Training a Dog Breed Classifier from Scratch (using CNNs)

The Road Ahead

- Motivation
- Step 1: Import Dog Datasets
- Step 2: Pre-process the Data
- Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
- Step 4: Test the Model
- Conclusion

Motivation

In this notebook we want to train a Convolutional Neural Network from scratch to classify dog breeds.

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**

Moreover, we will have very little training data. Overall, 8351 images for 133 categories.

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

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.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
```

```
100%|██████████| 6680/6680 [01:12<00:00, 92.44it/s]
100%|██████████| 835/835 [00:08<00:00, 103.53it/s]
100%|██████████| 836/836 [00:07<00:00, 104.62it/s]
```

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Step 3.1: Create Model Architecture

There are a lot of possibilities to experiment with network architectures. The one below is a pretty basic and simple architecture as it uses sequences of `Conv` and `Pooling` Layers, which is typical for CNNs. With more layers one could probably achieve a better result than this network but also the learning time and the potential to overfit would increase.

In [3]:
```python
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, Dense
from keras.optimizers import RMSprop
from keras.models import Sequential


model = Sequential()

model.add(Conv2D(32, (3, 3), input_shape=(224, 224, 3)))
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(MaxPooling2D((2, 2)))

model.add(GlobalAveragePooling2D())
model.add(Dense(133, activation='softmax'))

model.summary()

model.compile(optimizer=RMSprop(lr=0.005), loss='categorical_crossentropy', metrics=['accuracy'])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 222, 222, 32)      896       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 32)      0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 109, 109, 64)      18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 54, 54, 64)        0         
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0         
_________________________________________________________________
dense_1 (Dense)              (None, 133)               8645      
=================================================================
Total params: 28,037
Trainable params: 28,037
Non-trainable params: 0
_________________________________________________________________
```

Step 3.2: Train the Model

We train the model for 12 epochs with 6680 images to train and 835 images to validate with. With more epochs and data augmentation we could possibly improve our results here. As with model architectures, there are a lot of ways to experiment with model training. Nevertheles, the ~2% validation accuracy we will get here are already far better than the random baseline.

In [4]:
```
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.from_scratch_model.hdf5',
                               verbose=1,
                               save_best_only=True)

history = model.fit(train_tensors, train_targets,
                    validation_data=(valid_tensors, valid_targets),
                    epochs=12,
                    batch_size=32,
                    callbacks=[checkpointer],
                    verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.9025 - acc: 0.0074Epoch 00001: val_loss improved from inf to 4.87801, saving model to saved_models/weights.best.from_scratch_model.hdf5
6680/6680 [==============================] - 28s 4ms/step - loss: 4.9025 - acc: 0.0075 - val_loss: 4.8780 - val_acc: 0.0156
Epoch 2/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8706 - acc: 0.0128Epoch 00002: val_loss did not improve
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8706 - acc: 0.0127 - val_loss: 4.9095 - val_acc: 0.0144
Epoch 3/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8644 - acc: 0.0120Epoch 00003: val_loss improved from 4.87801 to 4.84982, saving model to saved_models/weights.best.from_scratch_model.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8639 - acc: 0.0120 - val_loss: 4.8498 - val_acc: 0.0132
Epoch 4/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8560 - acc: 0.0161Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8561 - acc: 0.0160 - val_loss: 4.8725 - val_acc: 0.0132
Epoch 5/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8506 - acc: 0.0155Epoch 00005: val_loss did not improve
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8503 - acc: 0.0154 - val_loss: 4.8890 - val_acc: 0.0108
Epoch 6/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8472 - acc: 0.0135Epoch 00006: val_loss improved from 4.84982 to 4.82841, saving model to saved_models/weights.best.from_scratch_model.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8473 - acc: 0.0135 - val_loss: 4.8284 - val_acc: 0.0144
Epoch 7/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8389 - acc: 0.0164Epoch 00007: val_loss improved from 4.82841 to 4.82000, saving model to saved_models/weights.best.from_scratch_model.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8396 - acc: 0.0163 - val_loss: 4.8200 - val_acc: 0.0180
Epoch 8/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8334 - acc: 0.0170Epoch 00008: val_loss did not improve
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8333 - acc: 0.0169 - val_loss: 4.8295 - val_acc: 0.0251
Epoch 9/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8247 - acc: 0.0218Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8246 - acc: 0.0219 - val_loss: 4.8365 - val_acc: 0.0192
Epoch 10/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8178 - acc: 0.0195Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8176 - acc: 0.0195 - val_loss: 4.8303 - val_acc: 0.0251
Epoch 11/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8127 - acc: 0.0194Epoch 00011: val_loss improved from 4.82000 to 4.79109, saving model to saved_models/weights.best.from_scratch_model.hdf5
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8126 - acc: 0.0193 - val_loss: 4.7911 - val_acc: 0.0240
Epoch 12/12
6656/6680 [============================>.] - ETA: 0s - loss: 4.8102 - acc: 0.0198Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 27s 4ms/step - loss: 4.8100 - acc: 0.0198 - val_loss: 4.8074 - val_acc: 0.0228
```

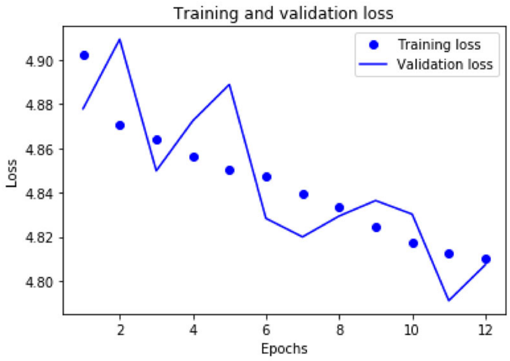
Step 3.3: Plot Training Progress

```
In [8]: from matplotlib import pyplot as plt


        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()
```



Step 4: Test the Model

```
In [6]: # Load the Model with the Best Validation Loss
        model.load_weights('saved_models/weights.best.from_scratch_model.hdf5')
```

```
In [7]: # 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
        test_accuracy = 100*np.sum(np.array(dog_breed_predictions)==np.argmax(test_targets, axis=1))/len(dog_breed_predictions)
        print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 2.7512%
```

Conclusion

In this notebook we trained a classifier to classify pictures of dogs into their breed from scratch using CNNs. For that task we only had very little data (Overall 8351 samlples for 133 classes). Nevertheless, we tried our best, used a typical CNN architecture and after just 12 epochs achieved a test accuracy of 2.75% (this may vary for different runs; but for me it was always >2%). That does not sound much but is still far better than a random guess, which would give us an acccuracy of ~0.75%.

The main options to improve our model would be to experiment with:

- a larger network; i.e. more layers
- more epochs
- data augmentation
- adding Dropouts
- different learning rates

But it may also be mentioned that to experiment with these options may need a lot of computing power and time.

In the end, with this little data we will get far better results using transfer learning instead of learning from scratch. An outstanding blog post for this topic is: [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)](https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html) by the keras author himself, Francois Chollet.