Course Project on Deep Learning and Reinforcement Learning

Main objective of the analysis

This project is focused on deep learning. Its main objective is to develop an image classifier to classify images of horses and zebras. This classifier should have as high accuracy as possible.

It seems that images this pair of animals (horses and zebras) are mostly used for training GAN models, while most commonly used animal pairs for classifiers are cats and dogs. But classification of horses and zebras is also an interesting task because they are related animals and can be mixed producing offspring. The main outward difference between them is that zebras have black and white stripes, which horses do not have. But depending on an image perspective, zebra's stripes might not be visible, and then a zebra and a horse might be confused.

In this project, I used two approaches to building deep learning model: 1) I built CNN model from scratch, 2) I also used transfer learning and used several pre-trained models that are available in Keras. I did tuning hyper-parameters with both approaches and then chose the model that had the best performance.

Brief description of the data set and a summary of its attributes

I used images of horses and zebras from three different data sets:

1. Data set "horse2zebra" from "cycle_gan" which is one of TensorFlow data sets. It contains: 1) 1,067 images of horses for training, 2) 1,334 images of zebras for training, 3) 120 images of horses for testing, 4) 140 images of zebras for training.

2. Data set "horse" available from image.cv website. It contains: 1) 709 images of horses for training, 2) 124 images of horses for validation, 3) 363 images of horses for testing.

3. Data set "zebra" which is also available from image.cv website. It contains: 1) 790 images of zebras for training, 2) 130 images of zebras for validation, 3) 380 images of zebras for testing.

I used these data sets in two combinations, depending on how many images I would like to use for training:

1st variant: 1) for training – images of horses and zebras from the folders "train" of the data set "horse2zebra", 2) for validation – images from the folders "test" of the data set "horse2zebra", 3) for testing – images of horses and zebras from the folders "test" of the data sets "horse" and "zebras".

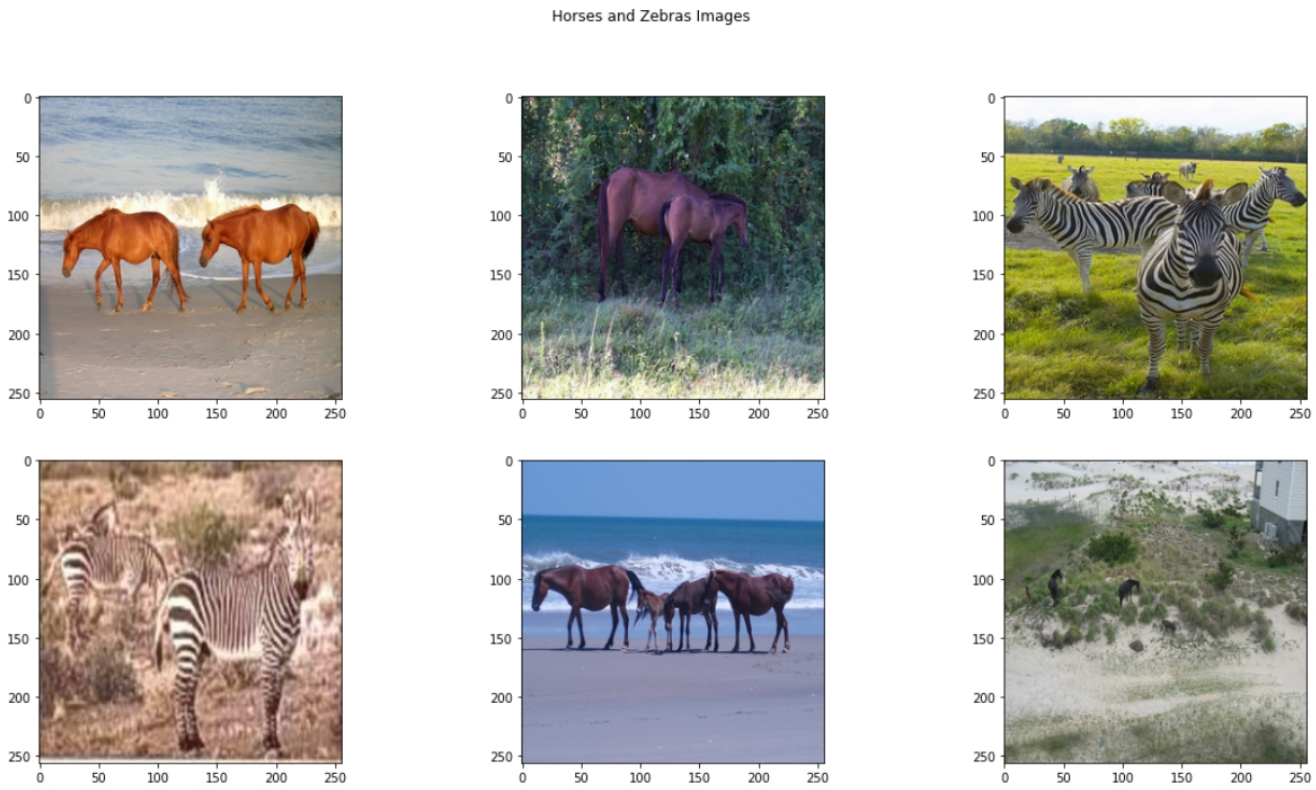
2nd variant: 1) for training – I added images of horses and zebras from the folders "train" of the data sets "horse" and "zebra", 2) for validation – I added images from the folders "val" of the same data sets, 3) for testing – I used the same images as in the 1st variant.

Therefore, I used the same images for testing in all the cases, but the number of images for training and validation was sometimes different.

Brief summary of data exploration and actions taken for data cleaning and feature engineering

Since the data sets used in this projects consist only of images, the usual process of EDA is not applicable here.

Some of the images from “horse2zebra” data set:



There was no corrupted images in these data sets and therefore there was no need in data cleaning.

As for feature engineering, I used various options with ImageDataGenerator, such as image rescaling (normalization), image resizing (which was needed for transfer learning because pre-trained models were trained with 224x224 images, but the images in the data sets that I used are 256x256), image augmentation, and some others.

In [23]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
      rescale=1./255
)

train_generator = train_datagen.flow_from_directory(
        'horse2zebra/training',
        target_size=(256, 256),
        batch_size=128,
        class_mode='categorical')
```

Summary of training deep learning classification models

1. Building CNN from scratch

CNN model has 5 convolution layers, 5 pooling layers, 1 flatten layer, and 2 dense layers. It has two outputs for 2 classes.

In [24]:
```
model = tf.keras.models.Sequential([
    # The input shape is the size of the images 256x256 with 3 bytes color
    # This is the first convolution
    tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(256, 256, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The third convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fourth convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fifth convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # Flatten the results to feed into a DNN
    tf.keras.layers.Flatten(),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    # 2 output neurons - for 2 classes
    tf.keras.layers.Dense(2, activation='softmax')
])
```

CNN model summary:

```
In [26]: model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 254, 254, 16)      448

 max_pooling2d_10 (MaxPoolin  (None, 127, 127, 16)     0
 g2D)

 conv2d_11 (Conv2D)          (None, 125, 125, 32)      4640

 max_pooling2d_11 (MaxPoolin  (None, 62, 62, 32)       0
 g2D)

 conv2d_12 (Conv2D)          (None, 60, 60, 64)        18496

 max_pooling2d_12 (MaxPoolin  (None, 30, 30, 64)       0
 g2D)

 conv2d_13 (Conv2D)          (None, 28, 28, 64)        36928

 max_pooling2d_13 (MaxPoolin  (None, 14, 14, 64)       0
 g2D)

 conv2d_14 (Conv2D)          (None, 12, 12, 64)        36928

 max_pooling2d_14 (MaxPoolin  (None, 6, 6, 64)         0
 g2D)

 flatten_2 (Flatten)         (None, 2304)              0

 dense_4 (Dense)             (None, 512)               1180160

 dense_5 (Dense)             (None, 2)                 1026

=================================================================
Total params: 1,278,626
Trainable params: 1,278,626
Non-trainable params: 0
_________________________________________________________________
```

I experimented with tuning hyper-parameters of this model by changing the following:

1) number of training examples: 2401 and 3899,

2) number of epochs: 15 and 100,

3) using image augmentation, which included:

- Rotation
- Shifting horizontally

- Shifting vertically
- Shearing
- Zooming
- Flipping

```
# All images will be augmented
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

I always used Adam optimizer and categorical cross entropy loss.

```
In [27]: model.compile(loss='categorical_crossentropy',
                      optimizer='adam',
                      metrics=['acc'])
```

After the model was trained, I evaluated it using test set which was not used during the training. The resulting table:

CNN model	Loss	Accuracy
training examples = 2401, epochs = 15, no augmentation	51.618988	0.846361
training examples = 3899, epochs = 15, no augmentation	18.367886	0.916442
training examples = 3899, epochs = 15, with augmentation	109.247787	0.746631
training examples = 3899, epochs = 100, no augmentation	35.793457	0.946092
training examples = 3899, epochs = 100, with augmentation	15.772952	0.950135

Even with training examples = 2401 and epochs = 15, the accuracy of the model was higher than 0.84, which is not so bad. Increasing the number of training examples to 3899 helped to increase accuracy to higher than 0.91. Increasing the number of epochs to 100 increased accuracy to over 0.94. And using image augmentation increased accuracy to 0.95.

However, it was somewhat strange that image augmentation with epochs = 15 decreased accuracy to 0.74, although it did increase accuracy with epochs = 100.

The highest accuracy reached with CNN model built from scratch was 0.95 with training examples = 3899, epochs = 100, and image augmentation. Although this accuracy was already quite high, I also experimented with pre-trained models available through Keras.

2. Transfer learning

I used 7 pre-trained models:

- ResNet50
- ResNet101
- ResNet152
- VGG16
- VGG19
- InceptionV3
- InceptionResNetV2

ResNet50:

```
In [20]: model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet50 (Functional)        (None, 2048)              23587712
_________________________________________________________________
dense (Dense)                (None, 2)                 4098
=================================================================
Total params: 23,591,810
Trainable params: 4,098
Non-trainable params: 23,587,712
_________________________________________________________________
```

ResNet101:

```
In [16]: model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet101 (Functional)       (None, 2048)              42658176
_________________________________________________________________
dense (Dense)                (None, 2)                 4098
=================================================================
Total params: 42,662,274
Trainable params: 4,098
Non-trainable params: 42,658,176
_________________________________________________________________
```

ResNet152:

```
In [14]: model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet152 (Functional)       (None, 2048)              58370944
_________________________________________________________________
dense (Dense)                (None, 2)                 4098
=================================================================
Total params: 58,375,042
Trainable params: 4,098
Non-trainable params: 58,370,944
_________________________________________________________________
```

VGG16:

```
In [11]: model.layers[0].trainable = False
         model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 512)               14714688
_________________________________________________________________
dense (Dense)                (None, 2)                 1026
=================================================================
Total params: 14,715,714
Trainable params: 1,026
Non-trainable params: 14,714,688
_________________________________________________________________
```

VGG19:

```
In [11]: model.layers[0].trainable = False
         model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg19 (Functional)           (None, 512)               20024384
_________________________________________________________________
dense (Dense)                (None, 2)                 1026
=================================================================
Total params: 20,025,410
Trainable params: 1,026
Non-trainable params: 20,024,384
_________________________________________________________________
```

InceptionV3:

```
In [13]: model.layers[0].trainable = False
         model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_v3 (Functional)    (None, 2048)              21802784
_________________________________________________________________
dense (Dense)                (None, 2)                 4098
=================================================================
Total params: 21,806,882
Trainable params: 4,098
Non-trainable params: 21,802,784
```

InceptionResNetV2:

```
In [11]: model.layers[0].trainable = False
         model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_resnet_v2 (Functio (None, 1536)              54336736
_________________________________________________________________
dense (Dense)                (None, 2)                 3074
=================================================================
Total params: 54,339,810
Trainable params: 3,074
Non-trainable params: 54,336,736
_________________________________________________________________
```

As can be seen from screenshots of model summaries, I added to them one output layer with 2 neurons – for 2 classes. The activation function is softmax.

First, I trained all the models with the same parameters:

1) training examples = 2401,

2) epochs = 2

3) no image augmentation,

4) Adam optimizer,

5) cross entropy loss.

The table of results for all the models:

Model	Loss	Accuracy
resnet50	0.024689	0.990566
resnet101	0.022430	0.993261
resnet152	0.036423	0.986523
vgg16	0.336455	0.921833
vgg19	0.285416	0.923181
inception_v3	0.039829	0.985175
inception_resnet_v2	0.016802	0.993261

ResNet101 and InceptionResNetV2 has the highest accuracy. However, since InceptionResNetV2 has a lower loss, I chose this model.

Then, I trained it with different hyper-parameters, like I trained the first CNN model:

1) number of training examples: 2401 and 3899,

2) number of epochs: 2, 10, and 40,

3) using image augmentation.

In all the cases, I used Adam optimizer, as usually.

Here is the table of the results:

Model	Loss	Accuracy
inception_resnet_v2 (training examples = 2401, no augmentation, epochs = 2)	0.016802	0.993261
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 2)	0.014949	0.993261
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 10)	0.004592	1.000000
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 40)	0.002143	1.000000
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 2)	0.425450	0.824798
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 10)	3.140969	0.535040
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 40)	14.981770	0.525606

As can be seen from the table, when I increased the number of testing examples from 2401 to 3899, accuracy was not changed, but loss decreased a little bit. But when I increased the number of epochs from 2 to 10, accuracy reached 1 (100%) and loss significantly decreased. And when I increased the number of epochs to 40, loss decreased even more.

100% accuracy can be considered as a perfect result. The model showed this result on the testing images which it did not see during the training. So, it was not overfitting.

However, with this model, image augmentation not only did not help to increase accuracy, but, on the contrary, it decreased it. And when I increased the number of epochs, accuracy decreased even more. I do not have an explanation for this effect.

Out of interest, I tested the model with image augmentation, that is, I used image augmentation with the test generator like this:

In [3]:
```
data_generator = ImageDataGenerator(
    preprocessing_function=preprocess_input,
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

In [4]:
```
test_generator = data_generator.flow_from_directory(
    'horse2zebra/testing',
    target_size=(image_resize, image_resize),
    shuffle=False)
```

I received the following results:

Model (testing with image augmentation)	Loss	Accuracy
inception_resnet_v2 (training examples = 2401, no augmentation, epochs = 2)	0.708961	0.512129
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 2)	0.725235	0.487871
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 10)	0.757049	0.487871
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 40)	0.942501	0.487871
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 2)	0.674945	0.578167
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 10)	0.622732	0.708895
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 40)	0.547358	0.749326

These results look more understandable. When the model was trained without image augmentation, it seemed to be confused by the augmented images at the testing. But when it was trained with image augmentation, it showed better results, and the longer it was trained, the higher the accuracy was achieved.

These results seem to indicate that if the model did not see images with various rotations, upside down images, and images with various other effects, it will be confused by such images.

The highest accuracy achieved at the augmented images was about 0.75 (for image augmentation during the training with epochs = 40). This is not very high accuracy. And the same model had accuracy only about 0.52 at non-augmented images, which is very bad.

Probably, it is not so important that a model would have a good performance on the augmented images because upside down images of horses and zebras and images with other special effects are not very common.

Recommended final model

First, I would like to briefly review the results of testing of different models.

1. CNN model built from scratch:

CNN model	Loss	Accuracy
training examples = 2401, epochs = 15, no augmentation	51.618988	0.846361
training examples = 3899, epochs = 15, no augmentation	18.367886	0.916442
training examples = 3899, epochs = 15, with augmentation	109.247787	0.746631
training examples = 3899, epochs = 100, no augmentation	35.793457	0.946092
training examples = 3899, epochs = 100, with augmentation	15.772952	0.950135

2. Transfer learning with pre-trained models:

Model	Loss	Accuracy
resnet50	0.024689	0.990566
resnet101	0.022430	0.993261
resnet152	0.036423	0.986523
vgg16	0.336455	0.921833
vgg19	0.285416	0.923181
inception_v3	0.039829	0.985175
inception_resnet_v2	0.016802	0.993261

3. Transfer learning with InceptionResNetV2:

Model	Loss	Accuracy
inception_resnet_v2 (training examples = 2401, no augmentation, epochs = 2)	0.016802	0.993261
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 2)	0.014949	0.993261
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 10)	0.004592	1.000000
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 40)	0.002143	1.000000
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 2)	0.425450	0.824798
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 10)	3.140969	0.535040
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 40)	14.981770	0.525606

The best results were reached for transfer learning with InceptionResNetV2 for training examples = 3899, epochs = 40, and no augmentation. Its accuracy on the test images (which were not used during training) reached 100%. Therefore, this is the model that can be recommended.

Summary Key Findings and Insights

The goal of this project was to develop a deep learning model for classification of images of horses and zebras.

First, I built a CNN model from scratch with 5 convolution layers. This model, trained on training examples = 3899, epochs = 100 and with image augmentation, reached 95% accuracy, which was a good result.

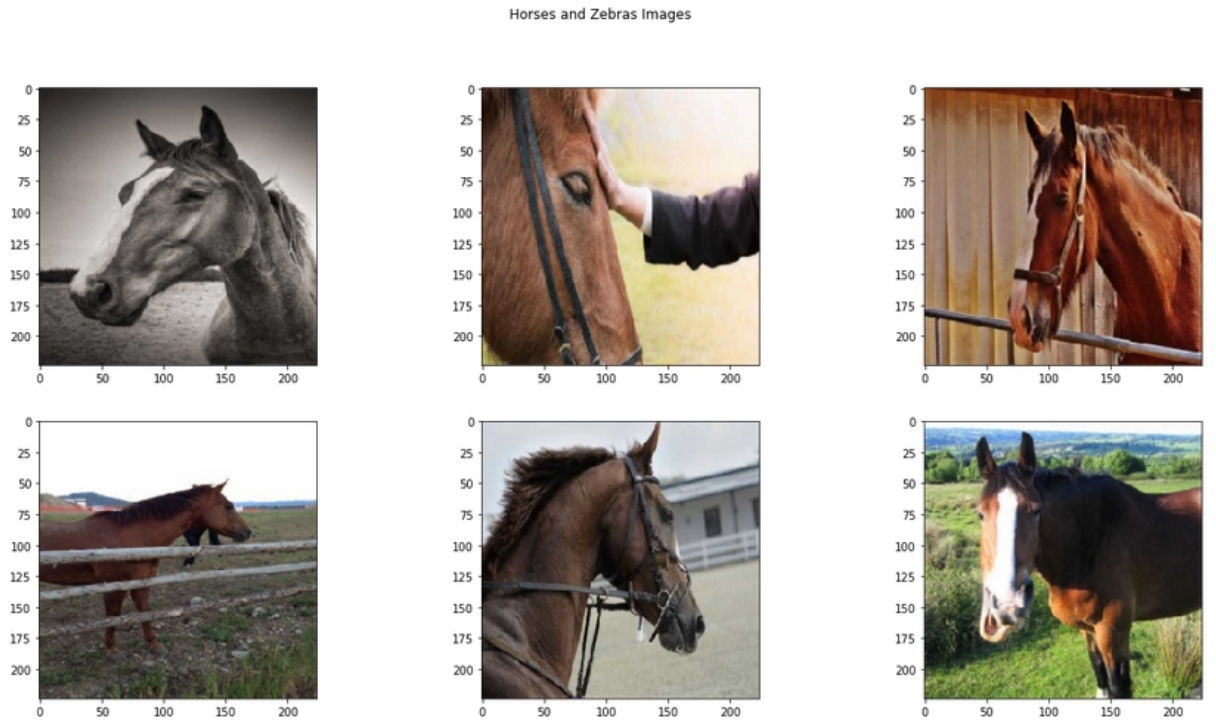
Then, I used transfer learning with pre-trained models. As can be expected, this helped to reach even better accuracy. Since InceptionResNetV2 had the highest accuracy and lowest loss among the 7 pre-trained models, I used this model.

Then, I trained it with different parameters and it gave the best results on training examples = 3899, epochs = 40 and without image augmentation. It reached 100% accuracy on testing images which it did not see during training.

However, there was some problems with image augmentation: when this modelwas trained with image augmentation it had much lower accuracy, and its accuracy decreased even more when the number of epochs was increased.

Testing the model:

```
fig.suptitle('Horses and Zebras Images')
plt.show()
```




```
In [21]: # Printing class predictions for the first 6 images
for i in range(0,6):
  if (predictions[i][0] >= 0.5):
    print("Horse")
  elif (predictions[i][1] >= 0.5):
    print("Zebra")
```

```
Horse
Horse
Horse
Horse
Horse
Horse
```

As can be seen, the first 6 testing images were images of horses, and the model correctly identified them as horses.

This shows visually that the model indeed works correctly.

Suggestions for next steps in analyzing this data

Also the best model achieved the 100% accuracy, it seems that there is still a need to analyze its performance on augmented images.

The table of testing results of the model based on InceptionResNetV2 looks like this:

Model	Loss	Accuracy
inception_resnet_v2 (training examples = 2401, no augmentation, epochs = 2)	0.016802	0.993261
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 2)	0.014949	0.993261
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 10)	0.004592	1.000000
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 40)	0.002143	1.000000
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 2)	0.425450	0.824798
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 10)	3.140969	0.535040
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 40)	14.981770	0.525606

When this model was trained with image augmentation, it caused significant decrease of its accuracy. And this decrease became even more significant with the increase of the number of epochs. The model trained on epochs = 40 without augmentation reached 100% accuracy, but when it was trained on the same number of epochs, its accuracy was only about 52%. This is very strange and needs to be analyzed.

On the other hand, the usage of image augmentation during the testing gave the following results:

Model (testing with image augmentation)	Loss	Accuracy
inception_resnet_v2 (training examples = 2401, no augmentation, epochs = 2)	0.708961	0.512129
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 2)	0.725235	0.487871
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 10)	0.757049	0.487871
inception_resnet_v2 (training examples = 3899, no augmentation, epochs = 40)	0.942501	0.487871
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 2)	0.674945	0.578167
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 10)	0.622732	0.708895
inception_resnet_v2 (training examples = 3899, with augmentation, epochs = 40)	0.547358	0.749326

Although performance of the model trained with image augmentation increased with the number of epochs when it was tested on augmented images (which it did not see during the training), but the

accuracy of the model trained without image augmentation with epochs = 40 was only about 48% when it was tested on augmented images, although its accuracy was 100% when it was tested on non-augmented images.

So, there seems to be a need to analyze more carefully how image augmentation during training and testing influences performance of the model.

The best result achieved on testing with image augmentation was about 75% which was not very high.

Ideally, a model should have a high accuracy when tested on both non-augmented and augmented images, but this result was not achieved in this project. So, it can be good to continue to work in this direction, experimenting with different models, image data sets, and tuning hyper-parameters.