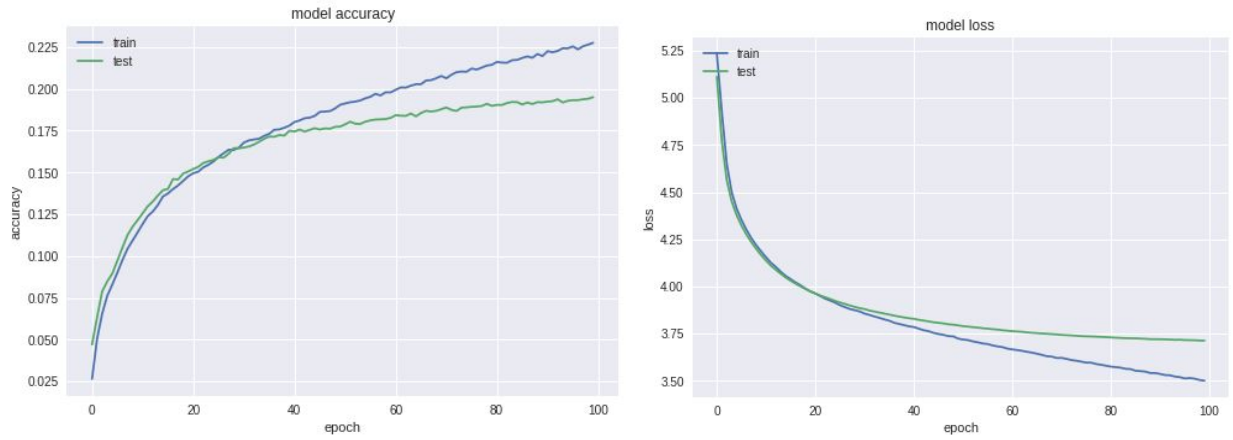



```
loss: 3.5408 - acc: 0.2347 - val_loss: 5.6596 - val_acc: 0.1260
```

The learning curve gives a reasonable slope. It is well pre-trained, and thus the slope is steep during the early epochs.

However, since vgg16 is much more complex, and as well proved to be better performed than the first pre-trained model, it is supposed to give a higher accuracy as results. Yet it's metrics tells otherwise. So the team assumed the problem is not mainly about the complexity of the model. It is most likely to be something data-wise.

Hypothesis 2:
The programme increased the training data by a scale of 28.5%. So it gives a result of:




```
loss: 3.5007 - acc: 0.2275 - val_loss: 3.7130 - val_acc: 0.1950
```

...which is a increasing in comparison with the first try. The problem can be data-wise. Tiny imagenet gives only 500 images per class, which could hardly be sufficient even for a human brain with 100 billion neurons...

Experiment 6: Fine Tuning

Data: Tiny Imagenet 200 as above.

Step 1:
Load and trim the pre-trained model as it is in Experiment 5.

Step 2:
Freeze the early layers in this model, and train only the late layers. For example:

```
……
<keras.layers.core.Activation object at 0x7f4ab492b518> False
<keras.layers.normalization.BatchNormalization object at 0x7f4ab4888898>
False
<keras.layers.convolutional.Conv2D object at 0x7f4ab48ea8d0> True
<keras.layers.core.Activation object at 0x7f4ab4888908> True
<keras.layers.normalization.BatchNormalization object at 0x7f4ab47be198>
True
<keras.layers.pooling.MaxPooling2D object at 0x7f4ab4807438> True
<keras.layers.core.Dropout object at 0x7f4ab47be4a8> True
```

Step 3:
Create a model, including the loaded model as its first layers.

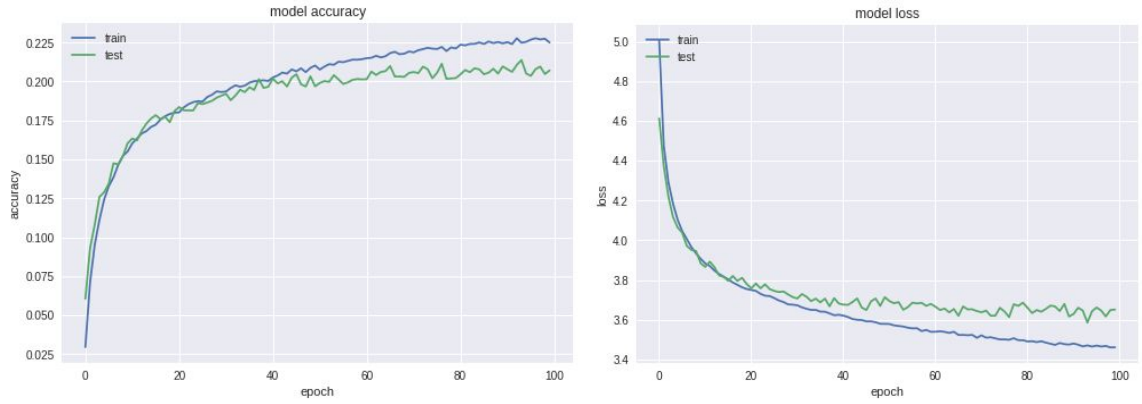
```
# Create the model
model = models.Sequential()

# Add the vgg convolutional base model
model.add(loaded_model)

# Add new layers
model.add(layers.Flatten())
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(200, activation='softmax'))
```

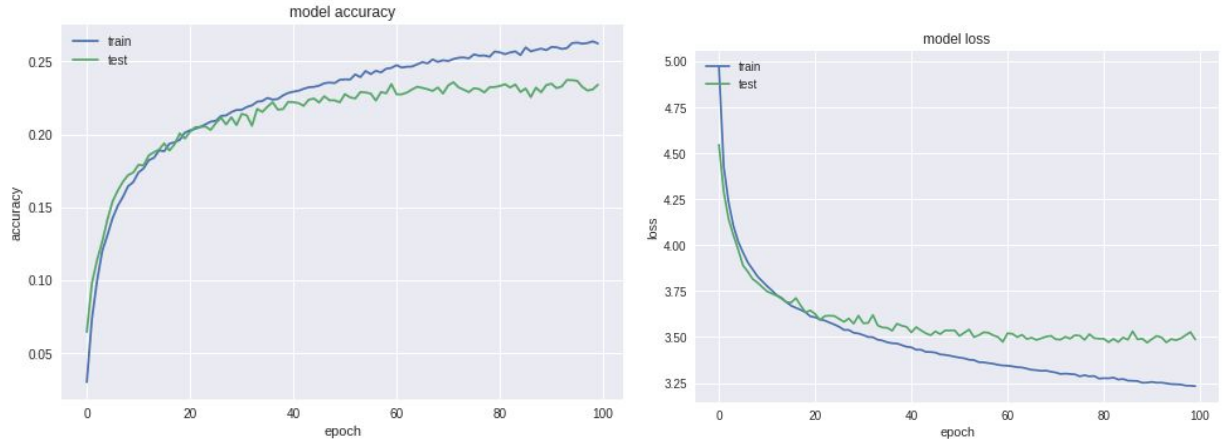
During the training, the trainable layers in loaded model it trained along with the additional layers.

The following figures shows the performance when:
5 layers are trainable:



loss: 3.4617 - acc: 0.2250 - val_loss: 3.6515 - val_acc: 0.2071

8 layers are trainable:

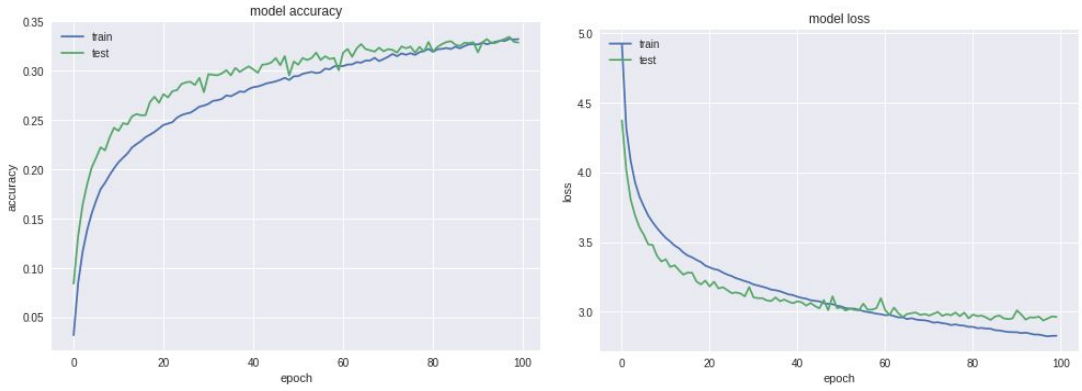


15s 164us/step - loss: 3.2330 - acc: 0.2623 - val_loss: 3.4872 - val_acc: 0.2341

13 layers are trainable:

18s 195us/step - loss: 3.0303 - acc: 0.2958 - val_loss: 3.3038 - val_acc: 0.2659

...
And when all layers are released...




```
31s 344us/step - loss: 2.8259 - acc: 0.3322 - val_loss: 2.9612 -
val_acc: 0.3288
```

An increase in validation scores appears while more layers are released for training, but the training also consumes much more time than ever. There can be a trade-off problem between performance and efficiency when applying fine-tuning.