Training a Dog Breed Classifier using Transfer Learning

The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

- Motivation
- Step 1: Import Dog Datasets
- Step 2: Pre-process the Data
- Step 3: Create a Model using Transfer Learning
- Step 4: Fine-tune the Model
- Conclusion

Motivation

In this notebook we want to use transfer learning to train a neural network that classifies dog breeds. This means we will take a neural network that has already learned weights for a related problem and adjust it to work for our problem. As our base network we will use the InceptionV3 (https://keras.io/api/applications/inceptionv3/) model with its ImageNet (http://www.image-net.org/) weights. ImageNet is a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories. 118 of theses categories are related to dog breeds.

For our problem we will have 8351 images of dogs to learn 133 categories/breeds from, overall. Which really is not that much.

It is worth to mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have great difficulty in distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany **Welsh Springer Spaniel**

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever **American Water Spaniel**

Likewise, recall that labradors come in yellow, chocolate, and black. Our vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador **Chocolate Labrador** **Black Labrador**

It may also be mentioned that random chance presents an exceptionally low bar (for our dataset): setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of ~0.75%.

Step 1: Import Dog Dataset

In [1]:
```python
from pathlib import Path
from typing import Tuple

import numpy as np
from sklearn.datasets import load_files
from keras import utils


def load_dataset(path: Path) -> Tuple[np.ndarray, np.ndarray]:
    """
    Loads the data given at path into source and target vectors using sklearn.datasets.load_files.
    :param path: Path to load data from.
    :return: the file names (representing the source vectors) and the target vector.
    """
    data = load_files(str(path))
    dog_files = np.array(data['filenames'])
    dog_targets = utils.to_categorical(np.array(data['target']), num_classes=133)
    return dog_files, dog_targets


dog_images_path = Path('../../..') / 'data' / 'dog_images'
train_path = dog_images_path / 'train'

train_files, train_targets = load_dataset(train_path)
valid_files, valid_targets = load_dataset(dog_images_path / 'valid')
test_files, test_targets = load_dataset(dog_images_path / 'test')


print('There are %d total dog categories.' % len(list(train_path.iterdir())))
print('There are %s total dog images.\n' % len(np.hstack([train_files, valid_files, test_files])))
print('There are %d training dog images.' % len(train_files))
print('There are %d validation dog images.' % len(valid_files))
print('There are %d test dog images.'% len(test_files))
```

```
Using TensorFlow backend.

There are 133 total dog categories.
There are 8351 total dog images.

There are 6680 training dog images.
There are 835 validation dog images.
There are 836 test dog images.
```

Step 2: Pre-process the Data

Step 2.1: Image Path to Tensor

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape

$$(\text{nb\_samples}, \text{rows}, \text{columns}, \text{channels}),$$

where `nb_samples` corresponds to the total number of images (or samples), and `rows`, `columns`, and `channels` correspond to the number of rows, columns, and channels for each image, respectively.

The `path_to_tensor` function below takes a string-valued file path to a color image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The function first loads the image and resizes it to a square image that is 224×224 pixels. Next, the image is converted to an array, which is then resized to a 4D tensor. In this case, since we are working with color images, each image has three channels. Likewise, since we are processing a single image (or sample), the returned tensor will always have shape

$$(1, 224, 224, 3).$$

The `paths_to_tensor` function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

$$(\text{nb\_samples}, 224, 224, 3).$$

Here, `nb_samples` is the number of samples, or number of images, in the supplied array of image paths. It is best to think of `nb_samples` as the number of 3D tensors (where each 3D tensor corresponds to a different image) in your dataset!

In [2]:
```python
from typing import Iterable

from keras.applications.inception_v3 import preprocess_input
from keras.preprocessing import image
from PIL import ImageFile
from tqdm import tqdm

ImageFile.LOAD_TRUNCATED_IMAGES = True


def path_to_tensor(img_path: str) -> np.ndarray:
    """
    Converts the image given by img_path into a 4D-tensor with shape (1, 224, 224, 3).
    :param img_path: path as str to the image to convert in a tensor.
    :return: the 4D-tensor of the image.
    """
    img = image.load_img(img_path, target_size=(224, 224))
    x = image.img_to_array(img)
    return np.expand_dims(x, axis=0)


def paths_to_tensor(img_paths: Iterable[str]) -> np.ndarray:
    """
    Converts all images given by img_paths into 4D-tensor with shape (1, 224, 224, 3)
        and stacks them vertically.
    :param img_paths: paths to the images to convert into tensors.
    :return: a 4D-tensor with shape (num_samples, 224, 224, 3).
    """
    list_of_tensors = [path_to_tensor(img_path) for img_path in tqdm(img_paths)]
    return np.vstack(list_of_tensors)


# We rescale the images by dividing every pixel in every image by 255
train_tensors = paths_to_tensor(train_files).astype('float32')/255
valid_tensors = paths_to_tensor(valid_files).astype('float32')/255
test_tensors = paths_to_tensor(test_files).astype('float32')/255

# Apply InceptionV3-specific pre-processing
train_tensors = preprocess_input(train_tensors)
valid_tensors = preprocess_input(valid_tensors)
test_tensors = preprocess_input(test_tensors)
```

```
100%|██████████| 6680/6680 [01:11<00:00, 93.68it/s]
100%|██████████| 835/835 [00:07<00:00, 105.84it/s]
100%|██████████| 836/836 [00:07<00:00, 107.05it/s]
```

Step 2.2: Data Augmentation

Even though we use transfer learning and therefore reuse data that the InceptionV3 network originally was trained on, our amount of training data with 6680 images for 133 categories in the training set is still very low. We try to compensate this with data augmentation. This means, we will not always feed the same 6680 images to our network but variants of them. For example, we rotate images, zoom in, take only parts or flip them. This increases the variance in our training set, prevents overfitting and gives the network more samples to learn from. We achieve data augmentation via the very handy keras ImageDataGenerator.

In [3]:
```python
from keras.preprocessing.image import ImageDataGenerator


BATCH_SIZE = 32

train_datagen = ImageDataGenerator(
        rotation_range=60,
        width_shift_range=0.4,
        height_shift_range=0.4,
        shear_range=0.4,
        zoom_range=0.4,
        horizontal_flip=True,
        fill_mode='nearest')

train_generator = train_datagen.flow(
    x=train_tensors,
    y=train_targets,
    batch_size=BATCH_SIZE)


test_datagen = ImageDataGenerator()

validation_generator = test_datagen.flow(
    x=valid_tensors,
    y=valid_targets,
    batch_size=BATCH_SIZE)
```

Step 3: Create a Model using Transfer Learning

Step 3.1: Create Model Architecture

As stated above we will reuse the InceptionV3 with its ImageNet weights here as our base model. If we load it with `include_top=False` it means that its last block, where the tensors get classified into one of 1000 ImageNet categories, is not loaded. Instead we will add our own classification block on top. An important thing to notice here is that we freeze the weights of the base model and just train our classification block. If we would not do that the base model would "forget" a lot of its learned weights due to the high loss at the start of training because of the untrained new classification block.

In [4]:
```
from keras.applications.inception_v3 import InceptionV3
from keras.layers import Dense, GlobalAveragePooling2D, Dropout
from keras.models import Model


base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
x = base_model.output

# Add our classification block
x = GlobalAveragePooling2D()(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.4)(x)
predictions = Dense(133, activation='softmax')(x)

# Combine base model and our classification block into one
model = Model(inputs=base_model.input, outputs=predictions)

# freeze all convolutional InceptionV3 layers
for layer in base_model.layers:
    layer.trainable = False

# compile the model (should be done after setting layers to non-trainable)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.5/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5 (https://github.com/fchollet/deep-learning-models/releases/download/v0.5/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5)
87916544/87910968 [==============================] - 1s 0us/step
```

Step 3.2: Train the Model

We train our classification block for 12 epochs with our augmented 6680 images in the training set. The results are very impressive already as we achieve a validation accuracy of ~68% with that little data.

In [5]:

```
from keras.callbacks import ModelCheckpoint


# The use of model checkpointing will only save the model that attains the best validation loss.
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.transfer_learning_model.hdf5',
                               verbose=1,
                               save_best_only=True)


NB_TRAIN_SAMPLES = train_tensors.shape[0]
NB_VALID_SAMPLES = valid_tensors.shape[0]


history = model.fit_generator(train_generator,
                   steps_per_epoch=NB_TRAIN_SAMPLES // BATCH_SIZE,
                   epochs=12,
                   validation_data=validation_generator,
                   validation_steps=NB_VALID_SAMPLES // BATCH_SIZE,
                   callbacks=[checkpointer],
                   verbose=1)
```

```
Epoch 1/12
207/208 [============================>.] - ETA: 0s - loss: 4.6549 - acc: 0.0576Epoch 00001: val_loss improved from inf to 4.88
771, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 65s 311ms/step - loss: 4.6529 - acc: 0.0579 - val_loss: 4.8877 - val_acc: 0.0108
Epoch 2/12
207/208 [============================>.] - ETA: 0s - loss: 3.7949 - acc: 0.1698Epoch 00002: val_loss improved from 4.88771 to
2.60335, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 281ms/step - loss: 3.7929 - acc: 0.1701 - val_loss: 2.6034 - val_acc: 0.4195
Epoch 3/12
207/208 [============================>.] - ETA: 0s - loss: 3.3520 - acc: 0.2226Epoch 00003: val_loss improved from 2.60335 to
1.60253, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 281ms/step - loss: 3.3508 - acc: 0.2229 - val_loss: 1.6025 - val_acc: 0.6058
Epoch 4/12
207/208 [============================>.] - ETA: 0s - loss: 3.1149 - acc: 0.2572Epoch 00004: val_loss improved from 1.60253 to
1.36168, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 281ms/step - loss: 3.1152 - acc: 0.2571 - val_loss: 1.3617 - val_acc: 0.6322
Epoch 5/12
207/208 [============================>.] - ETA: 0s - loss: 2.9926 - acc: 0.2767Epoch 00005: val_loss improved from 1.36168 to
1.23275, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 281ms/step - loss: 2.9937 - acc: 0.2769 - val_loss: 1.2327 - val_acc: 0.6562
Epoch 6/12
207/208 [============================>.] - ETA: 0s - loss: 2.9001 - acc: 0.2931Epoch 00006: val_loss improved from 1.23275 to
1.17304, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 280ms/step - loss: 2.9009 - acc: 0.2929 - val_loss: 1.1730 - val_acc: 0.6623
Epoch 7/12
207/208 [============================>.] - ETA: 0s - loss: 2.8498 - acc: 0.3088Epoch 00007: val_loss improved from 1.17304 to
1.14078, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 280ms/step - loss: 2.8512 - acc: 0.3085 - val_loss: 1.1408 - val_acc: 0.6647
Epoch 8/12
207/208 [============================>.] - ETA: 0s - loss: 2.7779 - acc: 0.3143Epoch 00008: val_loss improved from 1.14078 to
1.11666, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 281ms/step - loss: 2.7784 - acc: 0.3143 - val_loss: 1.1167 - val_acc: 0.6635
Epoch 9/12
207/208 [============================>.] - ETA: 0s - loss: 2.7937 - acc: 0.3157Epoch 00009: val_loss improved from 1.11666 to
1.11324, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 280ms/step - loss: 2.7918 - acc: 0.3162 - val_loss: 1.1132 - val_acc: 0.6755
Epoch 10/12
207/208 [============================>.] - ETA: 0s - loss: 2.7800 - acc: 0.3190Epoch 00010: val_loss improved from 1.11324 to
1.06686, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 280ms/step - loss: 2.7780 - acc: 0.3193 - val_loss: 1.0669 - val_acc: 0.6863
Epoch 11/12
207/208 [============================>.] - ETA: 0s - loss: 2.7353 - acc: 0.3291Epoch 00011: val_loss improved from 1.06686 to
1.06210, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 280ms/step - loss: 2.7362 - acc: 0.3289 - val_loss: 1.0621 - val_acc: 0.6827
Epoch 12/12
207/208 [============================>.] - ETA: 0s - loss: 2.7880 - acc: 0.3277Epoch 00012: val_loss improved from 1.06210 to
1.05781, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 58s 281ms/step - loss: 2.7918 - acc: 0.3271 - val_loss: 1.0578 - val_acc: 0.6767
```

Step 3.3: Plot Training Progress

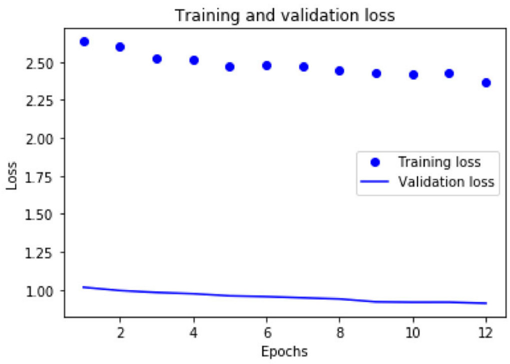
In [14]:
```
from keras.callbacks import History
from matplotlib import pyplot as plt


def plot_history(history: History) -> None:
    history_dict = history.history
    loss_values = history_dict['loss']
    val_loss_values = history_dict['val_loss']
    epochs = range(1, len(loss_values) + 1)

    plt.plot(epochs, loss_values, 'bo', label='Training loss')
    plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
    plt.title('Training and validation loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.show()


plot_history(history)
```



Step 3.4: Test the Model

In [7]:
```
model.load_weights('saved_models/weights.best.transfer_learning_model.hdf5')
```

In [8]:
```
def evaluate_model(model: Model, test_tensors: np.ndarray, test_targets: np.ndarray) -> float:
    """
    Evaluate the given model on the given test_tensors with the given test_targets.
    :param model: model to evaluate.
    :param test_tensors: tensors the model makes predictions for.
    :param test_targets: target vector to compare the model's predictions with.
    :return: the computed test accuracy.
    """
    # get index of predicted dog breed for each image in test set
    dog_breed_predictions = [np.argmax(model.predict(np.expand_dims(tensor, axis=0))) for tensor in test_tensors]

    # report test accuracy
    return 100 * np.sum(np.array(dog_breed_predictions) == np.argmax(test_targets, axis=1)) / len(dog_breed_predictions)


test_accuracy = evaluate_model(model, test_tensors, test_targets)

print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 65.1914%
```

Step 4: Fine-tune the Model

So far we only trained our new classification block. As its weights are now already well trained and not random anymore, we can start to unfreeze parts of our base model to improve our model's performance even further. So here we will unfreeze the top inception block (layer 280 and above) and learn again on our training set. For training we will use a very low learning rate as do not want to destroy already learned weights of the base model but just want to optimize them.

Step 4.1: Unfreeze Parts of Base Model

In [9]:
```python
from keras.optimizers import SGD


# visualize layer names and layer indices
for i, layer in enumerate(base_model.layers):
    print(i, layer.name)

# freeze first 280 layers and unfreeze the rest:
for layer in model.layers[:280]:
    layer.trainable = False
for layer in model.layers[280:]:
    layer.trainable = True


model.compile(optimizer=SGD(lr=1e-4, momentum=0.9), loss='categorical_crossentropy', metrics=['accuracy'])
```

```
0 input_1
1 conv2d_1
2 batch_normalization_1
3 activation_1
4 conv2d_2
5 batch_normalization_2
6 activation_2
7 conv2d_3
8 batch_normalization_3
9 activation_3
10 max_pooling2d_1
11 conv2d_4
12 batch_normalization_4
13 activation_4
14 conv2d_5
15 batch_normalization_5
16 activation_5
17 max_pooling2d_2
18 conv2d_9
```

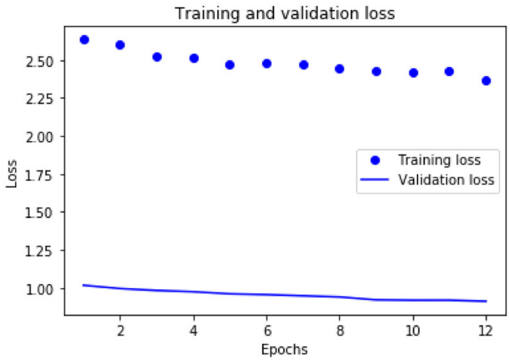
Step 4.2: Train the Model

In [10]:
```python
history = model.fit_generator(train_generator,
                    steps_per_epoch=NB_TRAIN_SAMPLES // BATCH_SIZE,
                    epochs=12,
                    validation_data=validation_generator,
                    validation_steps=NB_VALID_SAMPLES // BATCH_SIZE,
                    callbacks=[checkpointer],
                    verbose=1)
```

```
Epoch 1/12
207/208 [============================>.] - ETA: 0s - loss: 2.6358 - acc: 0.3438Epoch 00001: val_loss improved from 1.05781 to 1.01760, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 65s 314ms/step - loss: 2.6345 - acc: 0.3437 - val_loss: 1.0176 - val_acc: 0.6923
Epoch 2/12
207/208 [============================>.] - ETA: 0s - loss: 2.6028 - acc: 0.3469Epoch 00002: val_loss improved from 1.01760 to 0.99595, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 304ms/step - loss: 2.6024 - acc: 0.3469 - val_loss: 0.9959 - val_acc: 0.7055
Epoch 3/12
207/208 [============================>.] - ETA: 0s - loss: 2.5207 - acc: 0.3639Epoch 00003: val_loss improved from 0.99595 to 0.98330, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 304ms/step - loss: 2.5202 - acc: 0.3639 - val_loss: 0.9833 - val_acc: 0.7091
Epoch 4/12
207/208 [============================>.] - ETA: 0s - loss: 2.5135 - acc: 0.3619Epoch 00004: val_loss improved from 0.98330 to 0.97523, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 304ms/step - loss: 2.5146 - acc: 0.3618 - val_loss: 0.9752 - val_acc: 0.7043
Epoch 5/12
207/208 [============================>.] - ETA: 0s - loss: 2.4705 - acc: 0.3760Epoch 00005: val_loss improved from 0.97523 to 0.96115, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 303ms/step - loss: 2.4702 - acc: 0.3756 - val_loss: 0.9612 - val_acc: 0.7151
Epoch 6/12
207/208 [============================>.] - ETA: 0s - loss: 2.4818 - acc: 0.3741Epoch 00006: val_loss improved from 0.96115 to 0.95594, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 304ms/step - loss: 2.4825 - acc: 0.3738 - val_loss: 0.9559 - val_acc: 0.7139
Epoch 7/12
207/208 [============================>.] - ETA: 0s - loss: 2.4657 - acc: 0.3759Epoch 00007: val_loss improved from 0.95594 to 0.94835, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 304ms/step - loss: 2.4674 - acc: 0.3750 - val_loss: 0.9483 - val_acc: 0.7151
Epoch 8/12
207/208 [============================>.] - ETA: 0s - loss: 2.4467 - acc: 0.3753Epoch 00008: val_loss improved from 0.94835 to 0.94083, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 304ms/step - loss: 2.4437 - acc: 0.3760 - val_loss: 0.9408 - val_acc: 0.7163
Epoch 9/12
207/208 [============================>.] - ETA: 0s - loss: 2.4299 - acc: 0.3852Epoch 00009: val_loss improved from 0.94083 to 0.92158, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 303ms/step - loss: 2.4287 - acc: 0.3859 - val_loss: 0.9216 - val_acc: 0.7212
Epoch 10/12
207/208 [============================>.] - ETA: 0s - loss: 2.4194 - acc: 0.3814Epoch 00010: val_loss improved from 0.92158 to 0.91939, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 304ms/step - loss: 2.4225 - acc: 0.3812 - val_loss: 0.9194 - val_acc: 0.7224
Epoch 11/12
207/208 [============================>.] - ETA: 0s - loss: 2.4259 - acc: 0.3861Epoch 00011: val_loss did not improve
208/208 [==============================] - 63s 301ms/step - loss: 2.4258 - acc: 0.3861 - val_loss: 0.9195 - val_acc: 0.7212
Epoch 12/12
207/208 [============================>.] - ETA: 0s - loss: 2.3701 - acc: 0.3908Epoch 00012: val_loss improved from 0.91939 to 0.91232, saving model to saved_models/weights.best.transfer_learning_model.hdf5
208/208 [==============================] - 63s 304ms/step - loss: 2.3704 - acc: 0.3906 - val_loss: 0.9123 - val_acc: 0.7320
```

Step 4.3: Plot Training Progress

In [11]:
```
plot_history(history)
```



Step 4.4: Test the fine-tuned Model

In [12]:
```
model.load_weights('saved_models/weights.best.transfer_learning_model.hdf5')
```

In [13]:
```
test_accuracy = evaluate_model(model, test_tensors, test_targets)

print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 71.1722%
```

Conclusion

In this notebook we used transfer learning to create a classifier that predicts the dog breed for a given image. We saw that the results are quite impressive. By adding our own classifaction block on top of the InceptionV3 network (with ImageNet weights) we achieved an accuracy of ~68% on the test set; with just 12 epochs and 6680 images in the training set. One key step to make the most out of our data was to use data augmentation.

After we trained a first version of our own classifiaction block we could start to retrain parts of the InceptionV3 network to fine-tune our model and improve our results even more. Just unfreezing the last block (last 31 out of 310 layers) and training again for 12 epochs resulted in an improvement of the test accuracy of ~3%. With that we achieved >70% accuracy with only 6680 images in the training set with 133 classes. This is far better than any 'from scratch' approach could be.

Further Improvements

After unfreezing the last InceptionV3 block we could iteratively unfreeze more parts of the network to try to improve our results even more. Another option may be to experiment with different [base networks (https://keras.io/api/applications/)](https://keras.io/api/applications/) and different architectures for our classification block (although it seems to work quite well already).

Acknowledgement

This notebook was created as part of the Udacity Data Scientist Nanodegree program. Furthermore, it is highly inspired by this excellent [blog post (https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html)](https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html) authored by the author of the keras library himself, Francois Chollet.