

Загрузка данных из Kaggle

Для выполнения требуется Kaggle API Token. Инструкция (пункт 7):
<https://towardsdatascience.com/7-ways-to-load-external-data-into-google-colab-7ba73e7d5fc7>

Account->API Token -> загрузить его в основную папку Colab

```
!pwd & ls
```

```
/content  
kaggle.json  sample_data
```

Закроем доступ к ключу для всех остальных: вдруг ноутбук открыт где-то не на Колабе

```
!chmod 600 /content/kaggle.json
```

Укажем утилите kaggle на расположение ключа API с помощью переменной окружения

```
import os  
os.environ['KAGGLE_CONFIG_DIR'] = "/content"  # переменная окружения
```

Загрузим данные с Kaggle: используем утилиту kaggle, укажем на раздел "соревнования", задача "загрузить", соревнование про породы собак

```
!kaggle competitions download -c dog-breed-identification
```

```
Downloading dog-breed-identification.zip to /content  
97% 667M/691M [00:03<00:00, 222MB/s]  
100% 691M/691M [00:03<00:00, 228MB/s]
```

Проверим, что данные загружены

```
!ls
```

```
dog-breed-identification.zip  kaggle.json  sample_data
```

Разархивируем данные. Строк много, поэтому выведем только последние 10 командой tail из Linux

```
!unzip /content/dog-breed-identification.zip | tail -n 10
```

```
inflating: train/ffc532991d3cd7880d27a449ed1c4770.jpg  
inflating: train/ffca1c97cea5fada05b8646998a5b788.jpg  
inflating: train/ffcb610e811817766085054616551f9c.jpg  
inflating: train/ffcde16e7da0872c357fbc7e2168c05f.jpg  
inflating: train/ffcffab7e4beef9a9b8076ef2ca51909.jpg  
inflating: train/ffd25009d635cfd16e793503ac5edef0.jpg  
inflating: train/ffd3f636f7f379c51ba3648a9ff8254f.jpg
```

```
inflating: train/ffe2ca6c940cddfee68fa3cc6c63213f.jpg
inflating: train/ffe5f6d8e2bfff356e9482a80a6e29aac.jpg
inflating: train/fff43b07992508bc822f33d8ffd902ae.jpg
```

Посмотрим, что разархивировалось: должны были появиться новые файлы

```
!ls
```

```
dog-breed-identification.zip  labels.csv  sample_submission.csv
train
kaggle.json                  sample_data  test
```

```
!ls train | tail -n 5
```

```
ffd25009d635cfd16e793503ac5edef0.jpg
ffd3f636f7f379c51ba3648a9ff8254f.jpg
ffe2ca6c940cddfee68fa3cc6c63213f.jpg
ffe5f6d8e2bfff356e9482a80a6e29aac.jpg
fff43b07992508bc822f33d8ffd902ae.jpg
```

Предварительный анализ данных

Прочитаем разметку из загруженного файла

```
import pandas as pd
```

```
labels_df = pd.read_csv('labels.csv') # чтение данных из CSV в Pandas
DataFrame
```

Посмотрим, какие данные есть

```
labels_df.head(5)
```

	id	breed
0	000bec180eb18c7604dcecc8fe0dba07	boston_bull
1	001513dfcb2ffa9c82cccf4d8bbaba97	dingo
2	001cdf01b096e06d78e9e5112d419397	pekinese
3	00214f311d5d2247d5dfe4fe24b2303d	bluetick
4	0021f9ceb3235effd7fcde7f7538ed62	golden_retriever

```
labels_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10222 entries, 0 to 10221
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  -
0    id      10222 non-null    object
1    breed   10222 non-null    object
```

```
dtypes: object(2)
memory usage: 159.8+ KB
```

Видно, что столбца два: идентификатор (видимо, название картинки), и порода. Дополним название картинки расширением, чтобы можно было читать их из файла

```
labels_df['id'] = labels_df['id'] + '.jpg' # добавляем к названию
разрешение файла, чтобы можно было к нему обращаться
```

```
labels_df.head(5) # 5 первых строк из датасета
```

	id	breed
0	000bec180eb18c7604dcecc8fe0dba07.jpg	boston_bull
1	001513dfcb2ffa8c82cccf4d8bbaba97.jpg	dingo
2	001cdf01b096e06d78e9e5112d419397.jpg	pekinese
3	00214f311d5d2247d5dfe4fe24b2303d.jpg	bluetick
4	0021f9ceb3235effd7fcde7f7538ed62.jpg	golden_retriever

По выводу .info() выше видно, что пропущенных данных нет - все "non-null", и длина столбцов совпадает. Судя по описанию из Kaggle, в train - только картинки. Значит, смотреть числовые характеристики признаков (среднее, среднеквадратическое отклонение и прочие) смысла нет - данные строковые.

Тогда можно обратить внимание на столбец "порода" ("breed"). Посмотрим распределение пород

```
labels_df['breed'].value_counts() # количество вхождений изображений
собак каждой породы
```

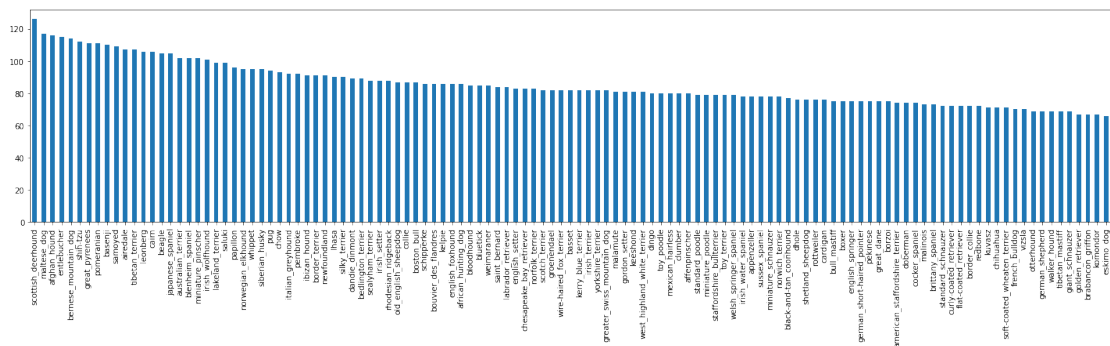
scottish_deerhound	126
maltese_dog	117
afghan_hound	116
entlebucher	115
bernese_mountain_dog	114
...	
golden_retriever	67
brabancon_griffon	67
komondor	67
eskimo_dog	66
briard	66

Name: breed, Length: 120, dtype: int64

Видно, что, хоть некоторых пород в два раза больше, чем других, тем не менее, нет таких пород, примеров которых совсем мало. Значит, можно считать, что датасет более-менее сбалансированный. Для наглядности построим столбчатую диаграмму численности каждой породы:

```
labels_df['breed'].value_counts().plot.bar(figsize=(25, 5)) #
распределение пород на графике
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f14518789d0>
```



Очевидно, что название картинки (похожее на какой-то хеш) не несёт в себе дополнительной информации, и не может быть использовано как признак - это всего лишь индекс. Поэтому посмотрим на сами картинки.

```
labels_df['id'][0] # название картинки
```

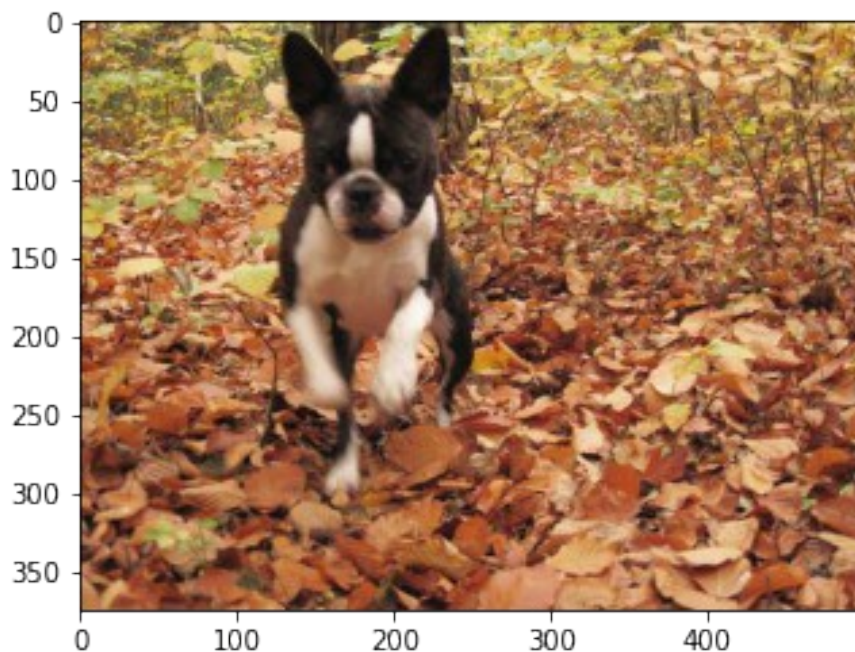
```
{"type": "string"}
```

```
import matplotlib.pyplot as plt
```

```
image = plt.imread('train/' + labels_df['id'][0]) # добавим путь  
через папку "train" и прочитаем картинку
```

```
plt.imshow(image) # нарисуем картинку
```

```
<matplotlib.image.AxesImage at 0x7f144f87df90>
```

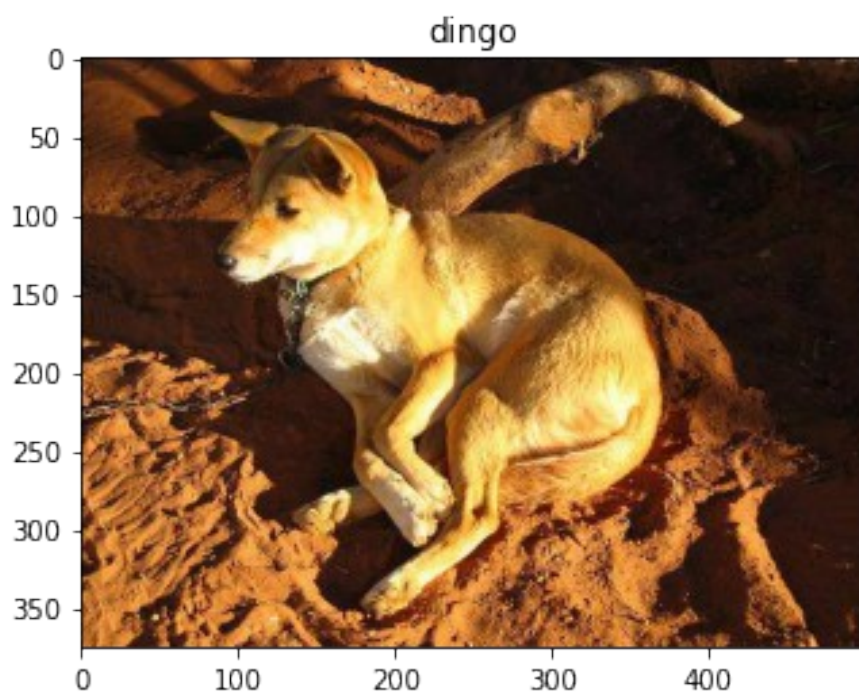
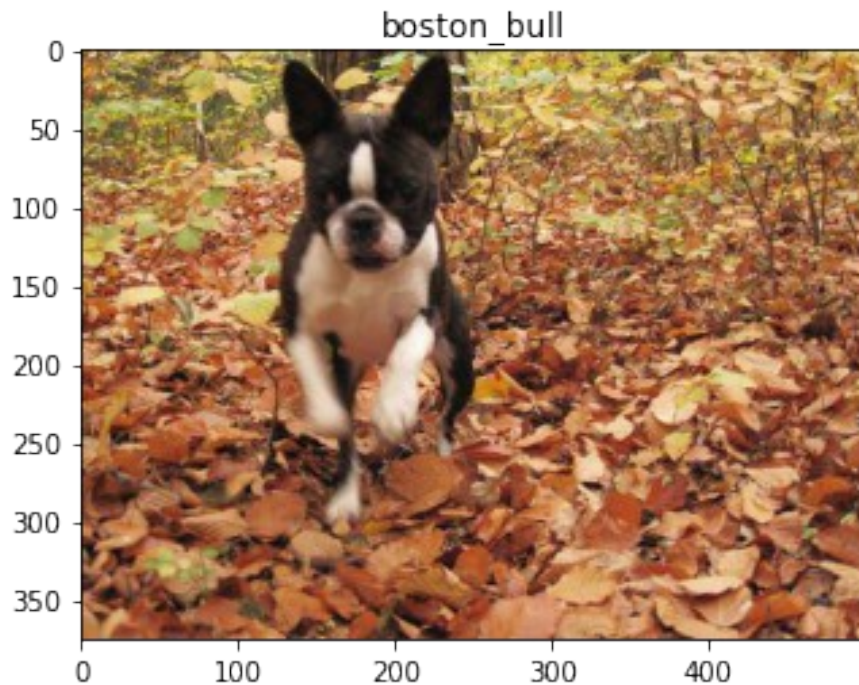


Нарисуем сразу несколько картинок с подписями

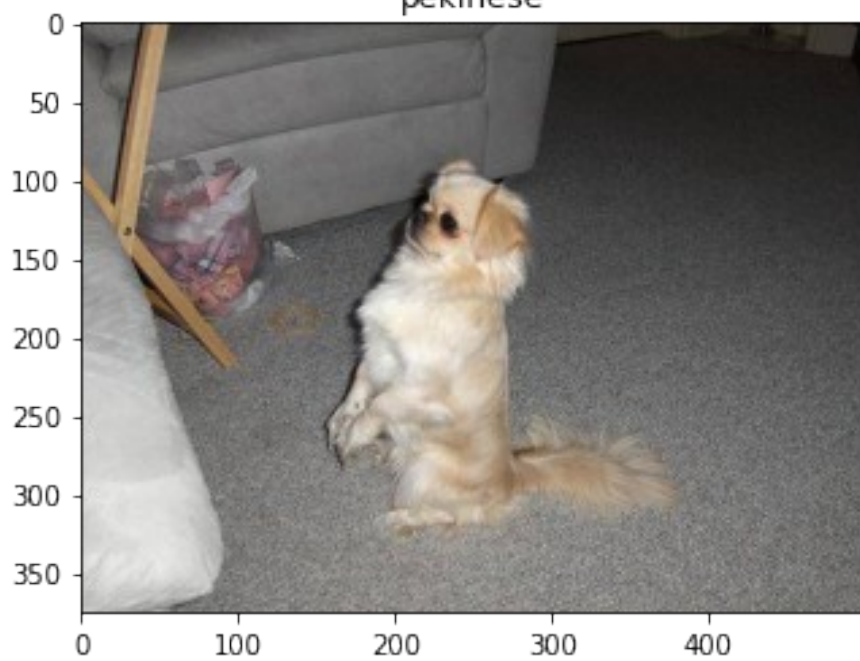
```

for img_path, breed in zip(labels_df['id'][:5], labels_df['breed']
[:5]): # для первых пяти картинок: для каждой пары "id" и "порода"
    plt.figure() # создаём новый объект для рисования картинки
    image = plt.imread('train/' + img_path) # добавим путь через
папку "train" и прочитаем картинку
    plt.imshow(image) # нарисуем картинку
    plt.title(breed) # подписываем породу

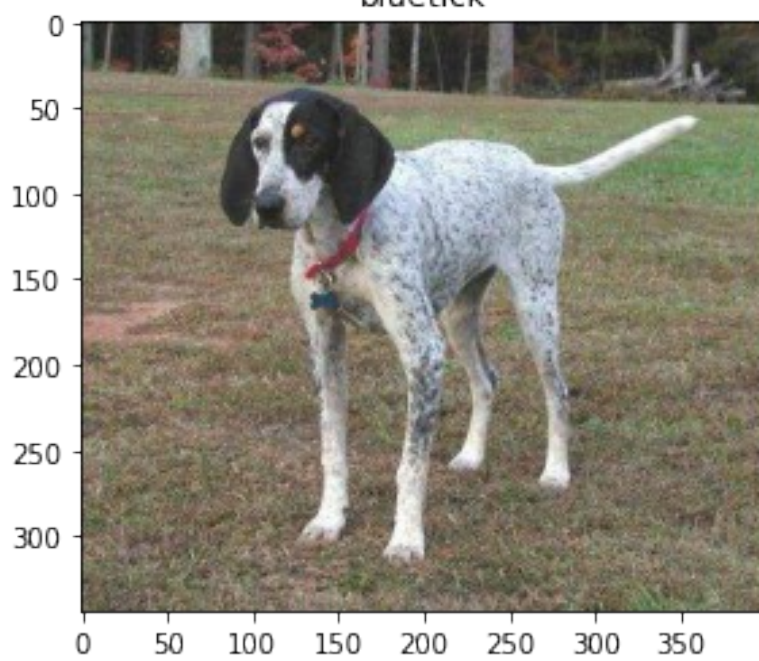
```

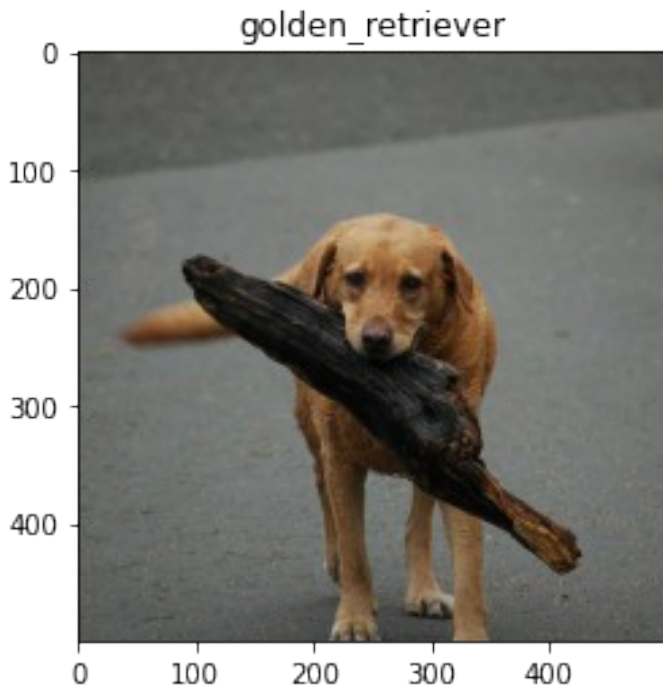


pekinese



bluetick





Базовая модель без машинного обучения

Какого-то признака, по которым можно было бы угадать породу без машинного обучения, не видно. А ещё видно, что все картинки разных размеров - для использования моделей МО надо привести их к одному.

Подготовка данных для моделей машинного обучения

Сначала надо получить данные в понятном формате. На курсе про нейронные сети от Андрея Созыкина рассказывали, что для этого нужно получить из модели отдельные пиксели, разделить их на 3 цвета (RGB) и разделить на 255, чтобы входные данные были массивом чисел от 0 до 1

Гугл советует для этого использовать функции из библиотеки Keras

https://www.tensorflow.org/api_docs/python/tf/keras/utils/load_img

```
import keras
import tensorflow as tf
```

Попробуем:

```
image = keras.preprocessing.image.load_img('train/' + labels_df['id']
[0]) # читаем картинку из файла

image_m = keras.preprocessing.image.img_to_array(image) # преобразуем
в числа
```

```
image_matrix = tf.image.rgb_to_grayscale(image_m) # преобразуем в grayscale (по трем цветам оттенков серого)
```

```
type(image_matrix) # тип преобразованной картинки
```

```
tensorflow.python.framework.ops.EagerTensor
```

```
image_matrix.shape # размерность
```

```
TensorShape([375, 500, 1])
```

Сейчас надо привести все картинки к одному размеру. Гугл отправляет к https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator, который умеет много всего интересного

Создадим генератор данных, который из картинок делает массивы одной размерности

```
data_gen = keras.preprocessing.image.ImageDataGenerator(
    rescale = 1.0 / 255, # яркость пикселя привести в 0..1
    validation_split = 0.2, # в качестве валидационной выборки взять 20%
    shear_range=0.2, # Интенсивность сдвига
    zoom_range=0.2, # Диапазон случайного увеличения
    horizontal_flip=True) # Произвольный поворот по горизонтали
```

Создадим тренировочную и тестовую (валидационную) выборки

```
train_gen = data_gen.flow_from_dataframe(labels_df, # откуда брать
    directory='train', # откуда брать
    x_col='id', # колонка, отвечающая
    y_col='breed', # колонка - ответы
    subset='training', # это -
    target_size=(200, 200), # размер
    batch_size=64, # по 64 картинки в
    seed=1) # для повторяемости
```

```
val_gen = data_gen.flow_from_dataframe(labels_df,
    directory='train', # тестовые
    x_col='id',
    y_col='breed',
    subset='validation', # а это -
    target_size=(200, 200),
```



```
batch_size=64,  
seed=1)
```

Found 8178 validated image filenames belonging to 120 classes.
Found 2044 validated image filenames belonging to 120 classes.

Снова попробуем нарисовать - посмотрим, что получилось. Посмотрим на один батч из валидационной выборки (val)

```
images = val_gen.next() # следующий батч валидационной выборки (шаг обучения)
```

```
images[0][0] # Первый индекс - выбор данных о пикселях [0] или One-Hot разметка лэйблов [1]. Второй индекс - номер картинки
```

```
array([[0.0672951 , 0.07513823, 0.07121667],  
       [0.07385404, 0.08169717, 0.07777561],  
       [0.06663878, 0.07448193, 0.07056036],  
       ...,  
       [0.20008701, 0.11089982, 0.03728492],  
       [0.19800618, 0.10882369, 0.03519467],  
       [0.19532941, 0.10615163, 0.03250849]],  
  
       [[0.06371383, 0.07155696, 0.06763539],  
       [0.06503396, 0.07287709, 0.06895553],  
       [0.06360118, 0.07144432, 0.06752275],  
       ...,  
       [0.18128774, 0.09109166, 0.02936636],  
       [0.1852229 , 0.09502681, 0.0332827 ],  
       [0.18323214, 0.09303606, 0.03127312]],  
  
       [[0.06185072, 0.06969385, 0.06577228],  
       [0.05571071, 0.06355385, 0.05963228],  
       [0.06243069, 0.07027382, 0.06635226],  
       ...,  
       [0.18369961, 0.09198474, 0.04315795],  
       [0.18444058, 0.0927304 , 0.04388479],  
       [0.18259338, 0.09088791, 0.04202348]],  
  
       ...,  
  
       [[0.47029933, 0.37226012, 0.12127969],  
       [0.4661695 , 0.36813027, 0.11714985],  
       [0.5006176 , 0.40257844, 0.15159802],  
       ...,  
       [0.28567418, 0.14879991, 0.02856793],  
       [0.26053458, 0.13183889, 0.02819003],  
       [0.25768968, 0.14542677, 0.07203298]],  
  
       [[0.4432461 , 0.3452069 , 0.09422648],  
       [0.4480843 , 0.35004508, 0.09906468],
```

```

        [0.46707812, 0.36903888, 0.11805848],
        ...,
        [0.2839918 , 0.14711753, 0.02688554],
        [0.27174765, 0.14477693, 0.03595309],
        [0.26547238, 0.15458679, 0.07397515]],

[[[0.4840128 , 0.3859736 , 0.13018763],
  [0.48865724, 0.39061803, 0.13484146],
  [0.48758984, 0.38955063, 0.13378349],
  ...,
  [0.29332945, 0.15645516, 0.03622318],
  [0.28585   , 0.1602745 , 0.04726506],
  [0.26745993, 0.15707865, 0.07035462]]], dtype=float32)

images[0][0].shape # размерность данных - по 200 пикселей на сторону
и 3 канала - RGB

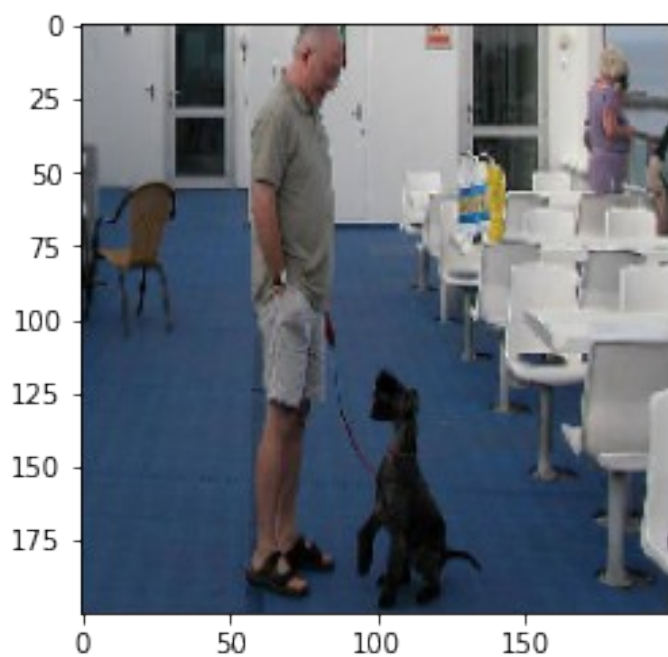
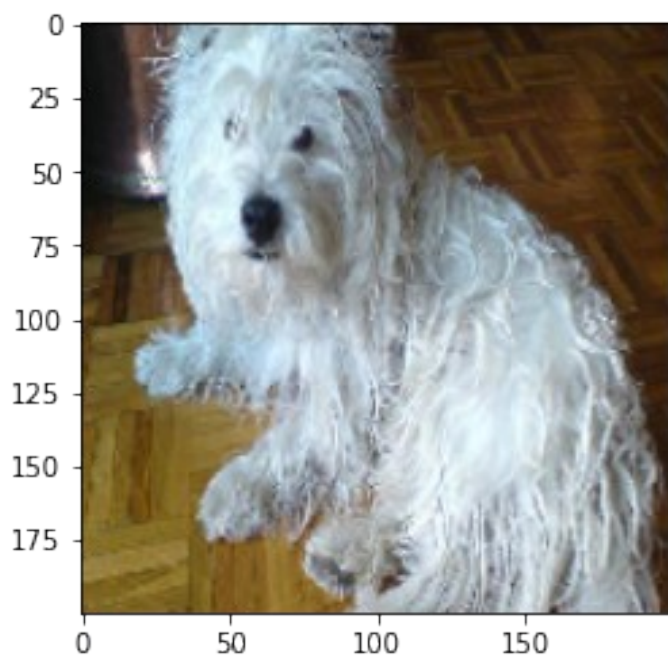
(200, 200, 3)

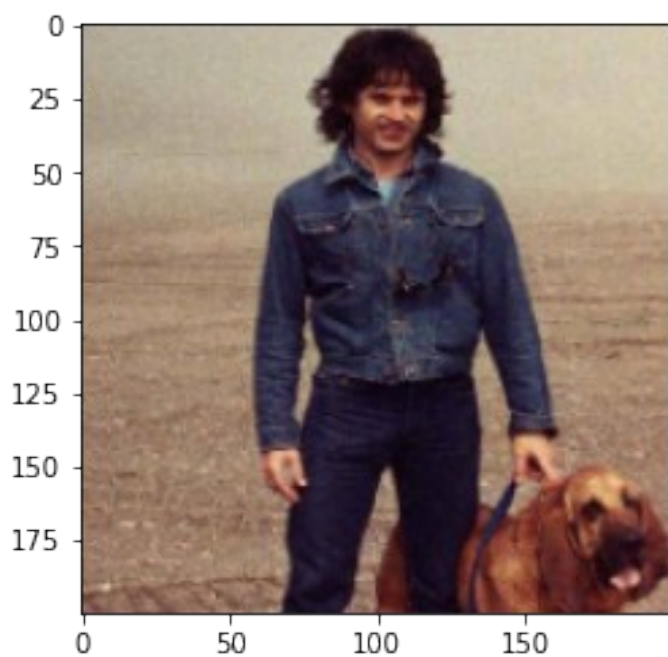
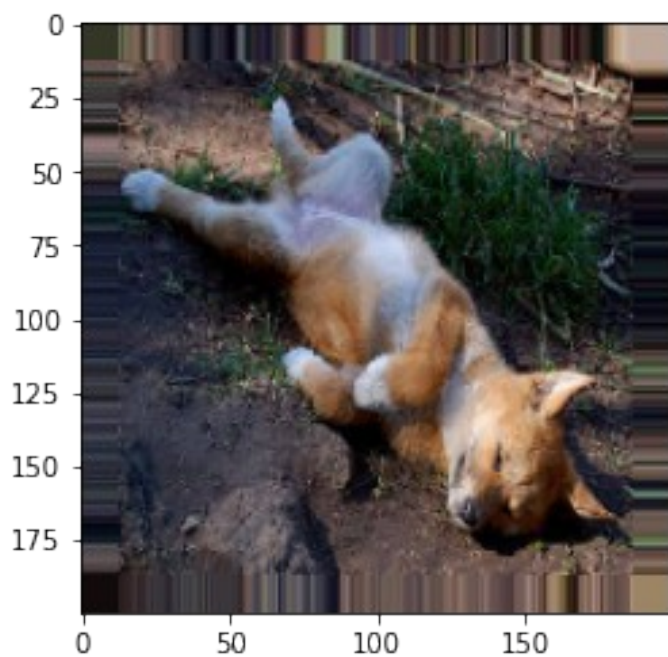
images[1][0] # разметка в One-Hot (Keras сразу сделал для удобства
обучения нейронки): ответы, известные из labels_df

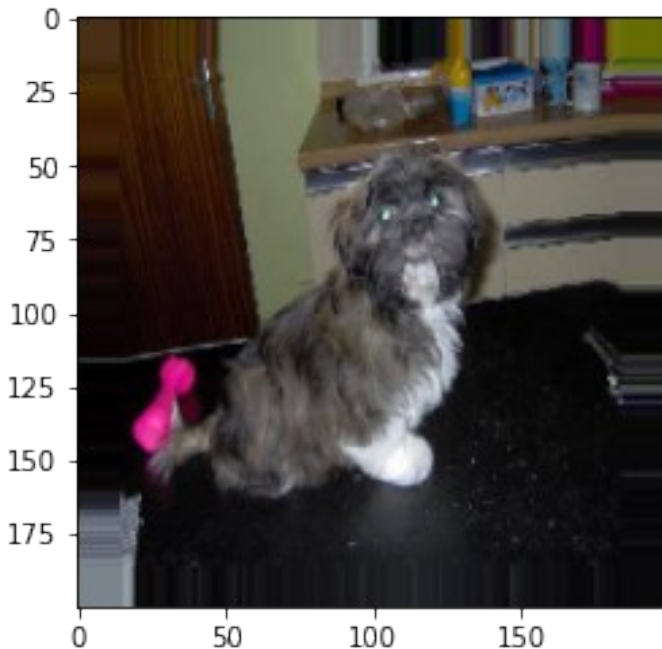
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
      0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
      0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
      0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
      0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
      0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
      0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.,
0.,
      0.], dtype=float32)

for image in images[0][:5]: # для пикселей первых 5 картинок батча
    plt.figure() # создаём новый объект для рисования картинки
    plt.imshow(image) # рисуем картинку

```







Данные готовы, теперь можно начинать обучение

Несколько моделей машинного обучения - от простых до сложных

Надо проверить, что GPU включено - иначе будет очень долго обучаться

```
import tensorflow as tf
print("Num GPUs Available: ",
      len(tf.config.experimental.list_physical_devices('GPU')))
```

Num GPUs Available: 1

Попробуем собственную простую нейронку - просто запоминать значения пикселей (пока без свёрток)

Описываем архитектуру сети

```
model = keras.models.Sequential() # создаём сеть с последовательными
# слоями
model.add(keras.layers.Flatten(input_shape=(200, 200, 3))) #
# превращаем трёхмерную картинку в одномерный массив
model.add(keras.layers.Dense( # полносвязный слой
    3000)) # количество нейронов на выходе слоя
model.add(keras.layers.Activation('relu')) # функция активации -
# вносим нелинейность
model.add(keras.layers.Dense(120)) # выходной слой, количество
# нейронов равно количеству пород (у нас One-Hot разметка)
```



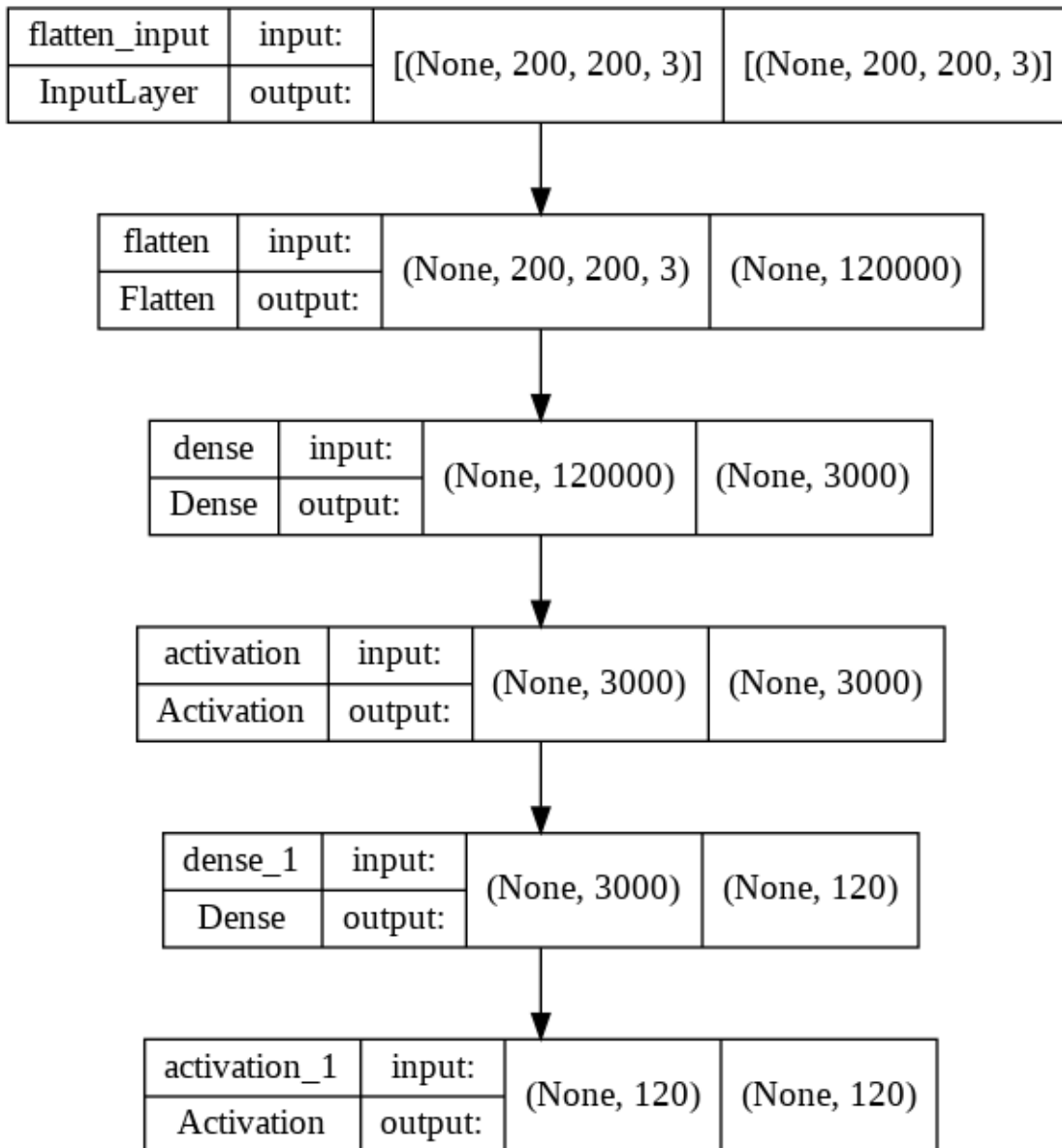
```
model.add(keras.layers.Activation('softmax')) # сигмоида для выбора  
ответа
```

Инициализируем сеть - запишем в память веса и прочее

```
model.compile(loss='categorical_crossentropy', # у нас несколько  
классов - поэтому категориальная  
optimizer='adam', # возьмём самый навороченный - чтобы  
и градиент по-умному считался, и шаг менялся интересно  
metrics=['accuracy']) # нужна какая-то метрика, чтобы  
смотреть, как модель обучается
```

Нарисуем архитектуру сети - из Keras почему-то не вызывается, на
StackOverflow написано, что надо импортировать TensorFlow (хотя Keras - это
надстройка над TensorFlow -_-)

```
import tensorflow as tf  
tf.keras.utils.plot_model(model, 'model.png', show_shapes=True)
```



Обучим нашу сеть

```
model.fit(train_gen, # генератор тренировочных данных
          validation_data=val_gen, # генератор валидационных данных
          epochs=10, # количество эпох для обучения
          steps_per_epoch=len(train_gen),
          validation_steps=len(val_gen))
```

Epoch 1/10

128/128 [=====] - 41s 316ms/step - loss: 106.0337 - accuracy: 0.0083 - val_loss: 4.7878 - val_accuracy: 0.0088

Epoch 2/10

128/128 [=====] - 41s 321ms/step - loss: 4.7884 - accuracy: 0.0112 - val_loss: 4.7885 - val_accuracy: 0.0088

Epoch 3/10

```

128/128 [=====] - 40s 313ms/step - loss:
4.7850 - accuracy: 0.0121 - val_loss: 4.7870 - val_accuracy: 0.0088
Epoch 4/10
128/128 [=====] - 40s 312ms/step - loss:
4.7836 - accuracy: 0.0125 - val_loss: 4.7860 - val_accuracy: 0.0088
Epoch 5/10
128/128 [=====] - 40s 315ms/step - loss:
4.7831 - accuracy: 0.0124 - val_loss: 4.7857 - val_accuracy: 0.0088
Epoch 6/10
128/128 [=====] - 41s 320ms/step - loss:
4.7808 - accuracy: 0.0124 - val_loss: 4.7858 - val_accuracy: 0.0088
Epoch 7/10
128/128 [=====] - 41s 319ms/step - loss:
4.7791 - accuracy: 0.0124 - val_loss: 4.7855 - val_accuracy: 0.0088
Epoch 8/10
128/128 [=====] - 42s 331ms/step - loss:
4.7787 - accuracy: 0.0124 - val_loss: 4.7850 - val_accuracy: 0.0088
Epoch 9/10
128/128 [=====] - 40s 316ms/step - loss:
4.7769 - accuracy: 0.0124 - val_loss: 4.7867 - val_accuracy: 0.0088
Epoch 10/10
128/128 [=====] - 43s 332ms/step - loss:
4.7776 - accuracy: 0.0122 - val_loss: 4.7850 - val_accuracy: 0.0088

```

<keras.callbacks.History at 0x7fe672580cd0>

Результат - ничего не обучилось. В целом, оно и логично - просто по цвету отдельного пикселя сложно что-то предсказать

Теперь попробуем сеть со свёртками - стандартное решение для обработки изображений

Архитектура модели

Статья на Хабре про свёртки: <https://habr.com/ru/post/454986/>

И ещё одна с гифками: <https://proglib.io/p/convolution>

```

model = keras.models.Sequential() # создаём сеть с последовательными
слоями

model.add(keras.layers.Conv2D( # операция свёртки - воздействуем
    32, # размерность выходного пространства - количество применяемых
    матриц (ядер) свёртки
    (3, 3), # kernel_size - высота и ширина окна двумерной свертки
    input_shape=(200, 200, 3), # размерность входных данных
    padding='same')) # отступы мы определим через padding = 'same', то
есть, мы не меняем размер изображения
model.add(keras.layers.Activation('relu')) # функция активации -
вносим нелинейность

```

```
model.add(keras.layers.MaxPooling2D( # пулинг - выбор максимального
    pool_size=(2, 2)) # размер окна
    значения в окне (как в ночной съёмке)
model.add(keras.layers.Dropout(0.2)) # каждую эпоху замораживаем
половину весов, чтобы не было переобучения
model.add(keras.layers.BatchNormalization()) # нормализует входные
данные, поступающие в следующий слой
```

```
model.add(keras.layers.Conv2D(32, (3, 3), padding='same'))
model.add(keras.layers.Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.BatchNormalization())
```

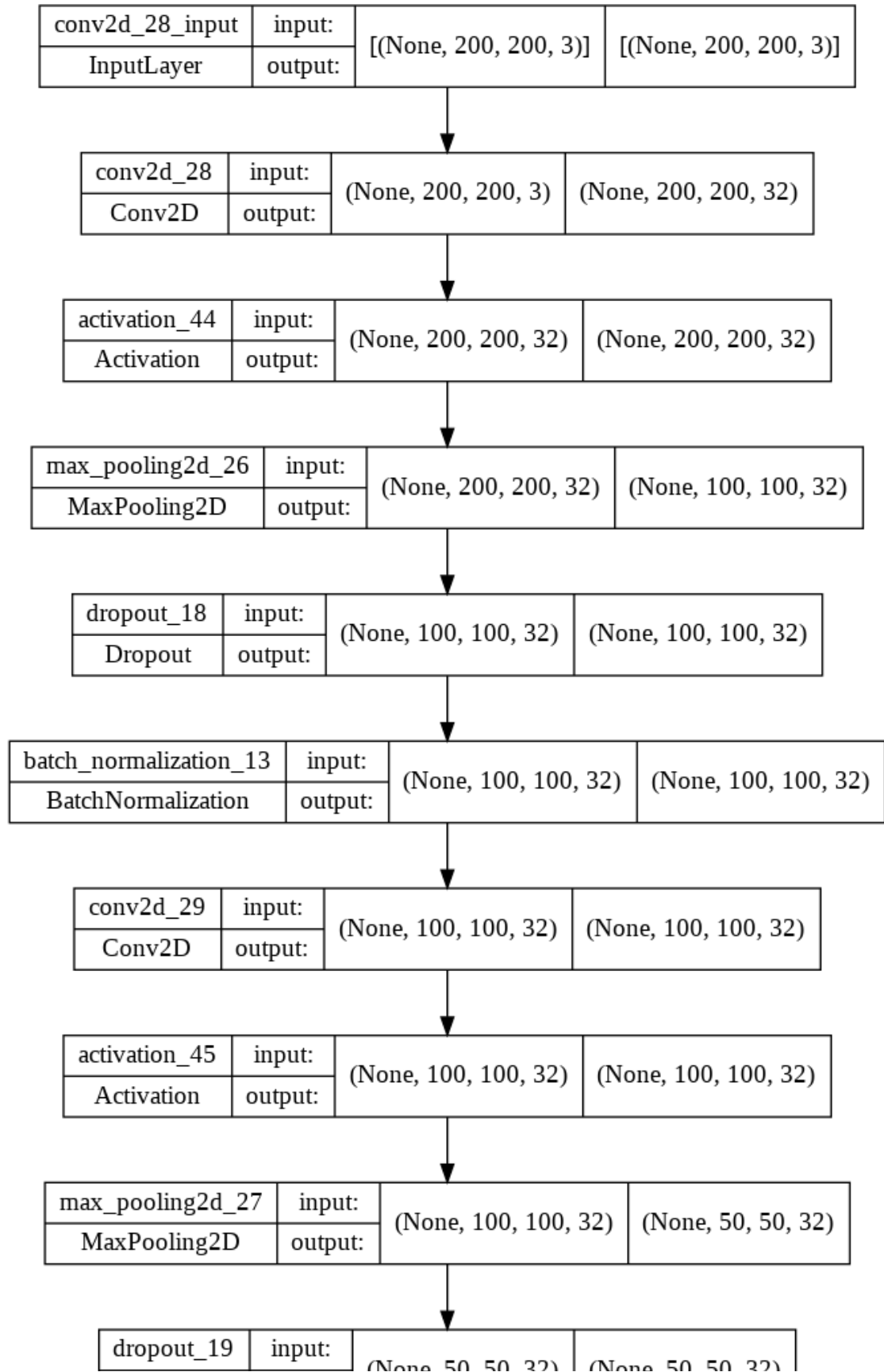
```
model.add(keras.layers.Conv2D(64, (3, 3), padding='same'))
model.add(keras.layers.Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.BatchNormalization())
```

```
model.add(keras.layers.Conv2D(64, (3, 3), padding='same'))
model.add(keras.layers.Activation('relu'))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Conv2D(128, (3, 3), padding='same'))
model.add(keras.layers.Activation('relu'))
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.BatchNormalization())
```

```
model.add(keras.layers.Flatten()) # превращаем трёхмерную картинку в
одномерный массив
model.add(keras.layers.Dropout(0.2))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(240)) # полносвязный слой
model.add(keras.layers.Activation('relu'))
model.add(keras.layers.Dropout(0.2)) # каждую эпоху замораживаем
половину весов, чтобы не было переобучения
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(120)) # выходной слой, количество
нейронов равно количеству пород (у нас One-Hot разметка)
model.add(keras.layers.Activation('softmax')) # сигмоида для выбора
ответа
```

```
model.compile(loss='categorical_crossentropy', # у нас несколько
классов - поэтому категориальная
    optimizer='adam', # возьмём самый навороченный - чтобы
и градиент по-умному считался, и шаг менялся интересно
    metrics=['accuracy']) # нужна какая-то метрика, чтобы
смотреть, как модель обучается
```

```
import tensorflow as tf
tf.keras.utils.plot_model(model, 'model.png', show_shapes=True)
```

```
model.fit(train_gen, # генератор тренировочных данных
          validation_data=val_gen, # генератор валидационных данных
          epochs=50, # количество эпох для обучения
          steps_per_epoch=len(train_gen),
          validation_steps=len(val_gen))
```

Epoch 1/50

128/128 [=====] - 119s 917ms/step - loss: 4.9428 - accuracy: 0.0230 - val_loss: 5.4473 - val_accuracy: 0.0103

Epoch 2/50

128/128 [=====] - 116s 911ms/step - loss: 4.4885 - accuracy: 0.0477 - val_loss: 5.9661 - val_accuracy: 0.0113

Epoch 3/50

128/128 [=====] - 115s 896ms/step - loss: 4.2561 - accuracy: 0.0723 - val_loss: 5.4820 - val_accuracy: 0.0205

Epoch 4/50

128/128 [=====] - 115s 903ms/step - loss: 4.0762 - accuracy: 0.0931 - val_loss: 4.8392 - val_accuracy: 0.0352

Epoch 5/50

128/128 [=====] - 114s 895ms/step - loss: 3.8914 - accuracy: 0.1170 - val_loss: 4.9677 - val_accuracy: 0.0362

Epoch 6/50

128/128 [=====] - 115s 895ms/step - loss: 3.7349 - accuracy: 0.1445 - val_loss: 4.4511 - val_accuracy: 0.0528

Epoch 7/50

128/128 [=====] - 116s 905ms/step - loss: 3.5967 - accuracy: 0.1658 - val_loss: 4.6488 - val_accuracy: 0.0533

Epoch 8/50

128/128 [=====] - 135s 1s/step - loss: 3.4481 - accuracy: 0.1857 - val_loss: 4.3297 - val_accuracy: 0.0763

Epoch 9/50

128/128 [=====] - 115s 899ms/step - loss: 3.3115 - accuracy: 0.2147 - val_loss: 4.2781 - val_accuracy: 0.0802

Epoch 10/50

128/128 [=====] - 115s 899ms/step - loss: 3.1644 - accuracy: 0.2405 - val_loss: 4.2403 - val_accuracy: 0.0812

Epoch 11/50

128/128 [=====] - 114s 891ms/step - loss: 3.0454 - accuracy: 0.2646 - val_loss: 4.2320 - val_accuracy: 0.0905

Epoch 12/50

128/128 [=====] - 113s 884ms/step - loss: 2.8727 - accuracy: 0.3001 - val_loss: 4.3232 - val_accuracy: 0.0900

Epoch 13/50

128/128 [=====] - 113s 883ms/step - loss: 2.7291 - accuracy: 0.3277 - val_loss: 4.0399 - val_accuracy: 0.1106

Epoch 14/50

128/128 [=====] - 113s 883ms/step - loss: 2.6013 - accuracy: 0.3591 - val_loss: 4.3043 - val_accuracy: 0.0978

Epoch 15/50

128/128 [=====] - 113s 880ms/step - loss:

2.4671 - accuracy: 0.3781 - val_loss: 4.6430 - val_accuracy: 0.0841
Epoch 16/50
128/128 [=====] - 114s 894ms/step - loss:
2.3259 - accuracy: 0.4188 - val_loss: 4.1987 - val_accuracy: 0.1150
Epoch 17/50
128/128 [=====] - 113s 882ms/step - loss:
2.1705 - accuracy: 0.4416 - val_loss: 4.3556 - val_accuracy: 0.0964
Epoch 18/50
128/128 [=====] - 113s 884ms/step - loss:
2.0497 - accuracy: 0.4740 - val_loss: 4.3555 - val_accuracy: 0.1194
Epoch 19/50
128/128 [=====] - 115s 896ms/step - loss:
1.9394 - accuracy: 0.4990 - val_loss: 4.3548 - val_accuracy: 0.1106
Epoch 20/50
128/128 [=====] - 113s 884ms/step - loss:
1.8528 - accuracy: 0.5204 - val_loss: 4.3920 - val_accuracy: 0.1189
Epoch 21/50
128/128 [=====] - 113s 883ms/step - loss:
1.7632 - accuracy: 0.5419 - val_loss: 4.8259 - val_accuracy: 0.0915
Epoch 22/50
128/128 [=====] - 114s 890ms/step - loss:
1.6415 - accuracy: 0.5696 - val_loss: 4.3686 - val_accuracy: 0.1262
Epoch 23/50
128/128 [=====] - 112s 878ms/step - loss:
1.5591 - accuracy: 0.5846 - val_loss: 4.9047 - val_accuracy: 0.0793
Epoch 24/50
128/128 [=====] - 113s 883ms/step - loss:
1.4668 - accuracy: 0.6094 - val_loss: 4.6121 - val_accuracy: 0.1032
Epoch 25/50
128/128 [=====] - 114s 892ms/step - loss:
1.3901 - accuracy: 0.6218 - val_loss: 4.5407 - val_accuracy: 0.1262
Epoch 26/50
128/128 [=====] - 114s 894ms/step - loss:
1.3532 - accuracy: 0.6303 - val_loss: 4.4894 - val_accuracy: 0.1150
Epoch 27/50
128/128 [=====] - 113s 883ms/step - loss:
1.2604 - accuracy: 0.6585 - val_loss: 4.8839 - val_accuracy: 0.1032
Epoch 28/50
128/128 [=====] - 113s 886ms/step - loss:
1.1944 - accuracy: 0.6731 - val_loss: 4.8560 - val_accuracy: 0.1174
Epoch 29/50
128/128 [=====] - 114s 895ms/step - loss:
1.1830 - accuracy: 0.6736 - val_loss: 4.7830 - val_accuracy: 0.1115
Epoch 30/50
128/128 [=====] - 113s 886ms/step - loss:
1.0991 - accuracy: 0.6981 - val_loss: 5.1434 - val_accuracy: 0.1125
Epoch 31/50
128/128 [=====] - 114s 890ms/step - loss:
1.0596 - accuracy: 0.7027 - val_loss: 4.7636 - val_accuracy: 0.1164
Epoch 32/50

128/128 [=====] - 113s 885ms/step - loss:
1.0076 - accuracy: 0.7194 - val_loss: 5.0906 - val_accuracy: 0.1067
Epoch 33/50
128/128 [=====] - 114s 891ms/step - loss:
1.0202 - accuracy: 0.7150 - val_loss: 5.0860 - val_accuracy: 0.1145
Epoch 34/50
128/128 [=====] - 113s 882ms/step - loss:
0.9852 - accuracy: 0.7168 - val_loss: 5.0081 - val_accuracy: 0.1204
Epoch 35/50
128/128 [=====] - 113s 881ms/step - loss:
0.9170 - accuracy: 0.7372 - val_loss: 4.9443 - val_accuracy: 0.1394
Epoch 36/50
128/128 [=====] - 113s 884ms/step - loss:
0.9185 - accuracy: 0.7389 - val_loss: 5.2501 - val_accuracy: 0.1071
Epoch 37/50
128/128 [=====] - 112s 874ms/step - loss:
0.8837 - accuracy: 0.7443 - val_loss: 5.3818 - val_accuracy: 0.0978
Epoch 38/50
128/128 [=====] - 113s 886ms/step - loss:
0.8387 - accuracy: 0.7567 - val_loss: 5.4417 - val_accuracy: 0.1115
Epoch 39/50
128/128 [=====] - 113s 882ms/step - loss:
0.8080 - accuracy: 0.7728 - val_loss: 5.3865 - val_accuracy: 0.1037
Epoch 40/50
128/128 [=====] - 113s 880ms/step - loss:
0.7809 - accuracy: 0.7740 - val_loss: 5.1610 - val_accuracy: 0.1262
Epoch 41/50
128/128 [=====] - 114s 889ms/step - loss:
0.7880 - accuracy: 0.7726 - val_loss: 5.3671 - val_accuracy: 0.1169
Epoch 42/50
128/128 [=====] - 113s 887ms/step - loss:
0.7561 - accuracy: 0.7875 - val_loss: 5.1651 - val_accuracy: 0.1277
Epoch 43/50
128/128 [=====] - 115s 903ms/step - loss:
0.7401 - accuracy: 0.7841 - val_loss: 5.1897 - val_accuracy: 0.1218
Epoch 44/50
128/128 [=====] - 114s 895ms/step - loss:
0.7291 - accuracy: 0.7821 - val_loss: 5.1637 - val_accuracy: 0.1233
Epoch 45/50
128/128 [=====] - 115s 897ms/step - loss:
0.7044 - accuracy: 0.7935 - val_loss: 5.4558 - val_accuracy: 0.1027
Epoch 46/50
128/128 [=====] - 115s 901ms/step - loss:
0.6773 - accuracy: 0.8039 - val_loss: 5.3111 - val_accuracy: 0.1272
Epoch 47/50
128/128 [=====] - 115s 896ms/step - loss:
0.6514 - accuracy: 0.8092 - val_loss: 5.2856 - val_accuracy: 0.1321
Epoch 48/50
128/128 [=====] - 115s 896ms/step - loss:
0.6650 - accuracy: 0.8059 - val_loss: 5.4133 - val_accuracy: 0.1204

```
Epoch 49/50
128/128 [=====] - 116s 904ms/step - loss:
0.6575 - accuracy: 0.8084 - val_loss: 5.3896 - val_accuracy: 0.1296
Epoch 50/50
128/128 [=====] - 115s 902ms/step - loss:
0.6257 - accuracy: 0.8136 - val_loss: 5.4072 - val_accuracy: 0.1252
```

```
<keras.callbacks.History at 0x7fe686560750>
```

```
model.save_weights('first_model_weights.h5') #Сохранение весов модели
model.save('path') #Сохранение модели
#model.load_weights('first_model_weights.h5') # Загрузка весов модели
#prediction = model.predict(x) #Использование модели для предсказания
```

```
INFO:tensorflow:Assets written to: path/assets
```

```
!zip -r /content/file.zip /content/path
```

```
adding: content/path/ (stored 0%)
adding: content/path/variables/ (stored 0%)
adding: content/path/variables/variables.data-00000-of-00001
(deflated 10%)
adding: content/path/variables/variables.index (deflated 74%)
adding: content/path/keras_metadata.pb (deflated 94%)
adding: content/path/saved_model.pb (deflated 90%)
adding: content/path/assets/ (stored 0%)
```

```
from google.colab import files
files.download("/content/file.zip")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Видно, что модель переобучается - точность на тренировочных данных гораздо больше, чем на тестовых. Хотя Dropout и так уже 0.5 - то есть половина весов замораживается... Можно попробовать или доработать датасет (делать перевероты-развороты картинок), или использовать большую предобученную модель

Inception ResNet

Попробуем использовать большую предобученную модель

```
import tensorflow_hub as hub
url =
'https://tfhub.dev/google/imagenet/inception\_resnet\_v2/classification/5'
inception_resnet = tf.keras.Sequential([
    hub.KerasLayer(url, trainable=False),
    tf.keras.layers.Dense(120, activation='softmax'),
```



```

])
inception_resnet.build((None, 224, 224, 3))
inception_resnet.compile(loss='categorical_crossentropy',
                        optimizer='adam', metrics=['accuracy'])
h = inception_resnet.fit(train_gen, validation_data=val_gen,
                        epochs=10,
                        steps_per_epoch=len(train_gen),
                        validation_steps=len(val_gen))

Epoch 1/10
128/128 [=====] - 67s 409ms/step - loss:
1.7433 - accuracy: 0.6835 - val_loss: 1.1488 - val_accuracy: 0.7495
Epoch 2/10
128/128 [=====] - 48s 376ms/step - loss:
0.7192 - accuracy: 0.8162 - val_loss: 1.1859 - val_accuracy: 0.7480
Epoch 3/10
128/128 [=====] - 48s 373ms/step - loss:
0.5274 - accuracy: 0.8453 - val_loss: 1.1791 - val_accuracy: 0.7495
Epoch 4/10
128/128 [=====] - 47s 369ms/step - loss:
0.4156 - accuracy: 0.8704 - val_loss: 1.2348 - val_accuracy: 0.7622
Epoch 5/10
128/128 [=====] - 48s 372ms/step - loss:
0.3513 - accuracy: 0.8864 - val_loss: 1.2772 - val_accuracy: 0.7573
Epoch 6/10
128/128 [=====] - 47s 367ms/step - loss:
0.2882 - accuracy: 0.9057 - val_loss: 1.3377 - val_accuracy: 0.7505
Epoch 7/10
128/128 [=====] - 47s 368ms/step - loss:
0.2456 - accuracy: 0.9187 - val_loss: 1.2596 - val_accuracy: 0.7696
Epoch 8/10
128/128 [=====] - 47s 365ms/step - loss:
0.2180 - accuracy: 0.9228 - val_loss: 1.2985 - val_accuracy: 0.7647
Epoch 9/10
128/128 [=====] - 47s 366ms/step - loss:
0.1850 - accuracy: 0.9367 - val_loss: 1.3346 - val_accuracy: 0.7520
Epoch 10/10
128/128 [=====] - 47s 365ms/step - loss:
0.1736 - accuracy: 0.9408 - val_loss: 1.3643 - val_accuracy: 0.7681

```

Как можно заметить, точность на тренировочных данных составляет ~94%, а на данных проверки ~77%.

Теперь начнем предсказание тестовых данных с помощью нашей модели.

```

test_df = pd.read_csv('sample_submission.csv')
breeds = test_df.columns[1:].tolist()
test_df['file_name'] = test_df['id'] + '.jpg'

from tensorflow.keras.preprocessing.image import ImageDataGenerator
test_gen = ImageDataGenerator(rescale=1.0 / 255)

```

```
test = test_gen.flow_from_dataframe(test_df,
                                    x_col='file_name',
                                    y_col=None,
                                    directory='test',
                                    class_mode=None,
                                    target=(224, 224))
```

Found 10357 validated image filenames.

Создадим таблицу с результатами для получения оценки на Kaggle

```
test_df.loc[:, breeds] = inception_resnet.predict(test)
test_df[breeds].head()
```

	affenpinscher	afghan_hound	african_hunting_dog	airedale \
0	0.000021	0.000149	0.000174	3.510588e-04
1	0.000053	0.000013	0.000003	7.597284e-07
2	0.000013	0.000023	0.000004	1.515151e-05
3	0.000189	0.001550	0.000060	1.888479e-04
4	0.000044	0.000011	0.000001	1.162445e-04

	american_staffordshire_terrier	appenzeller	australian_terrier
basenji \			
0	0.001242	0.000254	0.000088
0.002505			
1	0.000008	0.000301	0.000021
0.000006			
2	0.000053	0.000007	0.000153
0.000075			
3	0.001382	0.000371	0.000368
0.000577			
4	0.000091	0.000002	0.000009
0.000004			

	basset	beagle	...	toy_poodle	toy_terrier	vizsla
walker_hound \						
0	0.057014	0.368802	...	0.000041	4.749311e-03	0.013086
0.338184						
1	0.000012	0.000028	...	0.000014	1.599430e-05	0.000038
0.000159						
2	0.000063	0.000007	...	0.000028	8.674166e-07	0.000001
0.000004						
3	0.000068	0.000248	...	0.648986	3.320602e-03	0.000995
0.000043						
4	0.000027	0.000008	...	0.000073	8.851087e-07	0.000092
0.000045						

	weimaraner	welsh_springer_spaniel
west_highland_white_terrier \		
0	4.735024e-05	0.000030
		0.000054

1	7.571477e-05	0.000004	0.005308
2	7.783211e-07	0.000018	0.000289
3	1.616334e-04	0.000369	0.000602
4	1.699619e-05	0.000007	0.000013


	whippet	wire-haired_fox_terrier	yorkshire_terrier
0	0.000791	0.000103	0.000533
1	0.000001	0.000076	0.000006
2	0.000050	0.005386	0.000026
3	0.000579	0.000113	0.015526
4	0.000003	0.000009	0.000016

[5 rows x 120 columns]

```
test_df.drop('file_name',
axis=1).to_csv('InceptionResNet_submission.csv', index=False)
```

Загрузив поличившиеся результаты на Kaggle, получили оценку:

YOUR RECENT SUBMISSION



InceptionResNet_submission.csv

Submitted by Marsel Adilov · Submitted just now

Score: 8.94194

↓ Jump to your leaderboard position

Сравнение с двумя победителями

"Dog Breed: Transfer Learning combining 4 backbones"

In[20] автор преобразует изображение: случайно приближает, отзеркаливает и поворачивает изображение



****In[23**]** используется сразу 4 разных предобучающих модели (Xception, InceptionV3, InceptionResNetV2, NASNetLarge)

In[35] анализируются ошибки предсказания на тренировочной модели

"0.18 loss - Simple Features Extractors"

In [15] 4 разных предобученных модели (inception_features, xception_features, nasnet_features, inc_resnet_features)