Assignment 2 Report
Group 6
https://github.com/miquelmarti/tiny-imagenet-classifier/blob/master/tiny_imagenet_classifier.py

Experiment 1

This experiment required that we first convert the 64 by 64 tiny_images dataset into a 32 by 32 data set and then build a convolutional neural network classifier to classify all the images into 200 classes. After converting the images, we used two functions to load the images into our script. We split the images into both training and testing datasets with training having 50,000 images and testing having 10000 images.as shown in the image below;

```
Loading 200 classes
loading training images...
finished loading training images
loading test images...
finished loading test images
0
0
1
1
```

Train data

We then build a convolutional neural network described below;

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_215 (Conv2D)          (None, 2, 31, 32)         4128
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 1, 15, 32)         0
_________________________________________________________________
dense_9 (Dense)              (None, 1, 15, 1024)       33792
_________________________________________________________________
dense_10 (Dense)             (None, 1, 15, 512)        524800
_________________________________________________________________
dense_11 (Dense)             (None, 1, 15, 512)        262656
_________________________________________________________________
dense_12 (Dense)             (None, 1, 15, 512)        262656
_________________________________________________________________
dense_13 (Dense)             (None, 1, 15, 512)        262656
_________________________________________________________________
dense_14 (Dense)             (None, 1, 15, 512)        262656
_________________________________________________________________
dense_15 (Dense)             (None, 1, 15, 512)        262656
_________________________________________________________________
flatten_3 (Flatten)          (None, 7680)              0
_________________________________________________________________
dense_16 (Dense)             (None, 512)               3932672
_________________________________________________________________
dense_17 (Dense)             (None, 200)               102600
=================================================================
Total params: 5,911,272
Trainable params: 5,911,272
Non-trainable params: 0
_________________________________________________________________
None
```

The set up of the network below, including several hyperparameters such as learning rate, the number of epochs, momentum are shown in the graphic below. We used categorical_crossentropy as the loss function because we are trying to classify over 200

classes. As an optimiser, we used stochastic gradient descent and used accuracy as the evaluation metric.

```
# Compile model e
epochs = 30
lrate = 0.01
```

```
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
print(model.summary())
```

The best accuracy we got was however much lower than expected. After multiple iterations and experiments, the best accuracy we go was 12.67%

Experiment 2

AutoKeras is an open source python package embedded with a very easy to use deep learning library Keras. It uses a variant of ENAS, an efficient and most recent version of Neural Architecture Search. It automatically searches for the hyperparameters of deep learning models. It reduces the technical barrier and allows domain experts to utilise the power of machine learning for their business. The current version of Auto-Keras provides functions to automatically search for the hyperparameters of deep learning models.

We used autokeras to try to find the best optimal model on a resized tinyimages dataset from imagenet of size (32 by 32).

The using accuracy as the metric for evaluating performance, the best model that we got had an an accuracy of ….

The model design consisted of setting the time variables for which you wish the model to be running. We selected 1 hour, 2 hours and 4 hours. The biggest disadvantage of using autokeras is that it took a very long time before we got results, and as such it is very computationally intensive, even if it requires less skill to use

Experiment 3

Alex's alexnet network structure model in 2012 triggered a boom in the application of neural networks and won the championship of the 2012 image recognition contest, making CNN the core algorithm model in image classification.

The first layer input data is the original 32*32*3 image, which is convoluted by the convolution core of 3*3*3. The convolution core generates a new pixel for each convolution of the original image. The convolution core moves along the x-axis direction and Y-axis direction of the original image, and the moving step is 2 pixels. Therefore, the convolution core generates (32-3)/2+1=16 pixels , and 16*16 pixels of rows and columns form the pixel layer after convolution of the original image. There are 96 convolution cores, which generate 16 * 16 * 96 convoluted pixel layers. 96 convolution cores were divided into two groups, 48 convolution cores in each group. Two sets of convoluted pixel-level data of 16*16*48 are generated. These pixel

layers are processed by relu1 unit to generate active pixel layers, which are still two sets of 16*16*48 pixel layer data.

The input data of the second layer is the 16*16*96 pixel layer of the output of the first layer. In order to facilitate the subsequent processing, two pixels are filled on the left and right sides and the upper and lower sides of each pixel layer. The 16*16*96 pixel data is divided into two groups of 16*16*48 pixel data, and the two groups of data are processed in two different GPUs. Each group of pixel data is convoluted by a 5*5*48 convolution core, which generates a new pixel for each convolution of each group of data. The convolution core moves along the x-axis direction and Y-axis direction of the original image, and the moving step is one pixel. Therefore, the convolution core generates 8 pixels in the process of moving. 16*16 pixels of rows and columns form the pixel layer after convolution of the original image. There are 256 5*5*48 convolution cores, which are divided into two groups. Each group performs convolution operations on 16*16*48 pixels in a GPU. Two sets of 16*16*128 convoluted pixel layers are generated. These pixel layers are processed by relu2 unit to generate active pixel layers, which are still two groups of 16*27*128 pixel layers.These pixel layers are processed by pool operation (pooling operation), produce the output 8*8*96.

The input data of the third layer are two sets of 8*8*96 pixel layers of the second layer output; for the convenience of subsequent processing, one pixel is filled on the left and right sides and the upper and lower sides of each pixel layer; the two sets of pixel layer data are sent to two different GPUs for operation. There are 192 convolution cores in each GPU, and the size of each convolution core is 3*3*256. Therefore, the convolution core in each GPU can convolute all data of a set of 8*8*96 pixel layers. The convolution check generates a new pixel for each convolution of each set of data. The convolution core moves along the x-axis and Y-axis directions of the pixel-level data, and the moving step is 2 pixel. Therefore, the size of the convolution core after operation is 8. There are 8*8*96 convolution cores in each GPU. There are 8*8*192 convoluted pixel layers in two GPUs. These pixel layers are processed by relu3 units to generate active pixel layers, which are still two groups of 8*8*96 pixel layers with a total of 8*8*192 pixel layers..These pixel layers are processed by pool operation (pooling operation), produce the output 8*8*96.

The input data of the fourth layer are two sets of 8*8*96 pixel layers output from the third layer; for the convenience of subsequent processing, one pixel is filled on the left and right sides and the upper and lower sides of each pixel layer; the two sets of pixel layer data are sent to two different GPUs for operation. There are 192 convolution cores in each GPU. The size of each convolution core is 3*3*192. Therefore, the convolution core in each GPU can convolute a set of 8*8*96 pixel level data. The convolution check generates a new pixel for each convolution of each set of data. The convolution core moves along the x-axis and Y-axis directions of the pixel-level data, and the moving step is 2 pixel. Therefore, the size of the convolution core after the operation is 8. There are 8*8*96 convolution cores in each GPU. There are 13*13*256 convoluted pixel layers in two GPUs. These pixel layers are processed by relu4 units to

generate active pixel layers, which are still two groups of 8*8*96 pixel layers with a total of 8*8*256 pixel layers.

The fifth layer input data are two sets of 8*8*256 pixel layers of the fourth layer output; for the convenience of subsequent processing, one pixel is filled on the left and right sides and the upper and lower sides of each pixel layer; the two groups of pixel layer data are sent to two different GPUs for operation. There are 128 convolution cores in each GPU, and the size of each convolution core is 3*3*192. Therefore, the convolution core in each GPU can convolute a set of 8*8*256 pixel level data. The convolution check generates a new pixel for each convolution of each set of data. The convolution core moves along the x-axis and Y-axis directions of the pixel-level data, and the moving step is 2 pixel. Therefore, the size of the convolution core after the operation is 8. There are 8*8*256 convolution cores in each GPU. There are 8*8*256 convoluted pixel layers in two GPUs. These pixel layers are processed by relu5 units to generate active pixel layers, which are still two groups of 8*8*128 pixel layers with a total of 8*8*256 pixel layers. These pixel layers are processed by pool operation (pooling operation), produce the output 3*3*256.

The size of the input data of the sixth layer is 3*3*256, and the input data of the sixth layer is convoluted by a 3*3*256 size filter. Each 3*3*256 size filter convolutes the input data of the sixth layer to produce an operation result, which is output by a neuron. A total of 4096 3*3*256 size filters convolute the input data. 4 096 neurons are used to output the results; 4 096 neurons are used to generate 4 096 values by relu activation function;
and 4 096 neurons are used to output the results by drop operation.

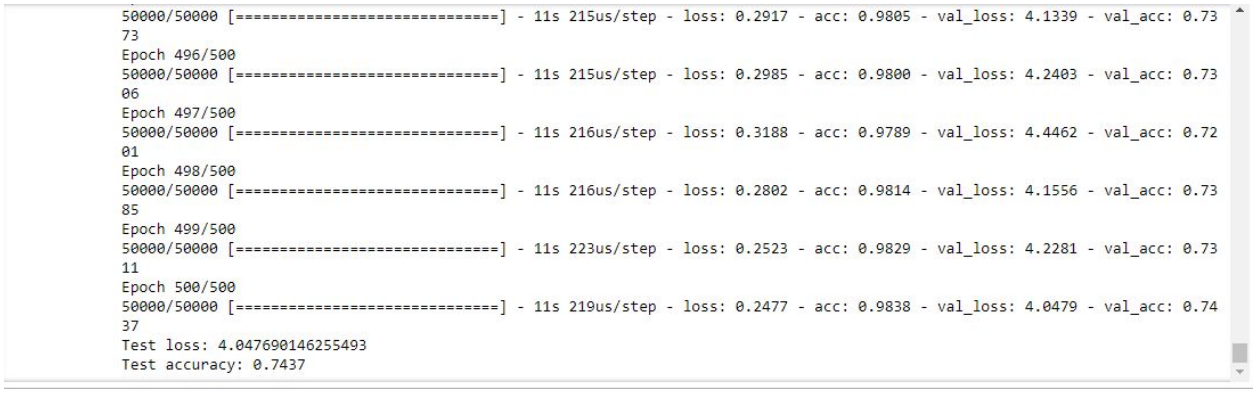
The 4096 data from the seventh level are fully connected with 1000 neurons from the eighth level, and the trained values are output after training. And for all layers structure:

Layer (type)	Output Shape	Param #
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
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
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
flatten_1 (Flatten)	(None, 256)	0
dense_1 (Dense)	(None, 4096)	1052672
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 4096)	16781312
dropout_2 (Dropout)	(None, 4096)	0
dense_3 (Dense)	(None, 200)	819400

max_pooling2d_3 (MaxPooling2	(None, 1, 1, 256)	0
batch_normalization_3 (Batch	(None, 1, 1, 256)	1024
flatten_1 (Flatten)	(None, 256)	0
dense_1 (Dense)	(None, 4096)	1052672
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 4096)	16781312
dropout_2 (Dropout)	(None, 4096)	0
dense_3 (Dense)	(None, 200)	819400

Total params: 22,370,760
Trainable params: 22,369,544
Non-trainable params: 1,216

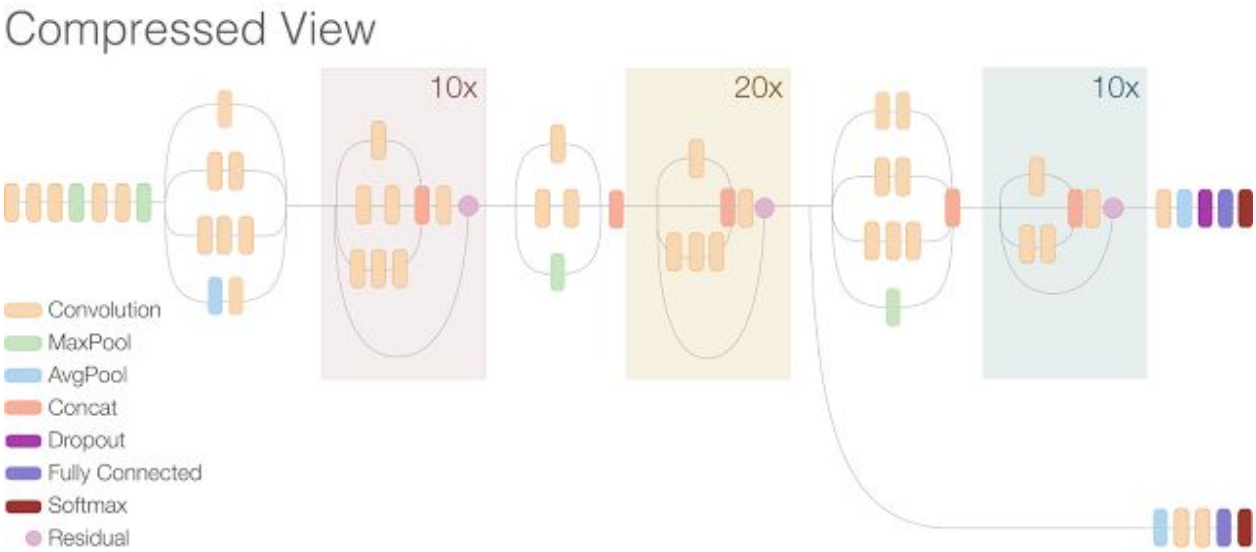
We set batch size = 64, learning rate = 0.01, epochs = 500 and optimizer=SGD.
The results as follows:



We got accuracy at 0.7437.

Experiment 4

Inception-ResNet-v2 is a variation of Inception V3 model, and it is considerably deeper than the previous Inception V3. Below in the figure is an easier to read version of the same network where the repeated residual blocks have been compressed. Here, notice that the inception blocks have been simplified, containing fewer parallel towers than the previous Inception V3. The Inception-ResNet-v2 architecture is more accurate than previous state of the art models.



At the top of the second Inception-ResNet-v2 figure, you'll see the full network expanded. Notice that this network is considerably deeper than the previous Inception V3. Below in the main figure is an easier to read version of the same network where the repeated residual blocks have been compressed. Here, notice that the inception blocks have been simplified, containing fewer parallel towers than the previous Inception V3.
We use InceptionResNetv2, which is already pre-trained on ImageNET database. Next we add some additional layers in order to train the network on CIFAR10 dataset.
We used the keras python deep learning library. Namely, we follow keras.applications tutorial. Here is the example to load the InceptionResNetV2 CNN with keras:

```
from keras.applications.inception_resnet_v2 import InceptionResNetV2
model = InceptionResNetV2(include_top=False, weights='imagenet', \
                input_tensor=None, input_shape=None, pooling=None, classes=1000)
```

Here is how we create model for InceptionResNetV2:

```
def create_model_incv2():
    tf_input = Input(shape=input_shape)
    base_model = InceptionResNetV2(input_tensor=tf_input, weights='imagenet',
include_top=False)
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(1024, activation='relu')(x)
```

```
    predictions = Dense(n_classes, activation='softmax')(x)
    model = Model(inputs=base_model.input, outputs=predictions)
    return base_model, model
```

For this experiment, we use CIFAR10 dataset. Because InceptionResNetV2 model has limitations that width and height should be no smaller than 75 and original size is 32*32, we need resize the images:

```
def data_generator(sess,data,labels):
    def generator():
        start = 0
        end = start + batch_size
        n = data.shape[0]
        while True:
            batch_of_images_resized = sess.run(tf_resize_op, {batch_of_images_placeholder:
data[start:end]})
            batch_of_images__preprocessed = preprocess_input(batch_of_images_resized)
            batch_of_labels = labels[start:end]

            start += batch_size
            end   += batch_size
            if start >= n:
                start = 0
                end = batch_size
            yield (batch_of_images__preprocessed, batch_of_labels)
    return generator
```

The summary of model as follows:

```
]: model.summary()
```

block8_10_conv (Conv2D)	(None, None, None, 2	933920	block8_10_mixed[0][0]
block8_10 (Lambda)	(None, None, None, 2	0	block8_9_ac[0][0] block8_10_conv[0][0]
conv_7b (Conv2D)	(None, None, None, 1	3194880	block8_10[0][0]
conv_7b_bn (BatchNormalization)	(None, None, None, 1	4608	conv_7b[0][0]
conv_7b_ac (Activation)	(None, None, None, 1	0	conv_7b_bn[0][0]
avg_pool (GlobalAveragePooling2	(None, 1536)	0	conv_7b_ac[0][0]
predictions (Dense)	(None, 1000)	1537000	avg_pool[0][0]

```
Total params: 55,873,736
Trainable params: 55,813,192
Non-trainable params: 60,544
```

We set batch size = 64, learning rate = 0.001, epochs = 10 and optimizer=SGD.

The results as follows:

```
Epoch 1/10
782/781 [==============================] - 517s 661ms/step - loss: 0.3835
Epoch 2/10
782/781 [==============================] - 514s 657ms/step - loss: 0.3141
Epoch 3/10
782/781 [==============================] - 501s 640ms/step - loss: 0.2883
Epoch 4/10
782/781 [==============================] - 500s 639ms/step - loss: 0.2708
Epoch 5/10
782/781 [==============================] - 508s 650ms/step - loss: 0.2574
Epoch 6/10
782/781 [==============================] - 505s 645ms/step - loss: 0.2465
Epoch 7/10
782/781 [==============================] - 502s 642ms/step - loss: 0.2368
Epoch 8/10
782/781 [==============================] - 502s 643ms/step - loss: 0.2283
Epoch 9/10
782/781 [==============================] - 503s 643ms/step - loss: 0.2203
Epoch 10/10
782/781 [==============================] - 501s 641ms/step - loss: 0.2129
```

And accuracy:

```
10000/10000 [==============================] - 43s 4ms/step
```

```
Out[24]: 0.886
```

So we obtained 88.6% on testing dataset.

Experiment 5

Transfer learning works well because we use a network which is pretrained on a dataset that is similar to the dataset that we are trying to train on. This network has already learnt to recognize the trivial shapes and small parts of different objects in its initial layers. By using a pretrained network to do transfer learning, we are simply adding a few dense layers at the end of the pretrained network and learning what combination of these already learnt features help in recognizing the objects in our new dataset. As such we are training only a few dense layers. We are also using a combination of these already learnt trivial features to recognize new objects. All this helps in making the training process very fast and require very less training data compared to training a convolutional neural network from scratch.

The advantages of transfer learning;

1: You do not need a very big training dataset while using transfer learning
2: Transfer learning also helps lower the cost of training because you do not need a lot of computational power to train a network..As we are using pre-trained weights and only have to learn the weights of the last few layers.

For our experiment we used a Mobilenet model that had already been trained on the imagenet dataset. We discarded the last 1000 layer neuron and added multiple dense layers at the end of the model. This can be done by setting (IncludeTop=False) when importing the model.
We want as many neurons in the last layer of the network as the number of classes we wish to identify. Since we are trying to predict just 10 classes of objects, we change the last layer with a softmax activation function to 10.

The final model we used is shown below;

```
base_model=MobileNet(weights='imagenet',include_top=False)
x=base_model.output
x=GlobalAveragePooling2D()(x)
x=Dense(1024,activation='relu')(x) #we add dense layers so that the model can learn more complex functions and classify for better results.
x=Dense(1024,activation='relu')(x) #dense layer 2
x=Dense(512,activation='relu')(x) #dense layer 3
x=Dense(512,activation='elu')(x) #dense layer 3
x=Dense(512,activation='relu')(x) #dense layer 3
x=Dense(512,activation='relu')(x) #dense layer 3
preds=Dense(10,activation='softmax')(x) #final layer with softmax activation
```

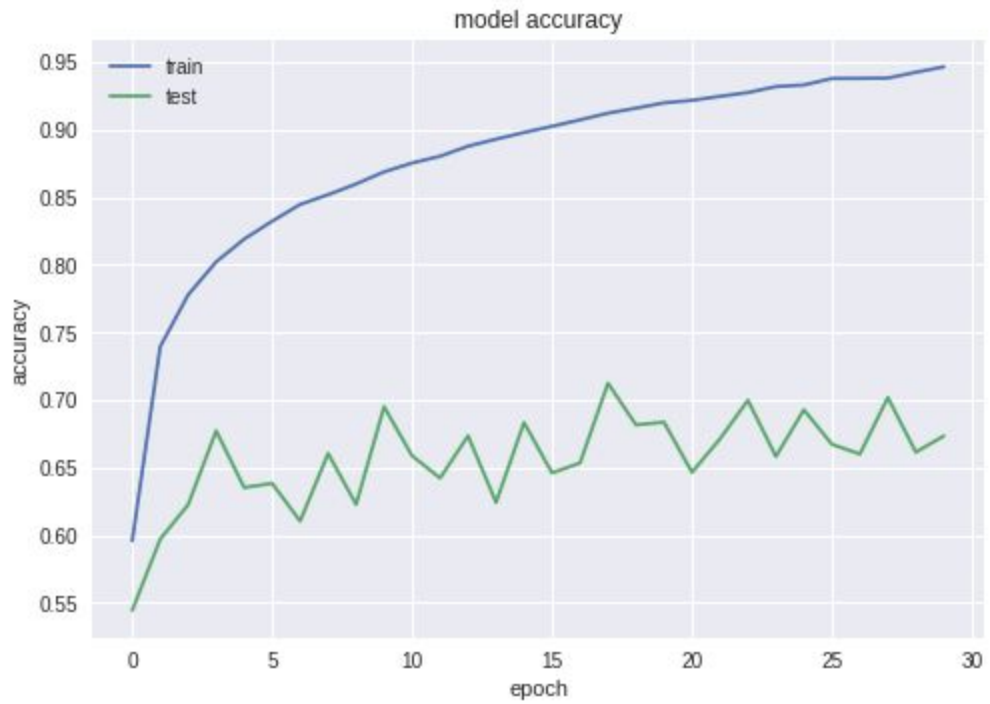
Here is a snapshot of the model summary

```
conv_dw_11 (DepthwiseConv2D) (None, None, None, 512)   4608
conv_dw_11_bn (BatchNormaliz (None, None, None, 512)   2048
conv_dw_11_relu (ReLU)       (None, None, None, 512)   0
conv_pw_11 (Conv2D)          (None, None, None, 512)   262144
conv_pw_11_bn (BatchNormaliz (None, None, None, 512)   2048
conv_pw_11_relu (ReLU)       (None, None, None, 512)   0
conv_pad_12 (ZeroPadding2D)  (None, None, None, 512)   0
conv_dw_12 (DepthwiseConv2D) (None, None, None, 512)   4608
conv_dw_12_bn (BatchNormaliz (None, None, None, 512)   2048
conv_dw_12_relu (ReLU)       (None, None, None, 512)   0
conv_pw_12 (Conv2D)          (None, None, None, 1024)  524288
conv_pw_12_bn (BatchNormaliz (None, None, None, 1024)  4096
conv_pw_12_relu (ReLU)       (None, None, None, 1024)  0
conv_dw_13 (DepthwiseConv2D) (None, None, None, 1024)  9216
conv_dw_13_bn (BatchNormaliz (None, None, None, 1024)  4096
conv_dw_13_relu (ReLU)       (None, None, None, 1024)  0
conv_pw_13 (Conv2D)          (None, None, None, 1024)  1048576
conv_pw_13_bn (BatchNormaliz (None, None, None, 1024)  4096
conv_pw_13_relu (ReLU)       (None, None, None, 1024)  0
global_average_pooling2d_1 ( (None, 1024)              0
```

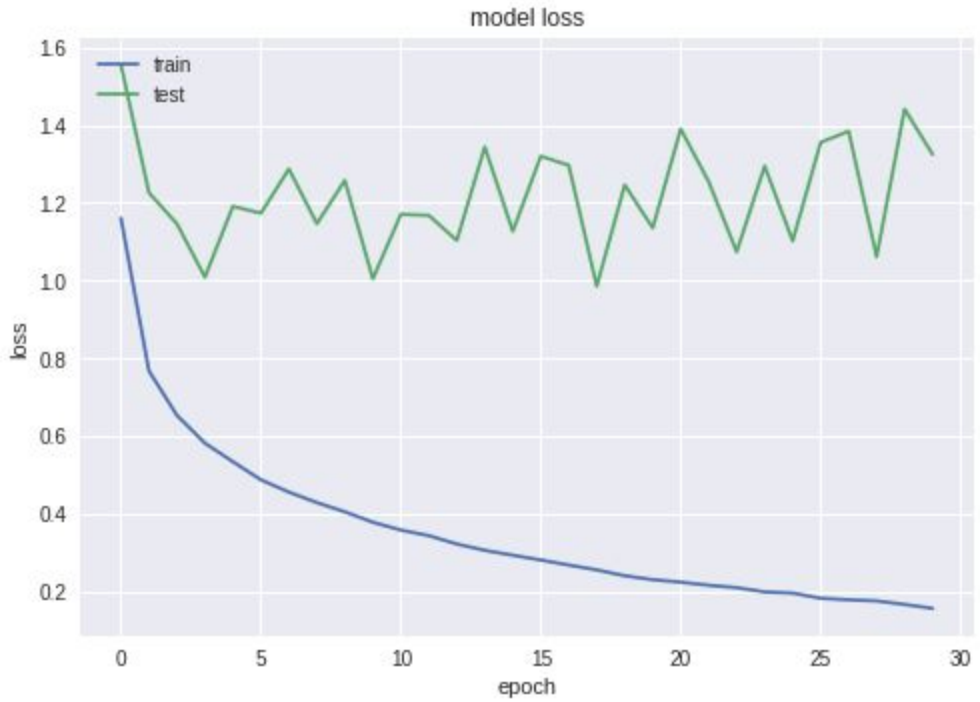
```
step_size_train=train_generator.n//train_generator.batch_size
history = model.fit_generator(generator=train_generator,
                  steps_per_epoch=step_size_train,
                  epochs=epochs, validation_data=(x_test, y_test))
```

We used a stochastic gradient descent optimizer and a learning rate of 0.01 with a batch size of 128 and 30 epochs. We used accuracy as the evaluation metric, the best accuracy we got on validation data was 67%

The graph below shows accuracy plotted against training epochs for both the training and testing results.



Although the model loss kept on going down for the training dataset, it remained in the range of 1.4 and 1 for the testing dataset.
Another important point to note is that the model took a significantly short time to train, compared to the amount of time needed to train the from scratch



Experiment 6

The task of fine-tuning a network is to tweak the parameters of an already trained network so that it adapts to the new task at hand. Thus, for fine-tuning, we want to keep the initial layers intact (or freeze them) and retrain the later layers for our task.

In this part, we present the process of fine-tuning the InceptionResNetV2 network.
Before we compile our model, we need to train only the top layers (which were randomly initialized).For example, freeze all convolutional layers:

```
for layer in base_model.layers:
    layer.trainable = False
```

Then we validate top layers:

```
result = model.predict(images, verbose=1)
y_pred = [ np.argmax( result[i] ) for i in range(n_testing) ]
np.sum( y_pred == y_test ) / n_testing
```

```
10000/10000 [==============================] - 38s 4ms/step
0.7323
```

Then, we train more layers. At this point, the top layers are well trained and we can start fine-tuning. Convolutional layers from InceptionResNetV2. We will freeze the bottom N layers and train the remaining top layers:

```
for layer in model.layers[:16]:
    layer.trainable = False
for layer in model.layers[16:]:
    layer.trainable = True
```

```
Epoch 1/10
782/781 [==============================] - 451s 576ms/step - loss: 0.2688
Epoch 2/10
782/781 [==============================] - 447s 572ms/step - loss: 0.0647
Epoch 3/10
782/781 [==============================] - 446s 570ms/step - loss: 0.0170
Epoch 4/10
782/781 [==============================] - 453s 580ms/step - loss: 0.0059
Epoch 5/10
782/781 [==============================] - 455s 581ms/step - loss: 0.0029
Epoch 6/10
782/781 [==============================] - 455s 582ms/step - loss: 0.0018
Epoch 7/10
782/781 [==============================] - 456s 584ms/step - loss: 0.0011
```

Epoch 8/10
782/781 [==============================] - 451s 577ms/step - loss: 7.8778e-04
Epoch 9/10
782/781 [==============================] - 466s 596ms/step - loss: 6.1613e-04
Epoch 10/10
782/781 [==============================] - 452s 578ms/step - loss: 5.0233e-04

10000/10000 [==============================] - 43s 4ms/step
0.9383

So we obtained 93.83% on testing dataset, which is higher than the result of experiment 4.

Summary of experiments:

Models or Methods	Accuracy(%)
Keras(network designed by group 6)	12.67%
Autokeras	
Alexnet	
InceptionResNetV2(pre-trained)	88.6%
InceptionResNetV2(with Transfer learning)	67%
InceptionResNetV2(with Fine tuning)	93.83%

Conclusion:
The network designed by group gains low accuracy because our network structure is simple. The increase of accuracy isn't in direct ratio with the increase of layers as test in our experiment 1. In order to get higher accuracy, we can use some well-designed and pre-trained model as our base model and adding several layers and functions to fit our dataset. In our research, InceptionResNetV2 is already a good model to use. Furthermore, by using transfer learning and fine tuning, we can transferring the learnt representations to another problem and making fine adjustment to our model thus improve performances.