Hard & Software II:
TensorFlow и Keras для задач компьютерного зрения

1. Введение

В этом модуле мы познакомимся с основными принципами, на которых основаны библиотеки для глубокого обучения, и подробно разберем библиотеки Keras и TensorFlow. В следующем модуле мы будем изучать библиотеку PyTorch.

> Keras в данный момент является частью библиотеки TensorFlow, хотя исторически эти библиотеки создавались независимо. Поэтому упоминая TensorFlow подразумеваем также и Keras, и наоборот.

В ЭТОМ МОДУЛЕ МЫ РАЗБЕРЁМ:

✅ На каких принципах основана работа библиотек для глубокого обучения?

✅ Как строить и обучать модели в Keras?

✅ Как подготавливать данные в Keras?

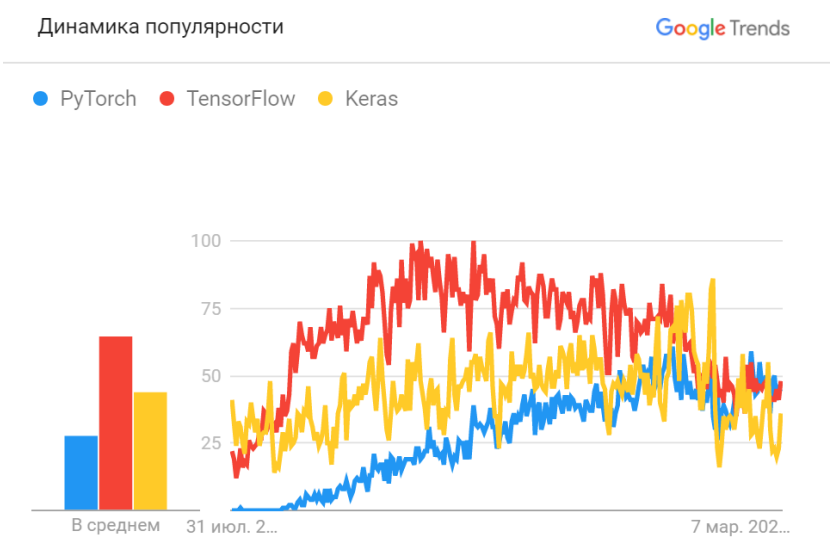
ВЫ НАУЧИТЕСЬ:

✅ Решать CV-задачу на Keras и TensorFlow: обучим модель, умеющую различать выражения лица человека с помощью датасета Facial expression comparison и самых современных на 2021 год сверточных сетей.

Почему мы изучаем сразу обе библиотеки: Keras и PyTorch?

Специалист по компьютерному зрению должен знать обе эти библиотеки, поскольку в работе вам часто придется использовать чужой код, а он может быть написан как на TensorFlow/Keras, так и на PyTorch. В дальнейшем в курсе CV-инженер вы в основном будете работать с библиотекой PyTorch, но в первом году обучения вы работали с библиотекой Keras. Мы также начнем с Keras, а затем изучим PyTorch.

Динамика популярности поисковых запросов «TensorFlow», «Keras» и «PyTorch» за последние 5 лет:



TensorFlow разработан в компании Google, PyTorch – в компании Facebook. Автором библиотеки Keras является François Chollet, работающий сейчас в Google. Постепенно библиотека Keras стала частью TensorFlow.

Как и в других разделах, материал модуля доступен в виде скринкастов (видео), ноутбуков и текста разделов под скринкастами. Наиболее полно материал изложен в тексте и на скринкасте, то есть **все, что проговаривается в скринкасте, есть в тексте под скринкастом**. В ноутбуках материал изложен в сокращенном варианте: в нем есть весь код, но не все пояснения.

Для освоения этого модуля вам может потребоваться повторить следующие понятия:

- Граф вычислений
- Нейронная сеть
- Обратное распространение ошибки
- Стохастический градиентный спуск
- Обучение мини-батчами
- Инференс

Инференс – это это процесс получения предсказаний модели на некоторых входных данных. *Некоторые слои нейронных сетей работают по-разному в режимах обучения и инференса.* Например, такие слои как "dropout", "gaussian dropout" или "batch normalization" (мы изучим их в дальнейшем) в процессе своей работы вносят "шум" в данные. Доказано, что это полезно для обучения и повышает обобщающую способность сети, то есть эти слои являются способами регуляризации. Однако если наша цель – получить на конкретных данных максимально точные предсказания, то мы больше не хотим вносить шум в данные. То есть dropout-слои отключаются в режиме инференса, а слой batch normalization меняет принцип своей работы.

Давайте перед началом модуля проверим свои знания.

Задание 1.1

Выберите утверждения, которые верны для **всех** искусственных нейронных сетей.

Примечание: здесь имеются в виду нейронные сети в глубоком обучении. Мы не рассматриваем нейронные сети, копирующие биологические сети (такие как проект Human Brain Project).

☑ Нейронные сети являются графами вычислений.

☐ Состоят из искусственных нейронов (перцептронов).

☐ Содержат один выходной слой нейронов.

Подсказка. Нейронные сети не всегда состоят из нейронов. Они могут содержать самые разнообразные векторные и матричные операции, которые не всегда представимы в виде слоя нейронов. Сеть может иметь несколько входов и/или несколько выходов. Например, один выходной слой сети выдает 4 числа, описывающих координаты лица человека на фото, а другой выходной слой выдает одно число, описывающее его пол (логистическая регрессия).

Задание 1.2

Выберите верные утверждения.

☐ После полносвязного слоя всегда должна идти нелинейная функция активации.

☐ В ходе обратного распространения ошибки рассчитываются производные предсказания (выходных данных) сети по всем весам сети.

☑ Выбор оптимизатора (SGD, adam и т. д.) не влияет на алгоритм расчета градиентов.

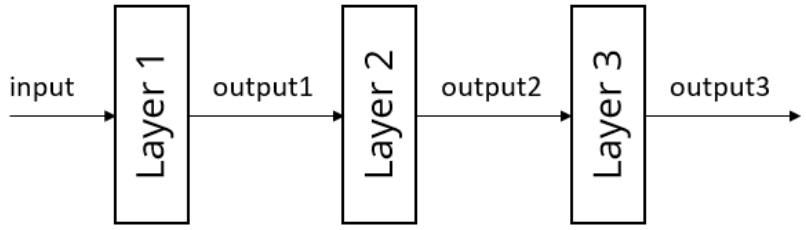
☑ Некоторые слои нейронных сетей работают по-разному в режиме обучения и инференса.

Подсказка. Выходной полносвязный слой может не иметь функции активации (или, что то же самое, его функцией активации может являться тождественная функция $f(x) = x$). В ходе обратного распространения ошибки рассчитываются производные *функции потерь* по весам сети. Оптимизатор влияет только на алгоритм обновления весов, используя градиент.

2. Принципы работы библиотек для глубокого обучения

Как вы помните, нейронная сеть – это последовательность векторных и матричных операций. Такие операции можно реализовать даже с помощью numpy. Для инференса этого будет достаточно. Но если мы хотим обучать сеть - то мы должны реализовать **алгоритм обратного распространения ошибки**. Именно это и есть ключевая возможность библиотек для глубокого обучения.

Рассмотрим модель из нескольких полносвязных (или любых других) слоев:



Чтобы воспользоваться алгоритмом обратного распространения ошибки нам нужно запоминать выходы всех слоев: *output1*, *output2*, *output3*. Если же мы делаем инференс, то после получения *output2* можно удалить из памяти *output1* - он нам больше не нужен. Аналогично, получив *output3* удаляем *output2*. Поэтому для обучения требуется существенно больше оперативной памяти, чем для инференса. За счет этого при инференсе можно использовать больший размер батча.

В библиотеках TensorFlow и PyTorch веса и выходные данные каждого слоя сети хранятся не в numpy-массивах, а в так называемых тензорах (объектах `tf.Tensor`, `tf.EagerTensor`, `torch.Tensor`). Основные отличия тензора от numpy-массива следующие:

1. Тензоры как правило хранятся в памяти GPU или TPU
2. История операций над тензорами (таких как сложение, умножение) запоминается таким образом, чтобы по графу вычислений можно было «пройти в обратную сторону» и рассчитать градиенты.

> Уточнение: понятие «тензор» в глубоком обучении не эквивалентно математическому понятию тензора, это разные вещи.

Помимо основной функциональности (дифференцирование графа вычислений) библиотеки TensorFlow и PyTorch также представляют широкий набор дополнительных возможностей:

- Большое количество разнообразных слоев для нейронных сетей и оптимизаторов
- Построение пайплайнов (pipeline): скачивание датасетов, объединение в батчи, аугментации
- Доступны наборы уже обученных нейронных сетей
- Модели можно сохранять в файлы и загружать из файлов

Статический и динамический граф вычислений

Существует два основных подхода к построению графов вычислений.

При *декларативном* (то есть описательном) подходе граф вычислений строится заранее, а затем по нему «пропускаются» данные. Выполняя операции над тензорами

мы не выполняем вычислений, а только строим граф (это напоминает построение трубопровода). Затем весь граф целиком мы запускаем одной командой. Граф вычислений при этом называется *статическим*.

При *императивном* подходе выполняя операции над тензорами мы сразу получаем результат. При этом граф вычислений не описывается заранее и называется *динамическим*.

Давайте разберем основные плюсы и минусы обоих подходов.

Статический граф вычислений
+ Библиотека «знает заранее» порядок операций в графе. Это позволяет эффективно оптимизировать граф в плане скорости вычислений и потребляемой памяти.
— Сложнее писать и отлаживать код по сравнению с динамическим графом вычислений.

Динамический граф вычислений
+ Код проще и интуитивно понятнее.
— Библиотека «не знает» какую операцию вы выполните следующей, что затрудняет оптимизацию скорости вычислений и потребляемой памяти. Однако есть способ преодолеть эту проблему путем анализа исходного кода модели и JIT-компиляции.

В **PyTorch** используется динамический граф вычислений.

В **TensorFlow версии 1** использовался только статический граф вычислений, что усложняло написание кода и заставило многих разработчиков перейти на PyTorch.

В **TensorFlow версии 2** граф вычислений можно строить как динамически (eager mode), так и статически (graph mode). Однако мы не будем писать код на чистом TensorFlow, а будем использовать Keras.

В **Keras** граф вычислений преимущественно статический, однако при желании можно переключить его в динамический режим (eager mode).

В целом разработчики TensorFlow придерживаются точки зрения о том, что по возможности лучше использовать статический граф вычислений для повышения производительности.

Разница на практике между статическим и динамическим графом вычислений станет более понятна из следующих разделов. При построении простых моделей этот момент не принципиален, он приобретает важность только если вы делаете что-то сложное и нестандартное.

3. Построение моделей в Keras

Начнем с импорта библиотеки:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Model, Sequential, layers
import numpy as np

print(tf.__version__) #2.5.0 (на 30.07.21) или выше
```

Немного истории. Раньше Keras был отдельной библиотекой, для которой были доступны разные вычислительные «бэкенды»: TensorFlow, Theano, CNTK, PlaidML. При этом Keras описывает высокоуровневые абстракции, такие как полносвязный слой и цикл обучения, и конвертирует их в более низкоуровневые инструкции, такие как матричное умножение, сложение, прямой и обратный проход. Эти операции выполняются с помощью бэкенда.

Начиная с версии 2.3 Keras поддерживает TensorFlow 2, а с версии 2.4 больше не поддерживает другие бэкенды кроме TensorFlow. С этого момента Keras становится частью библиотеки TensorFlow.

Keras и сейчас доступен в виде отдельной библиотеки (`!pip install keras`), но такой вариант импорта (`import keras`) считается устаревшим и может неправильно работать с версиями TensorFlow 2.5 и выше.

Построение модели Sequential

Для создания моделей в Keras есть два основных класса: `Sequential` и `Model`.

Класс `Sequential` является подклассом `Model` и описывает модель, состоящую из цепочки слоев, выполняющихся последовательно. Например, так мы создадим модель с двумя скрытыми слоями по 100 нейронов и функцией активации ReLU и выходным слоем из 10 нейронов с функцией активации softmax:

```
model = Sequential([
  layers.InputLayer(input_shape=(50,)),
  layers.Dense(100, 'relu'),
  layers.Dense(100, 'relu'),
  layers.Dense(10, 'softmax')
])
```

Примечание: в модель `Sequential` можно добавлять новые слои методом `.add()`, но в таком действии редко бывает необходимость. Удобнее сразу передавать в конструктор `Sequential` все слои в виде списка.

Слой **`InputLayer`** не выполняет вычислений и лишь указывает размер входных данных сети. Мы можем убрать этот слой и вместо этого указать параметр `input_shape` у первого полносвязного слоя:

```
model = Sequential([
  layers.Dense(100, 'relu', input_shape=(50,)),
  layers.Dense(100, 'relu'),
  layers.Dense(10, 'softmax')
])
```

Слои модели хранятся в виде списка в атрибуте **`.layers`**. Напечатать структуру модели можно вызывав метод **`.summary()`**:

```
model.summary()
```

```
Model: "sequential_19"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_37 (Dense)             (None, 100)               5100
_________________________________________________________________
dense_38 (Dense)             (None, 100)               10100
```

```
_________________________________________________________________
dense_39 (Dense)             (None, 10)                1010      
=================================================================
Total params: 16,210
Trainable params: 16,210
Non-trainable params: 0
```

Каждому слою присвоено уникальное имя, для каждого слоя созданы массивы весов, общее количество весов каждого слоя указано в столбце `Param #`. Снизу указано суммарное количество весов сети. Поскольку у полносвязных слоев все веса обучаемы, то количество обучаемых весов равно общему количеству весов. В столбце Output Shape по первой оси указан размер None. Эта ось отвечает за размер батча – он может быть любым.

Любую **модель можно вызвать как функцию**, передав в качестве параметра батч входных данных и получив батч выходных данных в виде тензора. Тензор затем можно преобразовать в numpy-массив методом `.numpy()`:

```
batch_size = 64
input_batch = np.zeros((batch_size, 50))
output = model(input_batch).numpy() #получаем массив размером (64, 10)
```

Пояснение: если некий python-объект вызывается как функция, то неявно для этого объекта вызывается метод `.__call__()`.

При вызове модели как функции можно передать дополнительный параметр **`training=True`**, который означает запуск модели в режиме обучения (например, будет работать слой Dropout). По умолчанию модель запускается в режиме инференса.

```
model = Sequential([
  layers.Dropout(0.999)
])

print(model(np.ones((1, 5))).numpy())
print(model(np.ones((1, 5)), training=True).numpy())
```

```
[[1. 1. 1. 1. 1.]]
[[0. 0. 0. 0. 0.]]
```

Вспомним, что слой Dropout случайным образом зануляет указанную долю значений в тензоре, передаваемом через этот слой. Этот слой работает в режиме обучения и не работает в режиме инференса.

При создании модели мы можем вообще не указывать размер входных данных (ни с помощью `InputLayer`, ни с помощью `input_shape` первого слоя). В этом случае размер входных данных будет рассчитан автоматически при первом вызове модели. В этот же момент будут созданы веса слоев. Если попытаться получить summary по модели до того, как созданы веса слоев, то будет выброшено исключение: `This model has not yet been built.`

Задание 3.1

Постройте модель, состоящую из трех последовательно соединенных полносвязных слоев с 300, 200 и 100 нейронами. Модель принимает на вход вектор длиной 400. Сколько весов суммарно у этой сети?

Примечание: функция активации в данном задании не имеет значения.

Ответ: 200600 или 200,600.

Решение:

```
model = Sequential([
  layers.InputLayer(400),
  layers.Dense(300),
  layers.Dense(200),
  layers.Dense(100),
])
model.summary()
```

Построение модели в функциональном стиле

Построение модели через конструктор класса **`Model`** – более общий способ, позволяющий строить разветвленные графы вычислений. Модель, аналогичная той, что мы строили выше, с функциональном стиле строится так:

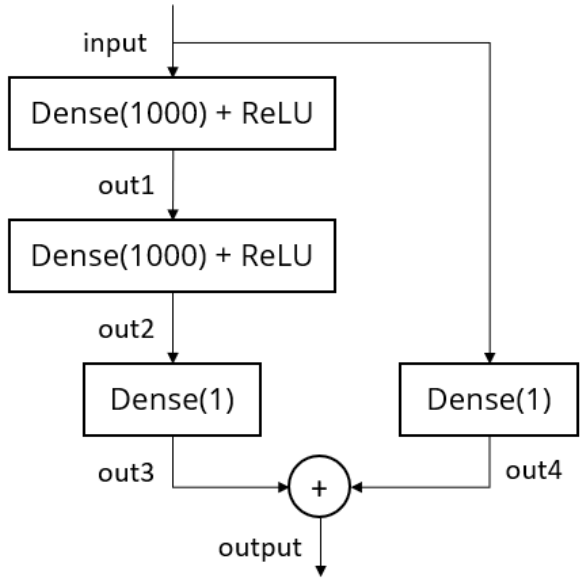
```
input = keras.Input(shape=(50,))
output1 = layers.Dense(100, 'relu')(input)
output2 = layers.Dense(100, 'relu')(output1)
output3 = layers.Dense(10, 'softmax')(output2)
model = Model(inputs=input, outputs=output3)
```

На этом примере хорошо видно, что мы строим статический граф вычислений (*см. предыдущий раздел*). Модель `Sequential` также является статическим графом.

Сначала мы указываем размер входных данных, затем создаем слои и вызываем каждый слой как функцию. При этом слой принимает и возвращает объект класса `KerasTensor`. При этом пока что не происходит никаких вычислений, мы лишь указываем последовательность операций. Затем мы вызываем конструктор класса `Model`, передав в него начало и конец построенного графа.

Лучше по возможности избегать функционального стиля, так как код получается сложнее, а значит больше шанс сделать ошибку и сложнее вносить изменения. Строить модель в функциональном стиле имеет смысл только при разветвленном графе вычислений.

Для примера предположим, что у нас есть глубокая полносвязная нейронная сеть (MLP) из 3 слоев, решающая задачу регрессии на табличных данных (50 признаков). Но при этом у нас есть подозрение, что ответ может линейно зависеть от некоторых входных признаков. Для того, чтобы сети было проще выучить такую зависимость, мы делаем «проброс» связи через всю сеть (skip connection).



Давайте построим эту модель:

```
input = keras.Input(shape=(50,))
out1 = layers.Dense(1000, 'relu')(input)
out2 = layers.Dense(1000, 'relu')(out1)
out3 = layers.Dense(1)(out2)
out4 = layers.Dense(1)(input)
output = out3 + out4
model = Model(inputs=input, outputs=output)
```

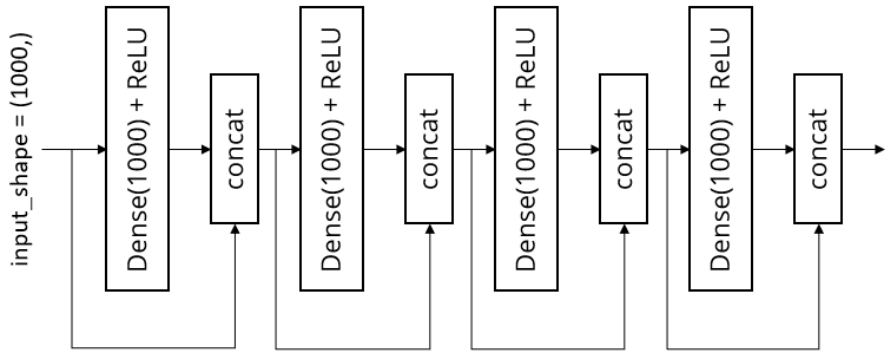
В предпоследней строке мы складываем друг с другом два промежуточных выхода модели (два объекта `KerasTensor`). Важно понимать, что *с точки зрения языка python* в данном коде используется перегрузка операторов, например `out3 + out4` вызывает `out3.__add__( out4)`. За счет этого мы можем складывать тензоры как числа. Также можно использовать следующий синтаксис: `tf.math.add(out3, out4)` или `layers.Add()([out3, out4])`. Результат во всех случаях будет эквивалентен.

У модели может быть несколько входов или выходов. Например, если мы хотим, чтобы модель возвращала не только `output`, то и `out3`, то мы создадим ее таким образом:

```
model = Model(inputs=input, outputs=[output, out3])
```

Задание 3.2

Одна из известных сверточных архитектур называется DenseNet. Она состоит из блоков, в которых каждый слой связан со всеми последующими. Попробуем построить ее аналог, но в нашем случае слои будут полносвязными. Архитектура сети показана на иллюстрации ниже.



Операция concat означает конкатенацию и выполняется с помощью слоя `layers.Concatenate()`. Посмотрите документацию по этому слою с примерами использования и постройте данную сеть с помощью Keras. Сколько весов суммарно у получившейся сети?

Ответ: 10004000 или 10,004,000.

Решение:

```
input = keras.Input(shape=(1000,))
output = input
for _ in range(4):
  output = layers.Concatenate()([output, layers.Dense(1000, 'relu')(output)])
model = Model(inputs=input, outputs=output)
model.summary()
```

Построение модели в комбинированном стиле

Если ветвления требуют от нас использовать функциональный стиль, то это не означает, что всю модель нужно строить в функциональном стиле. Можно использовать комбинированный подход:

```
input = keras.Input(shape=(50,))
branch1 = Sequential([
```

```
  layers.Dense(1000, 'relu'),
  layers.Dense(1000, 'relu'),
  layers.Dense(1)
])
branch2 = layers.Dense(1)
output = branch1(input) + branch2(input)
model = Model(inputs=input, outputs=output)
```

Иерархические модели

Класс `Model` является подклассом класса `Layer`, то есть каждая модель является слоем. Это означает, что модель можно использовать в качестве слоя другой модели. Например, в модуле `keras.applications` есть сеть `Xception`. Это сверточная сеть довольно сложной архитектуры, построенная в функциональном стиле. На странице документации вы можете посмотреть ее исходный код, нажав «View source on GitHub».

Сеть Xception является моделью, но ее можно использовать в качестве слоя другой модели, например таким образом:

```
from tensorflow.keras.applications import Xception
model = Sequential([
  Xception(include_top=False, input_shape=(150, 150, 3)),
  layers.GlobalMaxPool2D(),
  layers.Dropout(0.5),
  layers.Dense(10)
])
```

> Мы сейчас не рассматриваем подробно сеть Xception и слой `GlobalMaxPool2D`, мы изучим их позже. Сейчас это лишь иллюстрация построения вложенных моделей в Keras.

У получившейся модели 4 слоя, а у ее первого слоя (сети Xception) 132 слоя:

```
print(len(model.layers)) #4
print(len(model.layers[0].layers)) #132
```

Для сравнения, мы могли бы построить ту же самую модель таким образом:

```
base_model = Xception(include_top=False, input_shape=(150, 150, 3))
output = layers.GlobalMaxPool2D()(base_model.output)
output = layers.Dropout(0.5)(output)
output = layers.Dense(10)(output)
model = Model(inputs=base_model.input, outputs=output)
```

В этом случае `Xception` уже не будет вложенной моделью, все слои `Xception` станут слоями `model`. У модели `model` будет 135 слоев.

Один и тот же слой может принадлежать нескольким моделям. Например, пусть у нас есть модель из трех полносвязных слоев:

```
model = Sequential([
  layers.InputLayer(400),
  layers.Dense(300),
  layers.Dense(200),
  layers.Dense(100),
])
```

Если мы хотим получить выходное значение второго слоя модели, то можно запустить по очереди первый и второй слой:

```
input_data = np.zeros((64, 400))
l1, l2, l3 = model.layers
layer2_output = l2(l1(input_data)).numpy()
```

Но есть и другой способ: создать еще одну модель, входом которой является вход исходной модели, а выходом – выход второго слоя исходной модели. Это не приведет к копированию слоев или весов, новая модель будет использовать слои и веса старой модели.

```
submodel = Model(inputs=model.input, outputs=model.layers[1].output)
layer2_output = submodel(input_data).numpy()
```

В данном случае модель `model` является графом вычислений, а модель `submodel` является подграфом этого графа.

~~Задание 3.3 (не особенно важное, решил убрать)~~

~~Постройте Sequential-модель, аналогичную описанной выше, но оберните последние три слоя в еще одну вложенную модель Sequential. У получившейся модели должно быть всего 2 слоя: Xception и Sequential из трех последних слоев. После построения модели проверьте, чему равно следующее значение:~~

```
model.layers[-1].input.shape
```

~~◉ TensorShape([None, 5, 5, 2048]).~~

~~○ TensorShape([None, 2048]).~~

~~○ TensorShape([None, 150, 150, 3]).~~

~~Решение:~~

```
from tensorflow.keras.applications import Xception
model = Sequential([
  Xception(include_top=False, input_shape=(150, 150, 3)),
  Sequential([
    layers.GlobalMaxPool2D(),
    layers.Dropout(0.5),
    layers.Dense(10)
  ])
])
model.layers[-1].input.shape
```

Сохранение и загрузка моделей

Можно сохранить и загрузить модель из файла следующим кодом:

```
model.save('model.h5') #сохранение
model = keras.models.load_model('model.h5') #загрузка
```

Можно также загрузить в уже созданную модель веса из файла (`model.load_weights`), но обычно это не требуется. Расширения `.h5` и `.hdf5` эквивалентны и означают один и тот же формат файла - Hierarchical Data Format версии 5.

Дополнительные материалы

Документация по TensorFlow и Keras

Официальные гайды по TensorFlow

Еще один сайт с официальной документацией и гайдами по Keras

Документация по Model, Sequential, Input

4. Слои и их параметры в Keras

Модуль keras.layers содержит большое количество разнообразных слоев, из которых строятся нейронные сети. В этом разделе мы изучим полносвязные слои, функции активации, считывание и изменение весов слоя, а также кратко рассмотрим как выполнять произвольные операции в графе.

Так же как и модель, слой можно запустить как функцию, передав батч входных данных и получив результат. Если модель состоит из цепочки последовательно соединенных слоев, то запустив по очереди каждый слой получим тот же результат, как если бы запустили модель целиком.

Считывание и изменение весов слоя

Веса слоя могут быть получены в виде numpy-массива методом **`.get_weights()`**. Например, полносвязный слой размером N нейронов с `input_shape=(M,)` вернет пару из матрицы весов размером `(M, N)` и матрицы bias'ов размером `(N,)`.

Вручную изменить веса слоя можно методом **`.set_weights()`**. Этот метод принимает данные в том же формате, в каком возвращает их метод `.get_weights()`. Например, таким образом можно обнулить bias'ы полносвязного слоя с индексом 0:

```
layer = model.layers[0]
W, b = layer.get_weights()
layer.set_weights((W, np.zeros_like(b)))
```

Если вызвать функцию `.get_weights()` для модели (а не для слоя), то функция вернет кортеж (tuple) из всех весов модели. Например, если модель состоит из 3 полносвязных слоев, то функция `.get_weights()` вернет кортеж из 6 массивов: W1, b1, W2, b2, W3, b3.

При создании слоя не в составе модели веса слоя не будут созданы. Параметр `input_shape` будет проигнорирован. Для создания весов нужно один раз запустить слой, передав в него входные данные, либо вызвать метод `.build(input_shape)`:

```
layer = layers.Dense(100)
print([x.shape for x in layer.get_weights()])
layer.build((20,)) #или так: layer(np.zeros((1, 20)))
print([x.shape for x in layer.get_weights()])

[]
[(20, 100), (100,)]
```

Задание 4.1

Создайте уже известную нам из предыдущего раздела модель Xception. Переберите в цикле все ее слои и посчитайте суммарное количество весов для каждого слоя. Постройте график зависимости количества весов от номера слоя. Какой вывод можно сделать?

○ В первых слоях весов в среднем больше, чем в последних.

◉ В последних слоях весов в среднем больше, чем в первых.

○ Веса распределены равномерно по всей сети.

Решение:

```
from tensorflow.keras.applications import Xception
model = Xception(include_top=False, input_shape=(150, 150, 3))

def count_weights_in_layer(layer):
  array_lengths = [len(W.flat) for W in layer.get_weights()]
  return sum(array_lengths)

W = [count_weights_in_layer(layer) for layer in model.layers]

import matplotlib.pyplot as plt
plt.plot(W)
plt.show()
```

Пояснение к решению. В сверточных сетях как правило наибольшее количество весов сосредоточено в последних слоях. Колебания количества весов (вид графика в виде «расчески») обусловлены тем, что в некоторых слоях весов вообще нет, например в слое `Activation` и `MaxPooling2D`.

Функции активации в Keras

Вы уже знаете как создать полносвязный слой с выбранным количеством нейронов и функцией активации ReLU.

```
from tensorflow.keras import layers
layers.Dense(1000, activation='relu')
```

Если мы не указываем параметр activation, то получим линейный слой без функции активации. Функцию активации можно указать в виде отдельного слоя. Рассмотрим несколько примеров кода.

```
layers.Dense(1000),
layers.Activation('relu')
```

В этом примере мы создаем полносвязный слой без функции активации, а затем функцию активации как отдельный слой.

Для некоторых часто используемых функций активации, например ReLU или Softmax, в Keras существуют собственные классы:

```
model = Sequential([
  layers.Dense(100),
  layers.ReLU(),
  layers.Dense(10),
  layers.Softmax()
])
```

Когда мы использовали слой `Activation` или параметр `activation` в слое `Dense`, мы передавали в него название функции активации в виде строки. В этом случае Keras ищет ищет функцию активации с данным названием, для чего неявно вызывается следующий метод:

```
keras.activations.get('relu')
```

Данный метод ищет функцию активации по названию и возвращает ее в виде функции. Мы можем сразу указать ее в виде не строки, а функции:

```
layers.Dense(1000, activation=keras.activations.relu)
```

В следующих разделах мы будем работать с функциями потерь, оптимизаторами и метриками. Там действует тот же принцип: если функция потерь, оптимизатор или метрика указаны в виде строки, то Keras ищет соответствующий класс по названию. Но вместо названия в виде строки мы можем сразу указать нужный нам объект. В примере выше это функция `keras.activations.relu`.

Библиотека Keras изначально поддерживала разные вычислительные бэкенды. Сейчас она поддерживает только TensorFlow, и поэтому вместо функции-обертки из модуля `keras.activations` мы можем сразу указывать нужную нам операцию в TensorFlow:

```
layers.Dense(1000, activation=tf.nn.relu)
```

Здесь начинает проявляться гибкость в построении моделей на Keras. Вместо `tf.nn.relu` мы можем выполнить вообще любую операцию. Представим, что в одной из недавних статьей авторы предлагают использовать функцию активации $f(x) = \sin(x) + x$ и бьют с этой функцией рекорды точности. Мы можем легко реализовать эту функцию в нашей модели:

```
my_activation = lambda x: tf.math.sin(x) + x
l = layers.Dense(1000, activation=my_activation)
```

Полносвязный слой в Keras

Полносвязный слой `Dense` принимает на вход **массив данных с двумя осями**: первая ось отвечает за номер примера в батче, вторая ось за номер входного признака. Возвращает слой также массив с двумя осями, но теперь вторая ось отвечает за номер выходного нейрона.

Строго говоря, слой `Dense` может принимать и массив данных с большим количеством осей, но это работает неочевидным образом и не рекомендуется к использованию.

Давайте изучим остальные параметры, доступные в конструкторе полносвязного слоя.

```
use_bias=True #Boolean, whether the layer uses a bias vector.
```

Если выберем значение `False`, то слой не будет выполнять сложение, а только матричное умножение и (опционально) функцию активации.

```
kernel_initializer="glorot_uniform" #Initializer for the kernel weights matrix.
bias_initializer="zeros" #Initializer for the bias vector.
```

Способ инициализации весов. Матрица весов инициализируется способом glorot uniform (или xavier uniform, что то же самое), а вектор bias'ов инициализируется нулями. Здесь работает тот же принцип, что описан выше для функций активации: если нечто указано в виде строки, то идет поиск класса по названию. Например, строка `"glorot_uniform"` соответствует классу `keras.initializers.GlorotUniform`. Подробнее про инициализацию весов можно прочитать на странице документации.

Инициализация весов слоя каждый раз происходит случайным образом. Создав одинаковую модель дважды, мы получим две модели с разными начальными весами.

Если хочется достичь воспроизводимости результатов, то перед созданием модели можно зафиксировать зерно случайного генератора:

```
tf.random.set_seed(0)
```

Есть и еще одна тонкость: при обучении сверточных сетей на GPU процесс обучения идет недетерминированно, поэтому даже две сети с одинаковыми начальными весами после обучения станут разными.

Остальные параметры полносвязного слоя:

```
kernel_regularizer=None,
bias_regularizer=None,
activity_regularizer=None,
kernel_constraint=None,
bias_constraint=None
```

С помощью этих параметров можно указать способ регуляризации весов и ограничения, накладываемые на веса. Мы сейчас не будем углубляться в эти темы. Эти параметры также можно задавать в виде строк или объектов.

Слой Flatten

Как мы помним, полносвязный слой принимает на вход массив данных с двумя осями, то есть каждому примеру в батче соответствует вектор признаков. Что делать, если на входе у нас изображение, которое имеет две оси (черно-белое) или три оси (цветное), а мы хотим использовать для его обработки полносвязный слой? В этом случае нам нужно применить сначала слой `Flatten()`.

Слой `Flatten()` не имеет весов и «вытягивает в вектор» массив данных по каждому обучающему примеру, давая на выходе массив с двумя осями. Давайте посмотрим, как эта операция выглядела бы на numpy:

```
input = np.zeros((64, 256, 256, 3))
output = input.reshape(len(input), -1)
output.shape #(64, 196608)
```

Давайте посмотрим как функционирует модель с этим слоем:

```
from tensorflow.keras import layers, Sequential
model = Sequential([
  layers.Flatten(),
  layers.Dense(100, 'relu'),
  layers.Dense(10, 'softmax')
])

input = np.zeros((64, 256, 256, 3))
model(input).numpy().shape #(64, 10)
```

Задание 4.2

Сколько весов в примере выше имеет скрытый полносвязный слой после первого запуска модели?

Ответ: 19660900 или 19,660,900.

Решение: слой Flatten выдает массив размером (64, 256*256*3) = (64, 196608). Поэтому скрытый полносвязный слой имеет 100 нейронов и принимает на вход 196608 признаков. Матрица весов слоя имеет размер 196608*100, матрица bias'ов имеет размер 100. В сумме получаем 19660900 весов.

На практике если входные данные имеют больше 2 осей – то как правило это либо изображение, либо некая последовательность, например векторизованный текст. В обоих случаях используются специальные слои, которые не требуют слоя `Flatten()`. Забегая вперед можно сказать, что иногда слой `Flatten()` можно встретить в сверточных сетях после последнего сверточного слоя. Мы рассмотрим это в следующих модулях.

Lambda-слой

Нередко возникает необходимость встроить в граф вычислений какое-то нестандартное действие, для которого отсутствует уже готовый слой. Такие нестандартные действия бывают двух типов:

Stateless – выполнение операции, которая не содержит обучаемых параметров, например прибавление единицы к каждому элементу массива.

Stateful – выполнение операции, которая использует обучаемые параметры. Например это может быть вариация полносвязного слоя, где вместо матричного умножения выполняется скалярное произведение.

Для первого типа операций используется слой `Lambda`, для второго типа – подкласс класса `Layer`.

Слой **`Lambda`** принимает в качестве параметра функцию, которую нужно выполнить над входными данными. Эта функция может содержать операции TensorFlow (такие как `tf.nn.relu`, `tf.math.sin`, `tf.math.maximum` и многие другие) и арифметические операции. Последние по сути также являются операциями TensorFlow, так как используется перегрузка операторов (подробнее см. в разделе «Построение модели в функциональном стиле»).

Например, один из вариантов нормализации входных данных – деление значений пикселей входного изображения на 255. Это можно реализовать, добавив в начало модели следующий слой:

```
layers.Lambda(lambda x: x/255)
```

Хотя для данной цели в Keras есть специальный слой:

```
layers.experimental.preprocessing.Rescaling(1/255)
```

Задание 4.3

Рассмотрим следующую модель:

```
model = Sequential([
  layers.Lambda(lambda x: x/255),
  layers.Dense(1000, 'relu', use_bias=False),
  layers.Dense(1000, 'relu', use_bias=False),
  layers.Dense(1),
])
```

После того, как мы создали и обучили эту модель, мы перенесли слой Lambda из самого начала в самый конец модели:

```
model_new = Sequential(model.layers[1:] + model.layers[:1])
```

Повлияло ли то, что мы сделали, на функционирование модели, или же новая модель `model_new` всегда будет выдавать в точности такой же результат, что и старая модель `model`?

○ Однозначно повлияло.

○ Однозначно не повлияло.

◉ Зависит от точности вычислений.

Решение. Полносвязный слой модели с `use_bias=False` и функцией активации ReLU может быть описан такой формулой:

$$\text{output} = \max(0, \text{input@weights})$$

По формуле становится понятно, что умножение input на некую константу C эквивалентно умножению результата на C. Это же верно и для суперпозиции трех таких слоев. Это говорит о том, что мы можем либо разделить входные данные сети на 255, либо разделить выходные данные сети на 255 – результат останется тем же. Значит такой перенос слоя не должен влиять на функционирование сети.

Однако вспомним, что в компьютере вычисления выполняются с ограниченной точностью. В нейронных сетях иногда используют тип float16, который занимает всего 2 байта, а значит погрешность вычислений очень высокая. Это приводит к тому, что порядок операций играет роль, это можно проверить:

```
keras.backend.set_floatx('float16')
tf.random.set_seed(0)
model = Sequential([
  layers.Lambda(lambda x: x/255),
  layers.Dense(1000, 'relu', use_bias=False),
  layers.Dense(1000, 'relu', use_bias=False),
  layers.Dense(1),
])
print(model(np.ones((1, 10))))
model = Sequential(model.layers[1:] + model.layers[:1])
print(model(np.ones((1, 10))))
```

```
tf.Tensor([[0.0002923]], shape=(1, 1), dtype=float16)
tf.Tensor([[0.0002928]], shape=(1, 1), dtype=float16)
```

Конечно, в данном случае погрешность лишь в 4-й значащей цифре, но в целом о погрешности вычислений не стоит забывать, она может проявить себя в неожиданные моменты.

Заморозка весов слоя и модели

У слоев и моделей есть атрибут **`.trainable`**, который говорит о том, будут ли веса данного слоя обучаться или будут заморожены. Если веса слоя заморожены, то оптимизатор не будет обновлять эти веса в ходе обучения. Заморозка весов может пригодиться если мы используем уже обученную модель, дообучая только отдельные ее слои.

Изменение значения атрибута `.trainable` у модели приведет к рекурсивному изменению этого атрибута у всех ее слоев и вложенных моделей.

```
model = Sequential([
  layers.Dense(100, 'relu'),
  layers.Dense(10),
])

assert model.layers[0].trainable == True
model.trainable = False
assert model.layers[0].trainable == False
```

После изменения атрибута `.trainable` нужно заново скомпилировать модель, чтобы изменения имели эффект. О компиляции и обучении мы будем говорить в следующем разделе.

5. Компиляция моделей в Keras

Процесс обучения модели начинается с ее компиляции. При этом мы указываем функцию потерь, оптимизатор и метрики качества. Пример:

```
model = Sequential([
  layers.InputLayer((100,)),
  layers.Dense(500, 'relu'),
  layers.Dense(10, 'softmax')
])
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='adam',
    metrics='accuracy'
)
```

Как обычно, параметры в функции `.compile()` можно указывать в виде строк или объектов. Рассмотрим типичные значения этих параметров.

Функции потерь

Вспомним принцип работы функции потерь. Имея батч предсказаний `y_pred` и батч эталонных ответов `y_true` мы можем посчитать значения функции потерь для каждого примера в батче, сравнивая предсказания с эталонными ответами. Затем мы можем посчитать сумму или среднее значений функции потерь по всем примерам в батче, получив одно число. Минимизация этого числа будет означать минимизацию функции потерь по каждому примеру независимо.

В Keras функции потерь доступны в виде функций или объектов. Они взаимозаменяемы, например можно использовать `keras.losses.mean_squared_error` или `keras.losses.MeanSquaredError()`, разницы не будет.

Давайте изучим работу этих функций на примере:

```
batch_size = 8
y_true = np.zeros((batch_size, 10))
y_pred = np.ones((batch_size, 10))
print(keras.losses.mean_squared_error(y_true, y_pred).numpy())
print(keras.losses.MeanSquaredError()(y_true, y_pred).numpy())
```

```
[1. 1. 1. 1. 1. 1. 1. 1.]
1.0
```

Как видим, во втором варианте считается среднее значение функции потерь по всему батчу, а в первом варианте возаращается значение функции потерь по каждому примеру. Если говорить еще точнее, то в первом случае усреднение делается только по последней оси, а батч выходных данных иногда может иметь много осей (например, если выходом сети является изображение).

Таким образом, что при использовании mean_squared_error результатом будет не один скаляр, а массив чисел. Однако расчет градиентов в Keras способен работать с целым массивом функций потерь, считая его сумму. Это означает, что если усреднение не было сделано при расчете функции потерь, то оно будет сделано при расчете градиентов.

Все это означает, что функции потерь, заданные в виде функций или объектов, взаимозаменяемы.

Также вспомним, что `loss` и другие параметры можно задавать в виде строк. Если в параметр `loss` передана строка, то keras будет искать по названию среди известных ему loss'ов.

Функция потерь (из модуля `keras.losses`)	Описание

`'mse'`
`'mean_squared_error'`
`mean_squared_error`
`MeanSquaredError()`
	Среднеквадратичная ошибка
	$\mathrm{MSE} = \mathrm{mean}((y - \hat{y})^2)$

`'bce'`
`'binary_crossentropy'`
`binary_crossentropy`
`BinaryCrossentropy()`
	Бинарная кроссэнтропия («logloss»)
	$\mathrm{BCE} = \mathrm{mean}(-y \log \hat{y} - (1-y) \log (1-\hat{y}))$

`'categorical_crossentropy'`
`categorical_crossentropy`
`CategoricalCrossentropy()`
	Категориальная кроссэнтропия («logloss»). Целевые данные должны быть представлены в one-hot кодировании.
	$\mathrm{CCE} = \mathrm{mean}(-y \log \hat{y})$

`'sparse_categorical_crossentropy'`
`sparse_categorical_crossentropy`
`SparseCategoricalCrossentropy()`
	Разреженная категориальная кроссэнтропия. Это та же самая категориальная кроссэнтропия, но целевые данные должны быть представлены в label кодировании.

Это наиболее часто используемые функции потерь, но есть и много других. Их можно найти здесь.

Бинарная кроссэнтропия используется с одним выходным нейроном и функцией активации выходного слоя `sigmoid`, категориальная – с несколькими выходными нейронами и функцией активации `softmax`.

В конструкторах `BinaryCrossentropy`, `CategoricalCrossentropy` и `SparseCategoricalCrossentropy` есть параметр `from_logits`, по умолчанию равный `False`. Если установить этот параметр в значение `True`, то функцию активации из выходного слоя следует убрать: она будет расчитана внутри функции потерь.

Такой вариант предпочтительнее, поскольку он несколько стабильнее в плане погрешности вычислений. Суть в том, что softmax использует экспоненту, а кроссэнтропия - логарифм. Если взять экспоненту, а затем логарифм от числа, которое существенно больше/меньше нуля, то можно получить плюс или минус бесконечность из-за погрешности вычислений. Параметр `from_logits=True` и отсутствие `softmax` в выходном слое избавляют от этой проблемы.

Метрики качества

Любую функцию потерь можно использовать в качестве метрики. Однако не любая метрика подойдет в качестве функции потерь: для этого она должна быть дифференцируема.

Типичным примером недифференцируемой метрики является точность (accuracy) – доля верных ответов в задаче классификации. Как обычно, ее можно указать в виде строки (`'acc'` или `'accuracy'`) или объекта `keras.metrics.Accuracy`. Можно указать сразу несколько метрик в виде списка.

```
model.compile(loss=..., optimizer=..., metrics='accuracy')
```

Конечно, Keras предоставляет возможность создавать собственные метрики в виде функции или подкласса класса `Metric`. Такая возможность требуется достаточно часто, поскольку во многих задачах выходные данные модели имеют сложный формат, и разумный выбор метрик качества является одной из главных составляющих успеха с точки зрения бизнеса.

Если вы собираетесь использовать метрики вручную, то нужно помнить одну важную деталь. Метрики в Keras работают неочевидным образом. В отличие от loss'ов, метрики обладают накопительным действием, то есть возвращают среднее значение по всем измерениям:

```
from keras.metrics import Accuracy
acc = Accuracy()
print(acc([0], [1]).numpy()) #0.0
print(acc([1], [1]).numpy()) #0.5
print(acc([1], [1]).numpy()) #0.6666667
```

Оптимизаторы

Оптимизатор также является объектом. Самым распространенным оптимизатором является **Adam** (**Ada**ptive **m**oment estimation).

```
model.compile(loss=..., optimizer='adam')
model.compile(loss=..., optimizer=keras.optimizers.Adam(1e-3))
```

Второй способ (указание оптимизатора в виде объекта, а не строки) является более гибким, так как он позволяет указывать параметры оптимизатора, в первую очередь learning rate. В качестве learning rate можно указать не только число, но и объект, описывающий стратегию его изменения со временем:

```
from tensorflow.keras.optimizers.schedules import *
lr = ExponentialDecay(initial_learning_rate=1e-
3, decay_steps=1000, decay_rate=0.9)
optimizer = Adam(lr)
```

`ExponentialDecay` описывает экспоненциальное затухание learning rate. Здесь следует внимательно подбирать параметры: слишком быстрое затухание может не дать сети успеть обучиться, а слишком медлленное просто не иметь эффекта при небольшом количестве эпох.

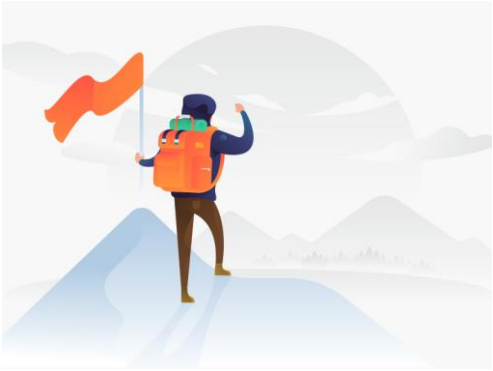
Вместо экспоненциального затухания можно использовать полиномиальное, ступенчатое или собственную стратегию управления learning rate. Подробнее можно почитать в документации.

Есть еще один способ управление learning rate: с помощью коллбэков. Learning rate будет понижаться в те моменты, когда метрика качества на валидации перестает расти. Мы рассмотрим этот метод в следующих разделах.

Что происходит при компиляции?

При компиляции модели задаются функция потерь, оптимизатор и метрики – больше никаких действий не происходит. Перекомпиляция модели заново требуется в случае, если вы изменили набор обучаемых весов (параметр `.trainable` у слоев модели). В этом случае состояние оптимизатора (накопленные моменты в случае градиентного спуска с моментом) будет утеряно.

6. Обучение моделей в Keras



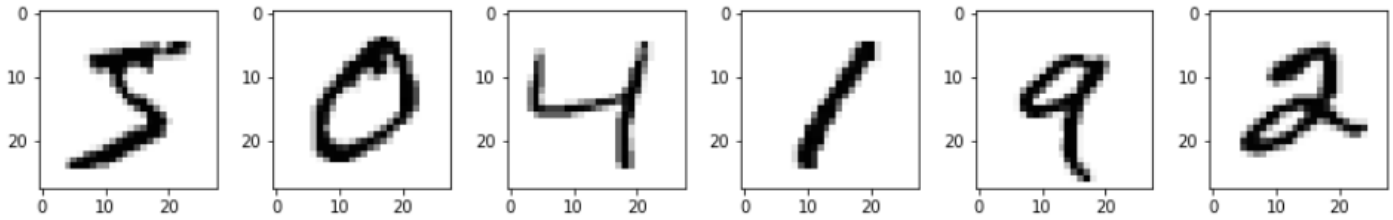
Наконец мы добрались до обучения моделей. Keras следует концепции «от простого к сложному». В простом случае обучение можно выполнить просто вызвав метод `.fit()` на обучающих данных. Если вы хотите встроить в процесс обучения дополнительные действия, то можно использовать коллбэки. Если же вам нужен полный контроль над процессом обучения, то вы можете написать кастомный цикл обучения.

Данные для обучения

В этом разделе для обучения мы будем использовать «Hello, world!» датасет из мира глубокого обучения – MNIST. Обычно задачи компьютерного зрения решаются сверточными сетями, но пока что мы будем использовать полносвязные: с ними будет проще работать.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Model, Sequential, layers, losses, optimizers

(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
X_train = X_train.reshape(-1, 28*28) / 255
X_test = X_test.reshape(-1, 28*28) / 255

print(X_train.shape, X_train.dtype) #(60000, 784) float64
print(X_test.shape, X_test.dtype) #(10000, 784) float64

print(y_train.shape, y_train[:10]) #(60000,) [5 0 4 1 9 2 1 3 1 4]
print(y_test.shape, y_test[:10]) #(10000,) [7 2 1 0 4 1 4 9 5 9]

import matplotlib.pyplot as plt
fig, axes = plt.subplots(1, 6, figsize=(15, 2))
for img, ax in zip(X_train, axes.flat):
  ax.imshow(img.reshape(28, 28), cmap='Greys')
plt.show()
```

Исходные данные – по 784 пикселя для каждого из 60000 и 10000 тестовых изображений, значения пикселей от 0 (фон) до 1 (цифра). Целевые данные – изображенные цифры от 0 до 9 в label-кодировании.



Часто при обучении используются две отложенные выборки: на валидационной настраиваются гиперпараметры, на тестовой проводится финальное тестирование. Сейчас мы для упрощения используем только одну обучающую и одну отложенную выборку.

Давайте обучим модель!

Наличие GPU многократно ускоряет обучение моделей. Проверьте, что Tensorflow подключен к GPU, выполнив команду:

```
tf.config.list_physical_devices('GPU')
```

Если у вас есть видеокарта Nvidia, но TensorFlow ее не видит, обновите TensorFlow с помощью conda или pip и следуйте инструкциям из этого гайда. Либо вы можете использовать облачные вычисления с помощью Google Colab, Kaggle Kernel, Gradient Community Notebooks и других платформ.

```
model = Sequential([
  layers.InputLayer(28*28),
  layers.Dense(500, 'relu'),
  layers.Dense(500, 'relu'),
  layers.Dense(10)
])
```

Наша модель не имеет softmax в выходном слое, в этом случае выходные значения называют **logits**. После применения softmax logits превращаются в **вероятности (probabilities)**.

```
model.compile(
    loss=losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=optimizers.Adam(),
    metrics='accuracy'
)

model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=10,
    batch_size=1024
)
```

```
Epoch 1/10
59/59 [==============================] - 1s 10ms/step - loss: 0.4663 -
accuracy: 0.8730 - val_loss: 0.1978 - val_accuracy: 0.9420
......
Epoch 10/10
59/59 [==============================] - 0s 6ms/step - loss: 0.0143 - accuracy:
0.9969 - val_loss: 0.0662 - val_accuracy: 0.9809
```

Как видим, каждая эпоха занимает 59 шагов – это размер обучающего датасета (60000) деленый на размер батча (1024). После каждой эпохи выдается значение функции потерь и метрики accuracy на обучающих и валидационных данных.

Мы достигли точности на валидации 98%, то есть модель ошибается лишь в одном из 50 случаев. Современные нейронные сети на датасете MNIST достигают точности 99.85%, то есть неправильно распознают лишь 1-2 изображения из тысячи.

Fit, predict и evaluate

Рассмотрим три основных метода для работы с моделями:

Метод **`.fit()`** обучает модель на данных. Если указан параметр validation_data или validation_split, то после каждой эпохи осуществляется валидация.

Метод **`.predict()`** возвращает предсказания модели на данных. В качестве альтернативы может вызывать модель как функцию, но тогда данные придется вручную делить на батчи.

Метод **`.evaluate()`** оценивает качество модели (loss и метрики) на данных.

Все три метода могут принимать в качестве данных либо массивы, либо итерируемые объекты, которые генерируют батчи обучающих данных.

> В python итерируемым объектом называется объект, который может вернуть некую последовательность элементов. У итерируемых объектов есть метод `obj.__iter__()`, который также может быть вызыван функцией `iter(obj)`. Этот метод возвращает итератор. Итератор хранит указатель на текущий элемент и имеет метод `iter.__next__()`, который также может быть вызыван функцией `next(iter)`. Этот метод как раз и возвращает следующий элемент.
>
> Мы не будем сейчас подробно рассматривать генераторы в Keras (такие как `ImageDataGenerator`, `Sequence` или `Dataset`), поскольку это очень обширная тема. Как правило подобные инструменты изучаются по мере необходимости, читая документацию и примеры со Stackoverflow. Позже мы будем решать на Keras задачу компьютерного зрения, и там мы будем работать с генераторами.

На странице документации вы можете найти подробное описание всех параметров методов `.fit()`, `.predict()` и `.evaluate()`. Одним из важных параметров в методе `.fit()` является параметр `callbacks`, который мы сейчас разберем.

Callbacks

Параметр **`callbacks`** в методе `.fit()` означает задание неких действий, которые будут выполнены после каждой эпохи или шага обучения. Давайте рассмотрим какие могут быть коллбэки и зачем они нужны.

> Подробнее о каждом из коллбэков и их параметрах читайте в документации.

```
from tensorflow.keras.callbacks import *
```

```
ModelCheckpoint('best_model.h5', monitor='val_accuracy',
                verbose=1, mode='max', save_best_only=True)
```

Этот коллбэк сохраняет модель в файл `best_model.h5` (в текущей рабочей директории) в том случае, если значение `val_accuracy` после текущей эпохи достигло максимума. Например, после первых 5 эпох точность росла, а в течение следующих 5 эпох падала. Загрузив модель из файла `best_model.h5` вы вернете состояние модели после первых 5 эпох, когда точность стала максимальной.

```
EarlyStopping(monitor='val_loss', patience=5)
```

Прекращает обучение модели, если `val_loss` не падает в течение 5 эпох. Это может говорить о переобучении либо о стагнации процесса обучения.

Совет: не торопитесь прекращать обучение, если метрика качества перестала расти. Приведем следующую цитату (источник):

Leave it training. I've often seen people tempted to stop the model training when the validation loss seems to be leveling off. In my experience networks keep training for unintuitively long time. One time I accidentally left a model training during the winter break and when I got back in January it was SOTA ("state of the art").

```
TerminateOnNaN()
```

Прекращает обучение, если loss стал равным NaN. Это может происходить по многим разным причинам. Среди самых очевидных причин:

- Слишком высокий learning rate
- Отсутствие нормализации входных данных
- NaN во входных данных
- Ошибки в архитектуре сети

```
LambdaCallback(on_epoch_end=lambda epoch, logs:
               print(f'Epoch {epoch} ended... Should we do something?'))
```

Позволяет выполнять любые действия после конца эпохи или шага обучения (батча), а также и в другие моменты.

Если требуется выполнять какие-то действия после каждой эпохи (например, визуализация процесса обучения на графиках, получение предсказаний или изменение обучающих данных), то можно сделать цикл, в котором будет вызываться метод `.fit()` с параметром `epochs=1`, а затем выполняться остальные необходимые действия. Это удобнее, чем использовать `LambdaCallback`. Мы рассмотрим такой пример в следующем разделе.

`ReduceLROnPlateau()`

Уменьшает learning rate в те моменты, когда начинает наблюдаться стагнация процесса обучения, то есть выбранная метрика (по умолчанию `val_loss`) не улучшается в течение нескольких эпох. Этот коллбэк является хорошей альтернативой плавному уменьшению learning rate от шага к шагу обучения (например `ExponentialDecay`, см. раздел «Компиляция моделей в Keras»).

Задания

В качестве заданий вам предлагается несколько экспериментов с обучением моделей.

Для выполнения каждого из заданий **используйте код подготовки данных, создания и компиляции модели из данного раздела**.

Используйте GPU для выполнения заданий для ускорения обучения.

Чтобы ваши ответы сошлись с верными, придерживайтесь следующих правил:

Не меняйте архитектуру модели и другие параметры, если этого не требуется в задании, иначе ответ может не сойтись. Если в задании требуется изменить какой-то параметр, то на другие задания это не распространяется.

В каждом задании под обученной моделью подразумевается состояние модели после последней эпохи. Не делайте возвращение к эпохе, на которой достигнута наилучшая точность на валидационном датасете, если это не требуется в задании.

Подсказка: используйте параметр `verbose=0` в методах `.fit()` и `.evaluate()`, чтобы информация о процессе обучения не выводилась в консоль.

Задание 6.1

Повторите процедуру создания и обучения модели 50 раз. Обучайте в течение 10 эпох. В каждом случае получите предсказания обученной модели на тестовых данных. Сохраните все предсказания в массив, они нам понадобятся далее. Чему равно медианное значение точности (accuracy) на тестовом датасете?

Подсказка: если предсказания даны в one-hot кодировании (logits или probabilities), а ответы в label-кодировании, то посчитать точность можно так (не перепутайте порядок аргументов):

```
np.mean(keras.metrics.sparse_categorical_accuracy(y_true, y_pred))
```

○ от 97.5% до 97.8%

◉ от 97.8% до 98.2%

○ от 98.2% до 98.5%

○ от 98.5% до 98.9%

Решение:

```
preds = []
from tqdm.notebook import tqdm
for i in tqdm(range(50)):
  model = Sequential([
    layers.InputLayer(28*28),
    layers.Dense(500, 'relu'),
    layers.Dense(500, 'relu'),
    layers.Dense(10)
  ])
  model.compile(
      loss=losses.SparseCategoricalCrossentropy(from_logits=True),
      optimizer=optimizers.Adam(),
      metrics='accuracy'
  )
  model.fit(X_train, y_train, epochs=10, batch_size=1024, verbose=0)
  preds.append(model.predict(X_test))
acc_fn = keras.metrics.sparse_categorical_accuracy
accuracies = [np.mean(acc_fn(y_test, pred)) for pred in preds]
np.median(accuracies)
```

Задание 6.2

Воспользуйтесь предсказаниями из предыдущего задания и усредните предсказания (**logits)** по всем моделям. Чему теперь равно значение точности?

○ от 97.6% до 97.9%

○ от 97.9% до 98.1%

○ от 98.1% до 98.3%

◉ выше 98.3%

Решение:

```
# + код предыдущего задания
preds_final = np.array(preds).mean(axis=0)
```

```
print(np.mean(acc_fn(y_test, preds_final)))
```

Задание 6.3

Теперь усредните предсказания в виде вероятностей (**probabilities)** по всем моделям. Чему теперь равно значение точности?

Подсказка: батч logit'ов можно превратить в батч вероятностей функцией `tf.nn.softmax`.

○ от 97.6% до 97.9%

○ от 97.9% до 98.1%

○ от 98.1% до 98.3%

◉ выше 98.3%

Решение:

```
# + код предыдущего задания
preds_final = np.array([tf.nn.softmax(x) for x in preds]).mean(axis=0)
print(np.mean(acc_fn(y_test, preds_final)))
```

Задание 6.4

Разделите тестовый датасет на две равные части: первые 5000 примеров и вторые 5000 примеров (не перемешивая). Создайте модель и обучите ее в течение 100 эпох, используя параметр `learning_rate=1e-2`. После каждой эпохи оцените точность на обоих тестовых датасетах.

Подсказка: сделайте цикл, в котором сначала вызывайте метод `.fit()` *с параметром* `epochs=1`*, а затем дважды метод* `.evaluate()`.

Посчитайте среднюю точность (усреднив все значения точности) на обоих тестовых датасетах. Насколько отличается средняя точность на двух датасетах?

○ слабо (0.5% и менее)

○ средне (0.5%-1.0%)

◉ сильно (1.0% и более)

Решение:

```
from tqdm.notebook import tqdm
X_test1, y_test1 = X_test[:5000], y_test[:5000]
X_test2, y_test2 = X_test[5000:], y_test[5000:]
val_accuracy1_history = []
val_accuracy2_history = []
for epoch_index in tqdm(range(100)):
  model.fit(X_train, y_train, epochs=1, verbose=0, batch_size=1024)
  val_loss1, val_accuracy1 = model.evaluate(X_test1, y_test1, verbose=0)
  val_loss2, val_accuracy2 = model.evaluate(X_test2, y_test2, verbose=0)
  val_accuracy1_history.append(val_accuracy1)
  val_accuracy2_history.append(val_accuracy2)
print(np.mean(val_accuracy2) - np.mean(val_accuracy1))
```

Задание 6.5

Посчитайте корреляцию Пирсона между точностью на первом и втором тестовых датасетах. Чему равно значение корреляции?

○ < 0.3

○ 0.3-0.6

◉ 0.6-0.85

○ > 0.85

Решение:

```
# + код предыдущего задания
from scipy.stats import pearsonr
print(pearsonr(val_accuracy1_history, val_accuracy2_history))
```

Из полученных результатов можно сделать вывод о том, что точность модели на некоем тестовом датасете зависит от трех составляющих:

1. Степень генерализации модели (насколько хорошо модель обучилась под задачу).
2. Случайный фактор, зависящий от размера датасета и конкретных примеров в нем.

Чем больше датасет – тем надежнее оценка точности и меньше влияние случайного фактора. Это говорит о том, что корреляция между оценками точности модели на двух больших тестовых датасетах будет выше, чем на двух маленьких. Например, если бы в предыдущих заданиях мы использовали бы два тестовых датасета всего по 500 чисел, то корреляция точности между ними была бы всего лишь в районе 0.2.

Поэтому если в процессе обучения наблюдается стагнация (точность на валидации перестала расти и начала случайно колебаться), то не торопитесь возвращать состояние модели к той эпохе, на которой точность на валидационном датасете была наибольшей. Возможно это был случайный скачок точности только для данного датасета.

7. Визуализация обучения в Keras

Воспользуемся кодом из раздела «Обучение моделей в Keras» для подготовки данных, создания и компиляции модели.

В этом разделе мы сначала рассмотрим визуализацию обучения модели с помощью matplotlib и обсудим несколько тонкостей, связанных с интерпретацией полученных данных. Затем покажем, как можно делать визуализацию с помощью инструмента TensorBoard.

Где хранится история loss и метрик?

Объект `keras.callbacks.History()` хранит историю loss и метрик после каждой эпохи. Если в списке коллбэков, переданных в метод `.fit()`, не было объекта `History`, то этот объект автоматически туда добавляется. Этот же объект возвращается из метода `.fit()`, а также записывается в параметр `model.history`.

Объект `History` имеет поле `history`, которое является словарем:

```
model.fit(X_train, y_train, epochs=2, batch_size=1024,
          validation_data=(X_test, y_test))
%precision %.3f #устанавливаем точность для метода IPython.display.display
display(model.history.history)
```

```
{'accuracy': [0.871, 0.953],
 'loss': [0.471, 0.161],
 'val_accuracy': [0.944, 0.963],
 'val_loss': [0.190, 0.126]}
```

В данном словаре 4 ключа: loss и accuracy при обучении и на валидации.

Рассмотрим один важный момент. История loss и accuracy на обучающих данных **усредняется** по каждой эпохе. Это приводит к тому, что эти показатели оказываются хуже, чем если бы они были рассчитаны на всем обучающем датасете в конце эпохи. Поэтому при визуализации график loss и accuracy на обучающих данных полезно сдвигать влево, от конца к середине эпохи.

> Часто встречается ситуация, когда вам требуется вызывать метод `.fit()` несколько раз. Например, если вы обучаете модель в несколько стадий или по

одной эпохе. Чтобы история не стиралась каждый раз, когда вы вызываете `.fit()` заново, вам нужно передавать в качестве коллбэка уже существующую историю:

```
model.history = keras.callbacks.History()
for epoch in range(10):
  model.fit(..., callbacks=[model.history])
```

В данном примере в `model.history` будет накапливаться история по всем 10 эпохам.

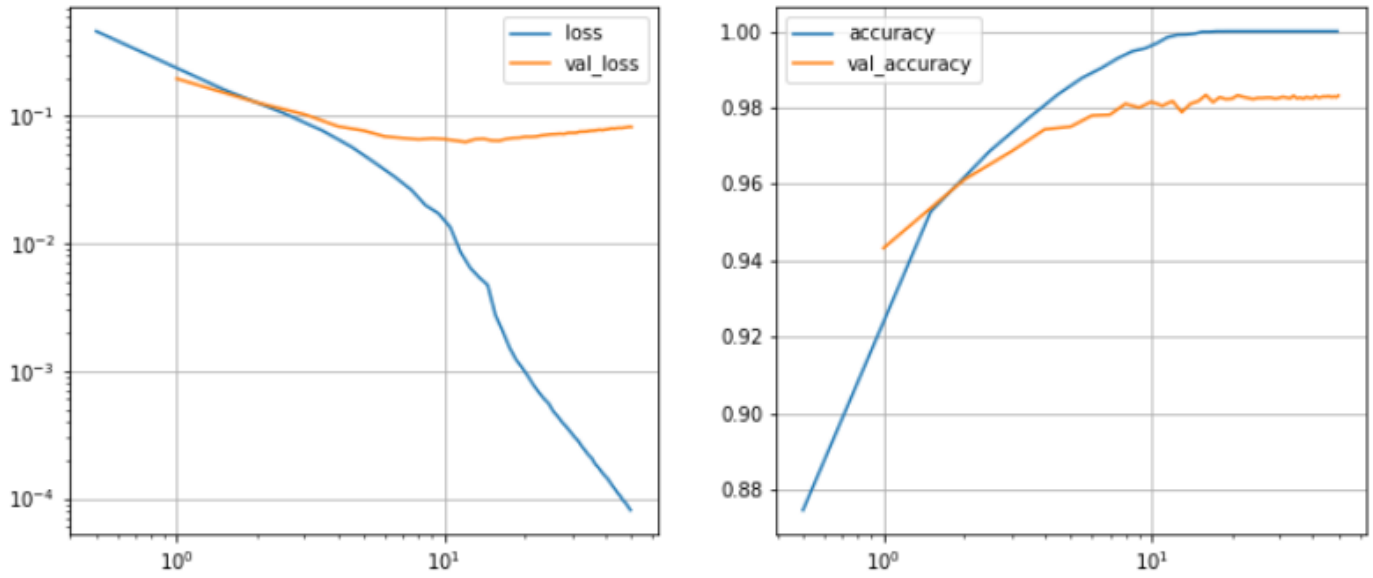
Рисуем историю с помощью matplotlib

Мы будем использовать ручной цикл, обучая модель по одной эпохе, а затем делая визуализацию. Такой подход более гибкий, чем использование `LambdaCallback`, и кроме того в функционировании `LambdaCallback` есть несколько неочевидных моментов, так что по возможности его лучше избегать.

```
from IPython.display import clear_output

model = Sequential([
  layers.InputLayer(28*28),
  layers.Dense(500, 'relu'),
  layers.Dense(500, 'relu'),
  layers.Dense(10)
])

model.compile(
    loss=losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=optimizers.Adam(),
    metrics='accuracy'
)

def visualize(history):
  clear_output(wait=True)
  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
  epochs = len(history.history['loss'])
  # на первом графике рисуем loss
  ax1.plot(np.arange(epochs) + 0.5, history.history['loss'], label='loss')
  ax1.plot(np.arange(epochs) + 1, history.history['val_loss'], label='val_loss')
  ax1.set_xscale('log')
  ax1.set_yscale('log')
  ax1.legend()
  ax1.grid()
  # на втором графике рисуем accuracy
  ax2.plot(np.arange(epochs) + 0.5, history.history['accuracy'], label='accuracy')
  ax2.plot(np.arange(epochs) + 1, history.history['val_accuracy'], label='val_accuracy
')
  ax2.set_xscale('log')
  ax2.legend()
```

```
  ax2.grid()
  plt.show()

model.history = keras.callbacks.History()
for epoch in range(50):
  model.fit(X_train, y_train, epochs=1, batch_size=1024,
            validation_data=(X_test, y_test), callbacks=[model.history])
  visualize(model.history)
```

В результате после каждой эпохи график обновляется. В конце обучения он выглядит так:



> Данный раздел предполагает базовое знание библиотеки matplotlib, поэтому мы не разбираем подробно все используемые методы из этой библиотеки.

Loss и accuracy на обучающих данных расчитывается усредненно по всей эпохе, и поэтому графики сдвинуты на пол-эпохи влево относительно графиков на валидации. Впрочем, этого можно и не делать для упрощения.

Для оси времени выбран логарифмический масштаб. Обучение нейронных сетей как правило замедляется со временем, и в конце обучения точность может расти настолько медленно, что это будет видно только в логарифмическом масштабе.

> Если рисовать только один график (без `plt.subplots`), то установить масштаб оси можно такой командой: `plt.gca().set_xscale('log')`. Полезно знать и другие возможные масштабы, такие как `'symlog'` и `'logit'`.

Проверьте себя, выполнив задание ниже. После этого мы разберем подробнее утверждения из этого задания.

Задание 7.1

Какие выводы можно сделать по графикам, приведенным выше?

☑ По графику loss видно, что модель стала переобучаться

☑ Модель выдает верный ответ на всех обучающих примерах.

☐ Имеет смысл продолжать обучение

☑ Если продолжать обучение, то loss на обучающих данных будет убывать.

☐ Loss на обучающих данных будет убывать с экспоненциальной скоростью.

Делаем выводы из графика (разбор задания 7.1)

Модель выдает верный ответ на всех обучающих примерах (точность равна единице).

Loss на обучающих примерах будет и далее падать, но не экспоненциально. На графике **обе** оси отрисованы в логарифмическом масштабе, а график выглядит как прямая линия, то есть при увеличении X в 10 раз Y уменьшится в 10*C раз, где C – константа, зависящая от наклона прямой. Это означает полиномиальную скорость падения.

Loss на валидации начал расти, то есть процесс генерализации модели остановился и пошел вспять, и теперь модель только «подгоняется» к конкретным обучающим примерам. График точности на валидации не растет, поэтому продолжать обучение не имеет смысла.

> Здесь важно заметить, что такой вывод не всегда верен. Бывают случаи, когда метрика на валидации внезапно начинает расти после долгого периода стагнации, например такое может быть при наличии сильного dropout'а внутри сети. Но в нашем случае модель очень простая, и здесь можно сделать однозначный вывод, что продолжать обучение не стоит.

Задание 7.2

Что из перечисленного может быть причиной того, что график точности на валидации опережает график точности на обучающих данных?

☑ В модель встроена регуляризация (например dropout), которая ухудшает точность при обучении, но не имеет эффекта при валидации.

☑ Мы обучаем модель с аугментациями изображений, а при валидации не делаем аугментации.

☑ При валидации мы делаем test-time аугментации (TTA), то есть усредняем предсказания на одном и том же изображении с разными аугментациями для повышения точности предсказаний.

☑ Мы сравниваем точность на валидации в конце эпохи и точность на обучающих данных, усредненную по всей эпохе.

☑ В валидационный датасет по случайности попали более легкие примеры, чем в обучающий датасет.

Визуализация с помощью TensorBoard

TensorBoard – это инструмент для анализа ML-моделей с интерактивным интерфейсом. TensorBoard доступен либо в виде веб-интерфейса (через браузер), либо в виде IPython-виджета (в python-ноутбуках).

TensorBoard работает независимо от TensorFlow, «каналом связи» между ними является папка с логами. TensorFlow пишет в нее логи, TensorBoard их считывает. Такой принцип работы позволяет использовать TensorBoard не только с TensorFlow, но и с PyTorch и другими инструментами. TensorBoard входит в состав python-пакта `tensorflow`, но может быть установлен отдельно:

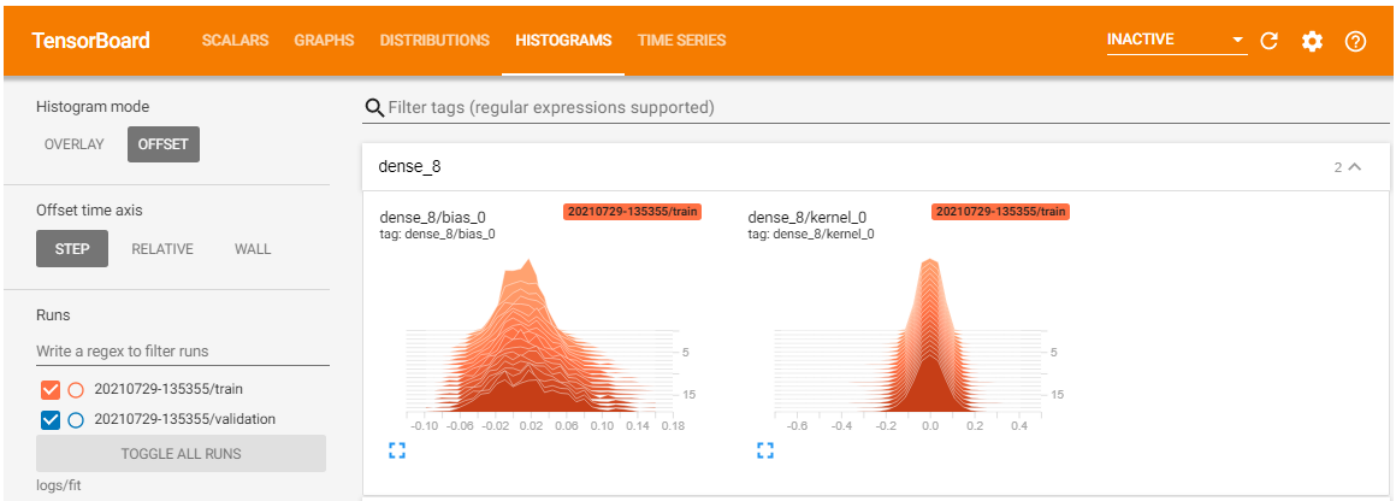
```
pip install tensorboard
```

Познакомиться с TensorBoard можно запустив ноутбук из официального гайда. В нем TensorBoard запускается как IPython-виджет.

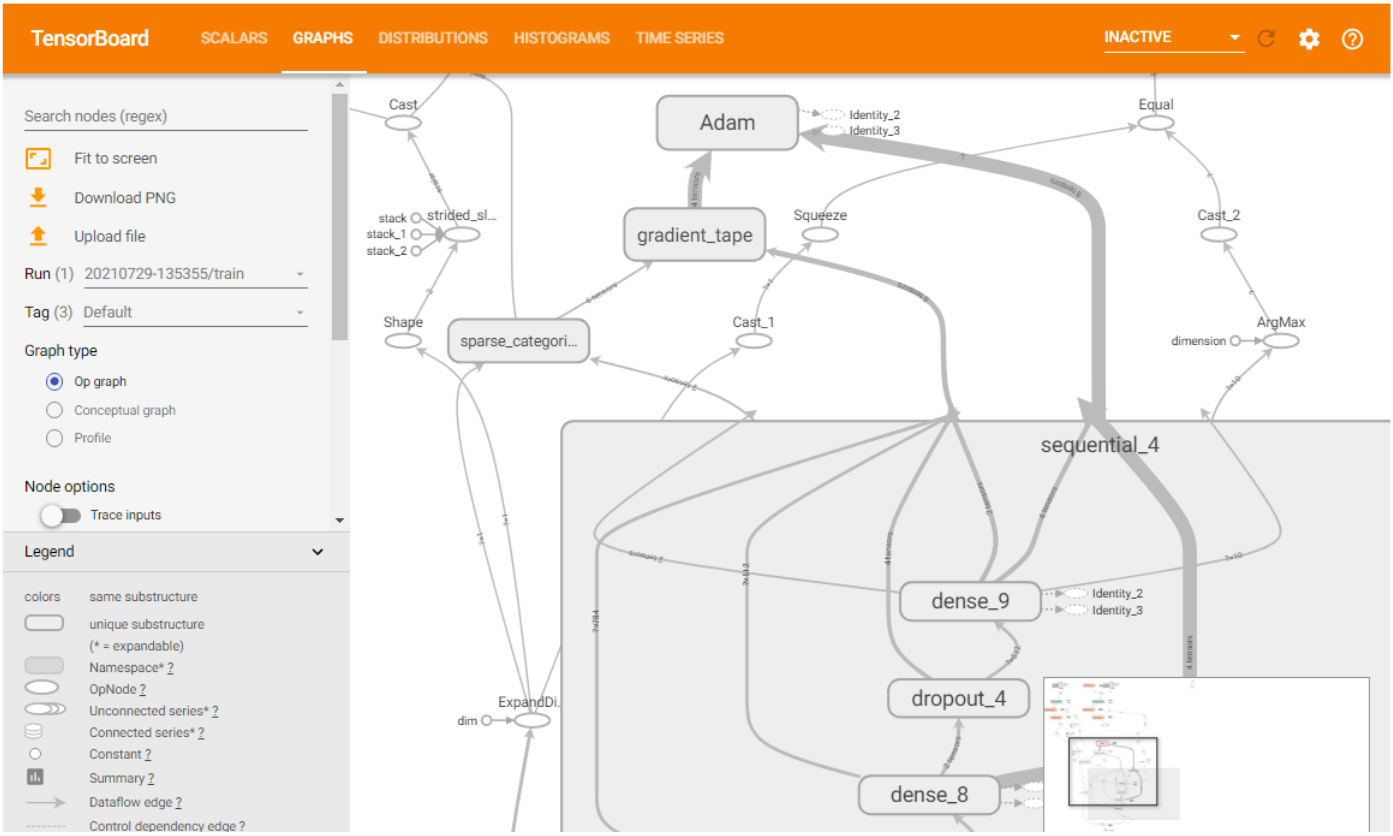
Есть несколько типов данных, которые можно визуализировать в TensorBoard:

Скаляры. Различные метрики, которые как правило собираются в конце каждой эпохи. Например, функция потерь и точность на обучающих и тестовых данных.

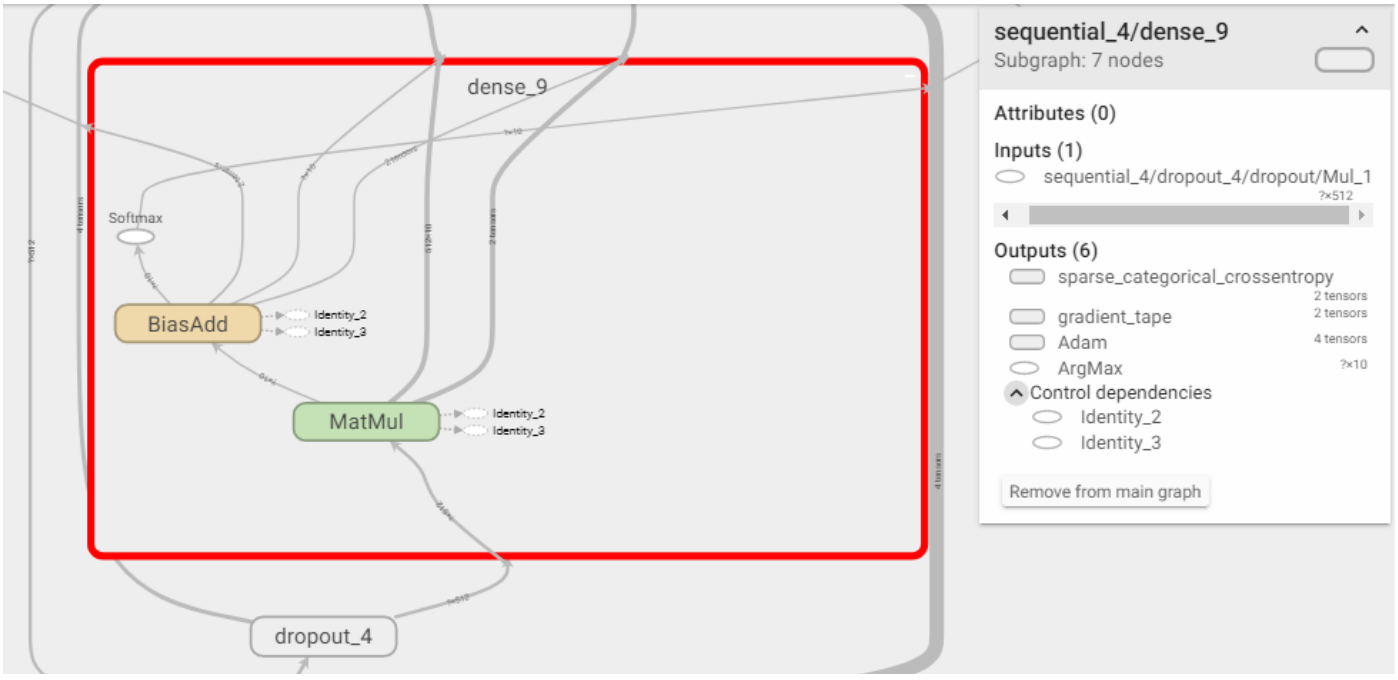
Гистограммы. На скриншоте ниже «dense_1/bias_0» означает вектор bias'ов первого полносвязного слоя модели, 20 гистограмм соответствуют 20 эпохам обучения. Мы видим как менялась гистограмма распределения bias'ов первого полносвязного слоя в процессе обучения. Рядом показаны 20 гистограмм для матрицы весов первого слоя («dense_1/kernel_0»).



Графы вычислений. Можно посмотреть, как выглядит целиком весь граф вычислений, включая функцию потерь и оптимизатор (в TensorFlow они также являются частями графа).



Элементы графа можно «раскрывать» двойным щелчком, если они в свою очередь состоят из вложенных элементов. Например, так устроен полносвязный слой:



В python-ноутбуках визуализация процесса обучения с помощью matplotlib во многих случаях предпочтительнее, поскольку является более гибкой. TensorBoard удобно использовать в тех случаях, когда отсутствует интерактивная сессия, то есть при запуске python-скриптов, с которыми можно взаимодействовать только через командную строку.

8. Кастомизация моделей и обучения в Keras

Давайте вспомним материал предыдущих разделов. В Keras мы строили модель в виде статического графа вычислений, то есть все узлы были заранее соединены друг с другом. Мы выбирали функцию потерь, оптимизатор и метрики качества, после чего вызывали метод `.fit()` для обучения модели. Однако во многих случаях нам может потребоваться более продвинутая функциональность.

В компьютерном зрении стараются использовать самые новые модели и подходы, которые показывают наилучшие результаты. Для этого требуется читать научные статьи и либо искать их имплементации в виде программного кода, либо самому программировать описанные в них схемы вычислений. И часто эти схемы довольно нетривиальны. В частности, специалисту по компьютерному зрению полезно уметь следующее:

- «Навешивать» на модель дополнительные loss'ы, то есть минимизировать одновременно несколько функций потерь (или их сумму, что то же самое).
- Обучать параллельно на нескольких датасетах и/или несколько моделей. Типичным примером являются GAN'ы (генеративно-состязательные сети), где две сети обучаются параллельно.
- Следить за тем, что происходит внутри сети в процессе обучения.

Построение нестандартных слоев и моделей

О построении нестандартных слоев и моделей с помощью создания подклассов `Layer`, `Model` или `Sequential` можно прочитать в официальном гайде. Поэтому сейчас мы не будем подробно останавливаться на этой теме, лишь упомянем одну важную деталь и приведем пример.

Если мы создаем модель или слой как подкласс, то мы должны определить метод `.call()`, который преобразует входные данные в выходные. Внутри метода `.call()` мы используем последовательность операций TensorFlow и возвращаем результат.

По умолчанию в Keras строится статический граф вычислений, это означает, что **метод `.call()` будет вызван лишь один раз** для построения статического графа, то есть соединения операций TensorFlow друг с другом как звеньев.

Если в методе `.call()` используются управляющие конструкции языка, такие как `if`, `for`, `while`, `continue`, `break`, то они также будут преобразованы в звенья графа, такие как `tf.cond` и `tf.while_loop`. Подробнее о том, как это происходит, можно прочитать здесь.

В некоторых случаях это может быть нежелательно, например если вы хотите в методе `.call()` использовать внешние переменные или функции из сторонних библиотек. Тогда в метод `.compile()` нужно передать параметр

run_eagerly=True, и модель будет выполняться как динамический граф вычислений, то есть метод .call() будет вызываться каждый раз при запуске модели. Это, однако, может существенно повлиять на производительность.

В качестве примера давайте создадим полносвязную сеть с двумя скрытыми слоями, которые выполняются каждый раз в случайном порядке:

```
class CustomModel(Model):
  def __init__(self):
    super(CustomModel, self).__init__()
    self.hidden_layers = [layers.Dense(784, 'relu') for _ in range(2)]
    self.head = layers.Dense(10, 'softmax')
  def call(self, input, training=None, order=None):
    output = input
    order = order or np.random.choice(range(2), 2, replace=False)
    for layer_idx in order:
      output = self.hidden_layers[layer_idx](output)
    return self.head(output)
```

Скомпилируем эту модель с параметром run_eagerly=True и обучим. Будем использовать для обучения подготовленные данные из раздела «Обучение моделей в Keras».

```
model = CustomModel()
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam', metrics='accuracy',
              run_eagerly=True)
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=20, batch_size=1024)
```

Теперь дважды получим предсказания: сначала запустим скрытые слои в одном порядке, затем в обратном порядке. В каждом случае расчитаем точность на валидации.

```
from keras.metrics import sparse_categorical_accuracy
preds1 = model(X_test, order=[0, 1]).numpy()
preds2 = model(X_test, order=[1, 0]).numpy()
assert not np.array_equal(preds1, preds2)
print(np.mean(sparse_categorical_accuracy(y_test, preds1))) #0.9832
print(np.mean(sparse_categorical_accuracy(y_test, preds2))) #0.9817
print(np.mean(sparse_categorical_accuracy(y_test, preds1 + preds2))) #0.984
```

Этот эксперимент демонстрирует высокую способность нейронных сетей к обучению. Даже если скрытые слои выполняются каждый раз в произвольном порядке, модель все равно смогла обучиться и выдает высокую точность. Более того, из данной модели можно сделать ансамбль, запустив скрытые слои во всех возможных порядках и усреднив предсказания.

Динамический граф вычислений проще программировать, но в большинстве случаев можно построить и статический граф. Например, для придуманной нами сети, в которой скрытые слои запускаются в случайном порядке, статический граф строится таким образом:

```
  def call_hidden_layers_in_order(self, input, order):
    output = input
    for layer_idx in order:
      output = self.hidden_layers[layer_idx](output)
    return output
  def call(self, input, training=None, order=None):
    if order:
      output = self.call_hidden_layers_in_order(input, order)
    else:
      all_possible_orders = list(permutations(range(self.hidden_layers_count
)))
      all_outputs_dict = {order: self.call_hidden_layers_in_order(input, ord
er)
                          for order in all_possible_orders}
      all_outputs = tf.stack(list(all_outputs_dict.values()))
      index = tf.random.uniform((), minval=0, maxval=len(all_outputs),
                                dtype=tf.int32)
      output = all_outputs[index]
    return self.head(output)
```

Посмотрев на код кажется, что мы вызываем слои во всех возможных порядках, и затем случайно выбираем один из получившихся результатов, что неэффективно с вычислительной точки зрения. Но если мы не передаем в `.compile()` параметр `run_eagerly=True`, то метод `.call()` вызывается только один раз и строит статический граф вычислений. Поэтому, наоборот, такой метод построения существенно ускорит обучение и инференс сети.

Кастомизация цикла обучения

Ранее для обучения мы всегда использовали метод `.fit()`, который делал за нас всю необходимую работу. Для большего контроля над тем, что происходит при обучении, мы можем написать цикл обучения вручную. Для этого нам нужно будет запускать модель, считать функцию потерь, брать откуда-то градиенты и передавать их в оптимизатор.

О том, как писать кастомный цикл обучения, написано в двух официальных гайдах (1, 2). Давайте рассмотрим пример. Будем использовать для обучения подготовленные данные из раздела «Обучение моделей в Keras».

```
class CustomSequential(Sequential):
  def train_step(self, data):
    x, y = data
```

```
    with tf.GradientTape() as tape:
        y_pred = self(x, training=True)
        loss = self.compiled_loss(y, y_pred, regularization_losses=self.losses)
    trainable_vars = self.trainable_variables
    gradients = tape.gradient(loss, trainable_vars)
    self.optimizer.apply_gradients(zip(gradients, trainable_vars))
    self.compiled_metrics.update_state(y, y_pred)
    return {m.name: m.result() for m in self.metrics}

model = CustomSequential([
  layers.InputLayer(28*28),
  layers.Dense(784, 'relu'),
  layers.Dense(784, 'relu'),
  layers.Dense(10, 'softmax')
])

model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam', metrics='accuracy')
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=20, batch_size=1024)
```

В данном коде мы переопределяем шаг обучения для модели. Те действия, которые ранее делались автоматически, теперь пишем вручную. Мы получаем на вход батч данных и выполняем следующие шаги:

1. Получаем предсказания модели в режиме обучения и рассчитываем loss. Эти операции мы делаем в контексте объекта `tf.GradientTape()`, чтобы затем получить градиенты.
2. У каждой модели есть параметр `.trainable_variables`, в котором хранится список тензоров, являющихся обучаемыми весами модели. Мы обращаемся к этому параметру.
3. Получаем значения градиентов для тензоров `trainable_vars` из объекта `tf.GradientTape()`.
4. Передаем веса и градиенты оптимизатору, который обновляет веса используя градиенты.
5. В последних двух строках метода `train_step` мы работаем с метриками.

Заключение

В данном разделе мы рассмотрели примеры написания кастомного метода call для модели и кастомного цикла обучения. Когда мы будем изучать PyTorch, мы увидим, что в нем такой способ программирования является стандартом. Впрочем, есть и высокоуровневый фреймворк PyTorch Lightning, который позволяет писать более лаконичный код.

9. Работа с данными в Keras

В предыдущих разделах мы рассмотрели процесс создания и обучения моделей, однако не менее важным и непростым является процесс подготовки данных.

При работе с датасетами возникают следующие технические сложности:

1. Датасет часто не помещается в память целиком, поэтому его следует постепенно подгружать с диска в процессе обучения модели.
2. Важно обеспечить высокую производительность процесса аугментаций и возможность распараллеливания.
3. Аренда GPU стоит денег, поэтому важно следить за загруженностью видеокарт, чтобы они не простаивали без дела. Для этого используются инструменты профилирования.

Как правило в метод `.fit()` передаются не массивы данных (как мы делали ранее), а генераторы. В данном разделе мы сначала рассмотрим наиболее простые варианты генераторов, а затем изучим `ImageDataGenerator`. В следующем разделе мы рассмотрим наиболее продвинутый и производительный инструмент `tf.data.Dataset`.

Для полноты картины упомянем, что при работе с датасетами также возникают следующие сложности общего характера:

1. Часто данные приходится собирать, очищать и размечать самостоятельно или с применением «рабочей силы», такой как Amazon Mechanical Turk.
2. Датасет может быть доступен только по запросу с заполнением лицензионного соглашения. При этом ответа может не последовать, либо сайт для отправки запроса может не работать.
3. Из-за авторских прав набор данных может содержать не сами изображения и видео, а гиперссылки на них. Их приходится скачивать самостоятельно, причем некоторые из них могли быть удалены с момента создания датасета.
4. Датасет может иметь неполную документацию, запутанную структуру папок, необычные форматы файлов.

Итераторы и keras.utils.Sequence

В метод `.fit()` можно передавать любой iterable-объект, который возвращает пары из батча исходных данных и батча ответов.

Давайте повторим что такое итераторы и генераторы в Python.

Итератор – это объект, имеющий метод `.__next__()` для получения следующего элемента последовательности. Если элементы закончились, метод выбрасывает исключение `StopIteration`.

Генератор – это функция, содержащая инструкцию `yield`. При выполнении такая функция возвращает итератор. Впрочем, на практике под «генератором» иногда понимают любой итератор, используя эти слова как синонимы.

Итерируемый объект (iterable-объект) – это объект, содержимое которого можно перебрать. Такой объект должен иметь метод `.__iter__()`, возвращающий итератор, или метод `.__getitem__()`, возвращающий элемент с указанным индексом. Во втором случае объект называется последовательностью. Итератор, возвращаемый методом `.__iter__()`, играет роль указателя на текущий элемент. Для итерируемого объекта можно одновременно создать несколько итераторов, они будут работать независимо.

Итераторы также являются итерируемыми объектами: их метод `.__iter__()` возвращает сам объект (`self`).

Приведем минималистичный пример, демонстрирующий как использовать итератор для обучения. Будем обучать модель на массиве из нулей.

```
from tensorflow.keras import Sequential, layers
import numpy as np

def zeros_generator():
  while True:
    yield np.zeros((64, 100)), np.zeros((64, 10))

model = Sequential([layers.Dense(10)])
model.compile(loss='mse', optimizer='adam')
model.fit(zeros_generator(), steps_per_epoch=10)
```

При обучении моделей рекомендуется использовать не простые iterable-объекты, а последовательности, которые могут вернуть элемент по его индексу (метод `.__getitem__()`). В этом случае обучение легче распараллеливается. Последовательность можно создавать как подкласс `keras.utils.Sequence`. Этот класс «заранее» не имеет никакой функциональности, всю функциональность вы пишете сами.

В документации приведен пример использования `Sequence`. Сейчас мы не будем подробно останавливаться на этом классе. Далее мы познакомимся более удобными инструментами, где меньше кода нужно будет писать вручную.

ImageDataGenerator

Данный класс предназначен для загрузки изображений (с диска или из numpy-массива), аугментаций и объединения в батчи. Покажем его работу на примере.

Загрузим датасет beans, содержащий 3 класса изображений листьев бобов. Один класс соответствует здоровым листьям (healthy), два класса – больным листьям (angular_leaf_spot и bean_rust).

```
!mkdir beans_dataset
%cd beans_dataset
!wget -q https://storage.googleapis.com/ibeans/train.zip
!wget -q https://storage.googleapis.com/ibeans/validation.zip
!wget -q https://storage.googleapis.com/ibeans/test.zip
!unzip -q train.zip && rm train.zip
!unzip -q validation.zip && rm validation.zip
!unzip -q test.zip && rm test.zip
!tree -d #требует !apt-get install tree
```

Структура папок:

```
.
├── test
│   ├── angular_leaf_spot
│   ├── bean_rust
│   └── healthy
├── train
│   ├── angular_leaf_spot
│   ├── bean_rust
│   └── healthy
└── validation
    ├── angular_leaf_spot
    ├── bean_rust
    └── healthy
```

В каждой из девяти папок хранятся изображения. С помощью кода ниже мы создаем объект train_generator, который можно использовать в методе `.fit()`.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
    #rescale=1/255,
    horizontal_flip=True,
    rotation_range=10,
    shear_range=0.2,
    brightness_range=(0.8, 1.2),
)

train_generator = train_datagen.flow_from_directory(
    'train',
    target_size=(128, 128),
```

```
    batch_size=32,
    class_mode='categorical',
    shuffle=True,
)
```

Создавая объект `ImageDataGenerator`, мы указываем необходимые преобразования, которые следует провести с каждым изображением. В первую очередь это аугментации. Затем, вызывая метод `flow_from_directory` (или методы `flow`, `flow_from_dataframe` – см. документацию), мы указываем откуда брать изображения. В результате мы получаем итератор.

После выполнения кода выше ни одно изображение не было считано с диска. Изображения считываются «по требованию», то есть когда мы начнем перебирать итератор.

Давайте изучим свойства `train_generator`.

Всего итератор может вернуть 1034 изображения, объединенные в 33 батча.

```
print(len(train_generator), train_generator.n) #33 1034
```

Именам классов (то есть именам папок на диске) были присвоены индексы.

```
print(train_generator.class_indices)
#{'angular_leaf_spot': 0, 'bean_rust': 1, 'healthy': 2}
```

Итератор возвращает пару из исходных и целевых данных: 32 изображения размером 128x128 с 3 цветовыми каналами (RGB) и метки классов в one-hot кодировании.

```
X, y = next(train_generator)
print(X.shape, y.shape) #(32, 128, 128, 3) (32, 3)
```

Команда `.reset()` позволяет вернуться к началу и перебирать изображения заново.

```
print(train_generator.batch_index) #1
train_generator.reset()
print(train_generator.batch_index) #0
```

После завершения перебора итератор вернется к началу автоматически, то есть исключение `StopIteration` никогда не будет выброшено. Поэтому в методе `.fit()` нам нужно указать параметр `steps_per_epoch`. Давайте обучим модель. Для упрощения кода мы сейчас не используем валидационный и тестовый датасеты.

```
from tensorflow.keras import Sequential, layers, losses
from tensorflow.keras.applications import EfficientNetB0
```

```
model = Sequential([
  EfficientNetB0(input_shape=(128, 128, 3), include_top=False),
  layers.GlobalMaxPool2D(),
  layers.Dropout(0.5),
  layers.Dense(3)
])
model.compile(
    loss=losses.CategoricalCrossentropy(from_logits=True),
    optimizer='adam',
    metrics='accuracy'
)
model.fit(train_generator, steps_per_epoch=len(train_generator), epochs=20)
```

Вывод:

```
Epoch 1/20
33/33 [==============================] - 18s 293ms/step - loss: 2.1308 -
accuracy: 0.6325
......
Epoch 20/20
33/33 [==============================] - 10s 312ms/step - loss: 0.0265 -
accuracy: 0.9894
```

Если мы выполняем аугментации, то всегда следует проверять визуально, что мы получаем в итоге, чтобы избежать ошибок. Давайте посмотрим на 6 изображений из нашего итератора:

```
import itertools
import matplotlib.pyplot as plt
import numpy as np

def show_first_images(generator, count=6, labels=True, figsize=(20, 5), normali
zed=False):
  fig, axes = plt.subplots(nrows=1, ncols=count, figsize=figsize)
  for batch, ax in zip(generator, axes.flat):
    if labels:
      img_batch, labels_batch = batch
      img, label = img_batch[0], np.argmax(labels_batch[0]) #берем по одному из
ображению из каждого батча
    else:
      img_batch = batch
      img = img_batch[0]
    if not normalized:
      img = img.astype(np.uint8)
    ax.imshow(img)
    # метод imshow принимает одно из двух:
    # - изображение в формате uint8, яркость от 0 до 255
    # - изображение в формате float, яркость от 0 до 1
    if labels:
      ax.set_title(f'Class: {label}')
  plt.show()
```

```
show_first_images(train_generator)
```

Использование albumentations с ImageDataGenerator

Ранее вы уже познакомились с библиотекой albumentations. Встроить ее в ImageDataGenerator можно с помощью указания параметра `preprocessing_function`:

```
!pip install albumentations -q -U

import albumentations as A

def augment(image):
  image = image.astype(np.uint8)
  aug = A.Compose([
      A.Cutout(p=0.5, num_holes=1, max_h_size=150, max_w_size=200),
      A.Cutout(p=0.5, num_holes=1, max_h_size=100, max_w_size=150), #второй раз
      A.Cutout(p=0.5, num_holes=1, max_h_size=80, max_w_size=100), #третий раз
      A.HorizontalFlip(p=0.5),
      A.ShiftScaleRotate(p=0.7),
      A.HueSaturationValue(p=1),
      # Читайте документацию прежде чем копировать и использовать этот код!
      # https://albumentations.ai/docs/api_reference/augmentations/transforms/
      # https://albumentations.ai/docs/examples/example/
      # Данный набор аугментаций - лишь пример
      # Подумайте сами как будет лучше, почитайте статьи в интернете по этой те
me
  ])
  return aug(image=image)['image']

my_datagen = ImageDataGenerator(
    preprocessing_function=augment
)

for _ in range(3):
  my_generator = my_datagen.flow_from_directory(
      'train',
      target_size=(384, 384),
      batch_size=4,
      shuffle=True
  )
  show_first_images(my_generator)
```

Часто бывает полезно смотреть исходный код используемых библиотек, это и предлагается в качестве задания к данному разделу.

Задание 9.1

Изучите исходный код ImageDataGenerator, чтобы ответить на вопрос: с помощью каких операций осуществляется случайный поворот изображения (параметр `rotation_range`)?

Подсказка.

В Python есть встроенные механизмы интроспекции. Например, имея некий метод, можно напечать его исходный код. Для начала поймем какой метод мы хотим исследовать. Сам объект `ImageDataGenerator` не умеет возвращать батчи из изображений. Для начала его нужно преобразовать в генератор, например с помощью вызова метода `.flow_from_directory()`, как это делалось в коде выше. Полученный объект `train_generator` является итератором, то есть имеет метод `.__next__()`, который возвращает батч изображений и меток. Посмотреть исходный код этого метода можно таким образом:

```
import inspect
print(inspect.getsource(train_generator.__next__))
```

Этот код напечатает следующее:

```
def __next__(self, *args, **kwargs):
```

```
    return self.next(*args, **kwargs)
```

Переменная `self` хранит текущий объект. Мы видим, что этот метод в свою очередь вызывает метод `train_generator.next`. Теперь таким же образом напечатаем код этого метода, и так далее. Таким образом мы будем постепенно углубляться во вложенные методы и приближаться к интересующему нас преобразованию.

Есть и другие способы интроспекции. Например, напечатав аттрибут `.__code__`, мы увидим имя файла с исходным кодом:

```
train_generator.__next__.__code__
```

Вывод:

```
<code object __next__ at 0x7fe4c0e0ad20, file "/usr/local/lib/python3.7/dist-packages/keras_preprocessing/image/iterator.py", line 103>
```

Как видим, исходный код хранится в файле `keras_preprocessing/image/iterator.py`. Можно поискать в интернете по имени файла и найти исходный код на Github либо продолжить применять интроспекцию с помощью `getsource`.

○ Поворот изображения выполняется с помощью операции `cv2.warpAffine`

◉ Поворот изображения выполняется с помощью операции `scipy.ndimage.interpolation.affine_transform`

○ Поворот изображения выполняется с помощью операции `tfa.image.rotate`

○ Поворот изображения выполняется с помощью операции `tf.raw_ops.ImageProjectiveTransformV3`

○ По исходному коду на Python этого нельзя понять

Решение:

(продолжение текста из подсказки к условию) Теперь напечатаем исходный код метода `.next()`:

```
print(inspect.getsource(train_generator.next))
```

Идем дальше:

```
print(inspect.getsource(train_generator._get_batches_of_transformed_samples))
```

В коде видим вызов двух методов: `get_random_transform` и `apply_transform`. Нас интересует последний.

```
print(inspect.getsource(train_generator.image_data_generator.apply_transform))
```

Отсюда идет вызов метода `apply_affine_transform`, который вызывается без префикса `self`. Это значит, что данный метод является внешним и не относится к данному объекту. Значит либо он объявляется в файле с исходным кодом, либо импортируется.

Посмотрим путь к файлу с исходным кодом:

```
train_generator.image_data_generator.apply_transform.__code__
```

Здесь проще всего поискать в интернете по запросу «image/image_data_generator.py github» или
«keras apply_affine_transform github», хотя можно и открыть данный файл локально. В итоге мы найдем исходный код этой функции.

Как видим, сначала создается матрица поворота rotation_matrix, затем эта матрица умножается на матрицы других преобразований (shift, shear, zoom). Таким образом мы получаем матрицу аффинного преобразования. Наконец к изображению (переданному в функцию как параметр `x`) эта матрица применяется с помощью метода `ndimage.interpolation.affine_transform` из библиотеки SciPy.

Выводы из этого задания рассмотрим в следующем разделе.

10. Работа с данными в Keras, часть 2

В предыдущем разделе мы рассмотрели построение генераторов изображений с аугментациями с помощью `ImageDataGenerator`. При этом для выполнения аугментаций не используются тензоры и операции TensorFlow. Это означает, что аугментации изображений не являются частью графа вычислений и выполняются на CPU, а не на GPU.

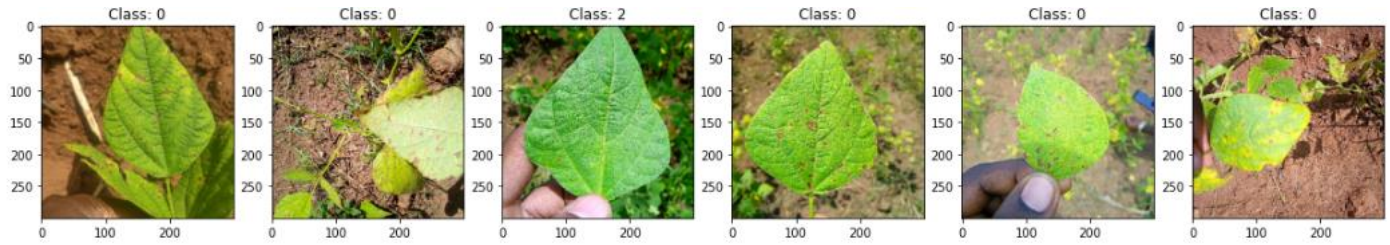
В этом разделе мы увидим, как сделать аугментации частью графа вычислений и выполнять их на GPU с большей производительностью.

Аугментации с помощью слоев Keras preprocessing

В данном методе аугментации (такие как поворот, отражение) являются слоями модели. Конечно многие аугментации могут не быть дифференцируемыми, но это и не требуется. Такие слои встраиваются в начало модели и не обучаются.

Для начала подготовим данные (используем код скачивания датасета beans из предыдущего раздела) и создадим генератор без аугментаций:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_generator = ImageDataGenerator().flow_from_directory(
    'train',
    target_size=(300, 300),
    batch_size=32,
    class_mode='categorical',
    shuffle=True,
)

show_first_images(train_generator)
```



Этот генератор возвращает изображения в формате float, значения пикселей меняются от 0 до 255 (так как не указан параметр `rescale`), то есть изображения не нормализованы.

Аугментации можно выполнять с помощью слоев из модуля `keras.layers.experimental.preprocessing`. Эти аугментации выполняются на GPU, что намного быстрее albumentations, который не использует GPU. Однако набор доступных аугментаций меньше. Для расширения набора доступных аугментаций можно использовать tensorflow_addons.

```
!pip install tensorflow_addons -q
import tensorflow_addons as tfa

from tensorflow.keras import Sequential, layers
from tensorflow.keras.layers.experimental import preprocessing
from tensorflow.keras.applications import EfficientNetB0

preprocessing_model = Sequential([
  preprocessing.RandomCrop(200, 200),
  preprocessing.RandomFlip(mode='horizontal'),
  layers.Lambda(lambda images: tfa.image.random_cutout(images, (100, 100))),
  preprocessing.RandomRotation(0.1),
])

my_model = Sequential([
  preprocessing_model,
  EfficientNetB0(weights='imagenet', input_shape=(200, 200, 3), include_top=Fal
se),
```

```
  layers.GlobalMaxPool2D(),
  layers.Dropout(0.5),
  layers.Dense(10)
])
```

Слои `preprocessing` работают только в режиме обучения, тогда как слой Lambda работает и при обучении, и при инференсе.

Визуализировать изображения с аугментациями можно выполнив следующий код:

```
def augment_batch_to_visualize(X, y):
  return preprocessing_model(X, training=True).numpy(), y

show_first_images((augment_batch_to_visualize(X, y) for X, y in train_generator
))
```

Модуль tf.data

Этот модуль представляет собой наиболее продвинутый пайплайн подготовки данных в TensorFlow.

Основным объектом, с которым мы будем работать, является `tf.data.Dataset`. `Dataset` хранит информацию о конечном или бесконечном наборе элементов и может возвращать эти элементы по очереди. То есть `Dataset` является итерируемым объектом. Элементом как правило являются входные и целевые данные для нейронной сети, либо батч таких данных. Элемент – как правило кортеж (в простом случае) или словарь. Dataset позволяет осуществлять с данными цепочки преобразований, которые осуществляются по мере необходимости (lazy evaluation).

Пока все выглядит достаточно просто, но есть одна тонкость. `Dataset` строится как статический граф вычислений.

\+ С одной стороны это является плюсом: граф компилируется и выполняется на языке C++, эффективно оптимизируется, масштабируется и распараллеливается.

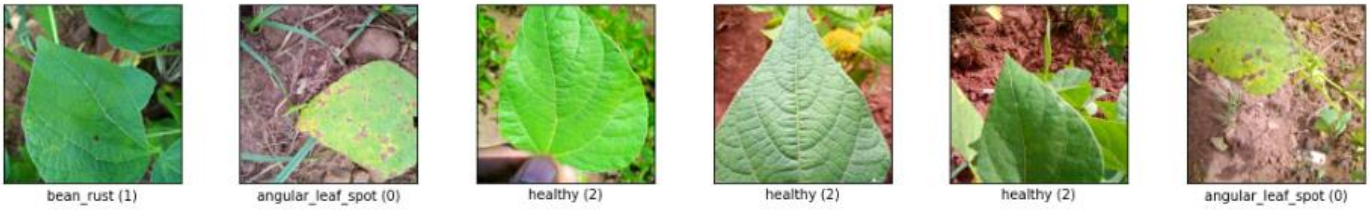
— С другой стороны, в статическом графе все операции должны быть операциями TensorFlow, что усложняет работу. Если требуется применить функцию из другой библиотеки (например из albumentations), то нужно «оборачивать» эту функцию в `tf.py_function`. В целом рекомендуется так не делать, а применять только операции TensorFlow. Например, граф с `tf.py_function` не сможет выполняться на ускорителе TPU.

Резюме: `tf.data.Dataset` является статическим графом вычислений, описывающим цепочку преобразований данных перед их подачей в нейронную сеть. В графе мы применяем только операции TensorFlow.

Если мы хотим создать `Dataset` из изображений, которые мы ранее загрузили в папку `beans_dataset`, то есть два способа.

Простой способ.

```
import tensorflow_datasets as tfds
builder = tfds.ImageFolder('.')
dataset = builder.as_dataset(split='train', shuffle_files=True)
tfds.show_examples(dataset, builder.info, rows=1, cols=6);
```



Здесь мы используем объект ImageFolder. Посмотрим в каком формате датасет возвращает элементы:

```
element_dict = next(iter(dataset))
{x: (y.shape, y.dtype) for x, y in element_dict.items()}
```

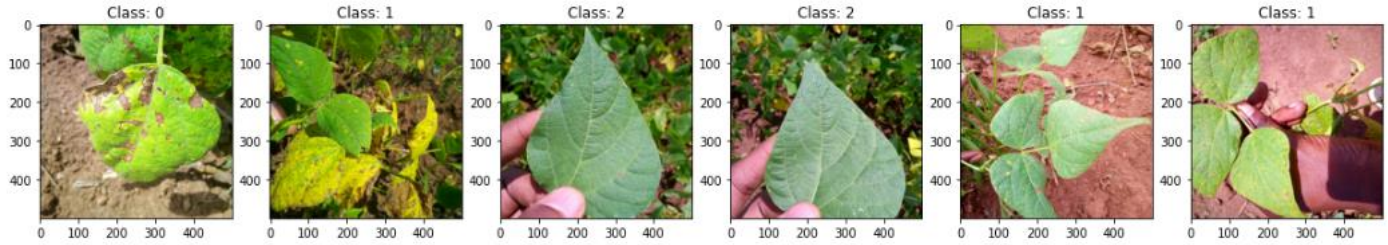
```
{'image': (TensorShape([500, 500, 3]), tf.uint8),
 'image/filename': (TensorShape([]), tf.string),
 'label': (TensorShape([]), tf.int64)}
```

Элемент является словарем, содержащим изображение 500x500, имя файла и метку класса (ту же информацию можно получить, обратившись к аттрибуту `dataset.element_spec`).

Нам осталось применить аугментации, объединить изображения в батчи и обучить модель. Аугментации можно встроить в модель (как мы делали это ранее) либо использовать функцию `map`.

Важно! Применяя аугментации (с помощью `tfa.image`, preprocessing layers, albumentations или любых других средств) всегда следите, в каком формате ваше изображение (float или uint8) и каков диапазон значений пикселей. В документации к функциям как правило написано о том, в каком формате они принимают и возвращают изображение.

```
import tensorflow as tf
import tensorflow_addons as tfa

# отбрасываем имена файлов и возвращаем кортеж (изображение, метка)
def to_tuple(image_dict):
  return image_dict['image'], tf.one_hot(image_dict['label'], depth=3)

dataset_augmented = dataset.map(to_tuple).batch(32)

show_first_images(dataset_augmented.as_numpy_iterator())
```



Советы

Следите за производительностью аугментаций. Создав пайплайн, запустите процесс расчета аугментаций на всем датасете и измерьте время. Так вы узнаете сколько процентов времени тратится на аугментации, а сколько остается на обучение.

Если аугментации отнимают слишком много вычислительных ресурсов, попробуйте найти ту операцию, которая занимает наибольший процент времени, и убрать ее или заменить на другую.

Если ваш датасет не слишком большой, а вы хотите повторять процесс обучения многократно, то вы можете выполнить наиболее «вычислительно тяжелые» аугментации заранее и сохранить по 10-20 разных вариантов изображения на диск, а в пайплайне оставить только «легкие» аугментации.

Если вам требуется максимальная производительность, то мы можете выполнять аугментации на одном сервере и в режиме реального времени передавать на другой сервер, где обучается модель.

Вы можете использовать файлы .tfrec для сохранения и загрузки изображений. В этом случае на диск записываются большие файлы, содержащие множество изображений, поэтому операция чтения с диска выполняется быстрее, чем если каждое изображение хранилось бы в виде отдельного файла.

В целом для оптимизации процесса обучения следует найти «бутылочное горлышко» (bottleneck) – операцию, которая выполняется дольше всех и

задерживает функционирование остальных узлов системы. Это может быть чтение с диска, передача по сети, аугментации или обучение. В первую очередь займитесь оптимизацией именно этой операции.

11. Решение CV-задачи в Keras, часть 1

Постановка задачи

Мы будем обучать нейронную сеть на датасете **Facial expression comparision (FEC)**. Исходными данными в датасете FEC являются тройки (триплеты) изображений, целевыми – номер изображения, которое наиболее непохоже на остальные. Таким образом, мы будем обучать модель находить то изображение в тройке, которое наиболее отличается от остальных.

Как это будет работать?

Входными данными сети будут являться изображения с лицами людей, выходными данными - вектора. Размер вектора можно выбрать произвольно в процессе построения сети, например, если выходной слой сети - `Dense(16)`, то сеть будет выдавать вектор длиной 16 для каждого изображения.

Идея в том, что для похожих по выражению лиц сеть выдает близкие (по евклидовому расстоянию) вектора, а для непохожие - далекие друг от друга вектора. Поэтому если мы передали на вход сети **f** изображения **A**, **B**, **C**, и вектор **f(C)** находится дальше от векторов **f(A)**, **f(B)**, чем эти вектора друг от друга, то значит изображение **C** наиболее отличается от остальных в данной тройке.

Для опимизации мы будем применять так называемый **triplet loss**. Это функция, которая принимает 3 вектора и возвращает число. Позже мы рассмотрим ее подробнее.

Зачем это нужно?

Вектора, которые выдает сеть, называются латентными признаками (latent features). Мы знаем, что эти вектора близки для похожих изображений и далеки для непохожих.

Такие вектора затем можно будет использовать как входные данные для другой нейронной сети, и эти вектора будут хорошо «кодировать» выражение лица на фотографии. Например, мы сможем объединить информацию о выражении лица с информацией о звуке и тексте, произносимом говорящим на видео, чтобы автоматически определять его эмоциональное состояние. Подробнее в видео.

Какие навыки потребуются?

В этом разделе часть кода вам предлагается написать самостоятельно. Для выполнения задания нужно изучить модуль по TensorFlow и Keras, уметь работать с библиотекой numpy и с изображениями. Для выполнения заданий часть информации придется искать в интернете: это полезный навык для программиста.

> К данному разделу прилагается ноутбук, в котором вам потребуется дописать недостающие участки кода и таким образом выполнить задания данного раздела. В случае затруднений вы можете открыть подсказку, пролистав эту страницу до нужного задания.

Для редактора: ноутбук
https://colab.research.google.com/drive/1u505mKUZ1H6jd6A5j4bE3WplN3XMjbne?usp=sharing

Скачиваем данные

```
!wget -q --show-progress https://storage.googleapis.com/oleg-zyablov/dlcourse/fec_dataset/labels.csv
!wget -q --show-progress https://storage.googleapis.com/oleg-zyablov/dlcourse/fec_dataset/images.zip
!unzip -q -o images.zip && rm images.zip
!nvidia-smi -L #проверяем наличие совместимой в TensorFlow видеокарты

import pandas as pd
df = pd.read_csv('labels.csv')
df
```

Фотографии лежат в папке `images_cropped` с именами файлов в формате `{id}.jpg`, в виде цветных изображений размером 112x112.

Разметка каждой тройки делалась с помощью шести и более аннотаторов, которые могли дать разные ответы. Столбец `votes{i}` означает долю аннотаторов, которые проголосовали за это изображение как за наиболее непохожее на остальные.

Со столбцом `triplet_type` мы работать не будем.

Задание 11.1

Посчитайте количество изображений в папке `images_cropped`.

Ответ: 136298 или 136,298

Решение:

```
# первый вариант (linux)
!ls images_cropped | wc -l

# второй вариант
import glob
print(len(glob.glob('images_cropped/*')))
```

Задание 11.2

Найдите количество уникальных id лиц, которые присутствуют в обеих выборках.

Подсказка: используйте [множества (sets)](#).

Ответ: 0

Решение:

```
train_df = df[df.subset == 'train']
test_df = df[df.subset == 'test']
train_faces = set(train_df.face1.tolist() \
     + train_df.face2.tolist() + train_df.face3.tolist())
test_faces = set(test_df.face1.tolist() \
     + test_df.face2.tolist() + test_df.face3.tolist())
train_faces_count = len(train_faces)
test_faces_count = len(test_faces)
train_and_test_faces_count = len(train_faces.intersection(test_faces))
```

Изучим тройки визуально, это всегда важно делать при работе с датасетом. В коде ниже метод `get_sample()` возвращает случайную тройку из датасета: 3 изображения в виде numpy-массивов в формате RGB и ответы к ним.

```
import cv2, os
import matplotlib.pyplot as plt
from matplotlib import gridspec
import numpy as np

images_dir = os.path.join(os.getcwd(), 'images_cropped')
def get_sample():
  row = df.sample(1).iloc[0]
  votes = [row.votes1, row.votes2, row.votes3]
  face_ids = [row.face1, row.face2, row.face3]
  filenames = [os.path.join(images_dir, f'{id}.jpg') for id in face_ids]

  # YOUR CODE STARTS
  face_images = ...
  # YOUR CODE ENDS

  # Проверка:
  for img in face_images:
```

```
    assert img.shape == (112, 112, 3)
    assert img.dtype == np.uint8

  return face_images, votes

fig = plt.figure(figsize=(24, 9))
outer = gridspec.GridSpec(4, 3)
for i in range(9):
    inner = gridspec.GridSpecFromSubplotSpec(1, 3, subplot_spec=outer[i])
    axes = [plt.Subplot(fig, inner[j]) for j in range(3)]
    for ax in axes:
      fig.add_subplot(ax)

    face_images, votes = get_sample()
    for (img, vote, ax) in zip(face_images, votes, axes):
      ax.imshow(img)
      ax.set_title(f'{vote:g}')
      ax.axis('off')

plt.show()
```

Задание 11.3

Допишите метод `get_sample()`. В переменной `face_images` должен оказаться список из трех изображений в формате RGB. Загрузите изображения с диска, например с помощью OpenCV. Помните, что OpenCV загружает изображения в формате BGR.

☑ У меня получилось!

Решение:

```
face_images = [cv2.imread(filename)[..., ::-1] for filename in filenames]
```

Задача найти наиболее непохожее на другие лицо – неоднозначная, и аннотаторы часто не сходятся между собой в ответах.

Создаем генераторы

В качестве генераторов данных можно использовать `Sequence`, `ImageDataGenerator` или `Dataset`. Мы будем использовать `Dataset`.

Для того, чтобы расчитать функцию потерь, мы должны пропустить через модель сразу три изображения. Это говорит о том, что стандартный вариант, в котором батч состоит из наборов независимых изображений и ответов к ним, нам не подходит.

Каким же образом мы будем строить цикл обучения?

Конечно мы можем написать цикл обучения вручную, пропуская через модель по 3 изображения, расчитывая loss и делая обратный проход. Это означает, что мы пропускаем через модель батч из 3 изображений. Такой размер батча слишком мал, и мы сильно потеряем в производительности. Нужно чтобы размер батча был больше.

Это означает, что нам нужно брать сразу несколько троек изображений, объединять их в батч и пропускать через модель.

Чтобы создать генератор, используя `tf.data.Dataset`, нам нужно определить источник данных и цепочку преобразований. Источником данных является датафрейм:

```
import tensorflow as tf

def dataframe_to_dataset(df):
  return tf.data.Dataset.from_tensor_slices(
      (df.face1, df.face2, df.face3,
       df.votes1, df.votes2, df.votes3)
  )

train_dataset = dataframe_to_dataset(df[df.subset == 'train'])
test_dataset = dataframe_to_dataset(df[df.subset == 'test'])
```

Мы создали первое звено цепочки преобразований. Эти генераторы возвращают кортежи из 6 чисел:

1. 3 идентификатора лица (колонки `face1, face2, face3`)
2. 3 числа, описывающие распределение голосов аннотаторов (колонки `votes1, votes2, votes3`)

В следующем звене мы заменим идентификаторы изображений на сами изображения, считав их с диска. Разрешение изображений мы изменим на 160x160. Полученные изображения мы объединим в батч из 3 изображений (на данном шаге). Целевые данные мы также объединим в один массив.

В итоге генератор будет возвращать пару из массивов формой (3, 160, 160, 3) и (3,).

```
def read_and_resize_image(id):
  path = 'images_cropped/' + tf.strings.as_string(id) + '.jpg'
  image = tf.image.decode_jpeg(tf.io.read_file(path), channels=3)

  # YOUR CODE STARTS
  image = ... #измените размер изображения на 160x160
  # YOUR CODE ENDS

  return image

def process_triplet(face1, face2, face3, votes1, votes2, votes3):
```

```
  img1 = read_and_resize_image(face1)
  img2 = read_and_resize_image(face2)
  img3 = read_and_resize_image(face3)
  return tf.stack([img1, img2, img3], axis=0),
         tf.stack([votes1, votes2, votes3], axis=0)

train_dataset = train_dataset.map(process_triplet)
test_dataset = test_dataset.map(process_triplet)

# проверка
for dataset in [train_dataset, test_dataset]:
  batch = next(iter(dataset))
  assert type(batch) == tuple and len(batch) == 2
  assert batch[0].shape == (3, 160, 160, 3)
  assert batch[1].shape == (3,)
```

Задание 11.4

Найдите в TensorFlow операцию, изменяющую размер изображения, и примените ее в функции `read_and_resize_image`, изменив размер изображения на 160x160.

☑ У меня получилось!

Решение:

```
image = tf.image.resize(image, (160, 160))
```

Задание 11.5

Проверьте, какой тип данных и диапазон значений пикселей имеют изображения, возвращаемые `train_dataset`?

○ uint8 от 0 до 255

○ float64 от 0 до 1

○ float32 от 0 до 1

◉ float32 от 0 до 255

Решение:

```
images, _ = next(iter(train_dataset))
print(images.dtype, images.numpy().max())
```

Важная деталь! В функциях `process_triplet` и `read_image` мы используем только операции TensorFlow и не используем операции из сторонних библиотек (таких как

numpy, openCV). При передаче функции `process_triplet` в метод `.map()` библиотека TensorFlow анализирует код функции `process_triplet` и строит статический граф вычислений (представьте трубопровод, по которому пока не течет вода). Граф вычислений затем выполняется на языке C++.

Теперь у нас есть батчи из 3 изображений, нам нужно объединить их в батчи большего размера. Пусть в батче будет 64 тройки изображений.

```
batch_size = 64
train_dataset = train_dataset.unbatch().batch(3*batch_size, drop_remainder=True
)
test_dataset = test_dataset.unbatch().batch(3*batch_size, drop_remainder=True)
```

Теперь обратитесь к документации чтобы узнать, как определить общую длину объекта `Dataset`.

Задание 11.6

Сколько батчей в обучающем и тестовом датасетах?

○ 1128384 и 97152

○ 376128 и 32384

◉ 5877 и 506

Решение:

```
len(train_dataset), len(test_dataset)
```

12. Решение CV-задачи в Keras, часть 2

Функция потерь и метрика

Нам осталось реализовать функцию потерь triplet loss по формуле из данной статьи. Для триплета (I1, I2, I3) с наиболее схожей парой (I1, I2) функция потерь задается формулой:

$$l(I_1, I_2, I_3) = max(0, \|e_{I_1} - e_{I_2}\|_2^2 - \|e_{I_1} - e_{I_3}\|_2^2 + \delta) + max(0, \|e_{I_1} - e_{I_2}\|_2^2 - \|e_{I_2} - e_{I_3}\|_2^2 + \delta),$$

Символом $\|x\|_2$ обозначается евклидова норма. Надстрочный индекс 2 означает возведение в квадрат.

```
@tf.function
def triplet_loss_on_single_triplet(far, near1, near2, delta):
  # YOUR CODE STARTS
  ...
  return ...
  # YOUR CODE ENDS

# проверка
loss = triplet_loss_on_single_triplet(tf.ones(16), tf.ones(16), tf.zeros(16), 0
.5)
assert isinstance(loss, tf.Tensor) #проверяем, что возвращен тензор
assert loss.dtype.is_floating #проверяем, что тип тензора - число с плавающей з
апятой
assert loss == 17.0 #проверяем, что возвращено верное значение
loss = triplet_loss_on_single_triplet(tf.ones(16), tf.zeros(16), tf.zeros(16),
0.5)
assert isinstance(loss, tf.Tensor)
assert loss.dtype.is_floating
assert loss == 0.0
```

Задание 12.1

Напишите функцию `triplet_loss_on_single_triplet`, которая считает triplet loss по указанной формуле и возвращает рассчитанное значение.

Параметры функции:

far – вектор, наиболее далекий от остальных (I3 в формуле). Массив с одной осью.

near1, near2 – векторы, близкие друг к другу. Массивы с одной осью.

delta – дельта в формуле

Для расчета можно использовать:

1. Арифметические операции
2. Функции tf.math.reduce_euclidean_norm и tf.math.maximum
3. Любые другие операции TensorFlow, какие сочтете нужными

☑ У меня получилось!

Решение:

```
@tf.function
def triplet_loss_on_single_triplet(far, near1, near2, delta):
```

```
  dist_n1n2 = tf.math.reduce_euclidean_norm(near1 - near2)**2
  dist_n1f = tf.math.reduce_euclidean_norm(near1 - far)**2
  dist_n2f = tf.math.reduce_euclidean_norm(near2 - far)**2
  return tf.math.maximum(0., dist_n1n2 - dist_n1f + delta) \
         + tf.math.maximum(0., dist_n1n2 - dist_n2f + delta)
```

Теперь мы немного модифицируем triplet loss так, чтобы он учитывал несогласие аннотаторов между собой. Затем напишем функцию, которая расчитывает средний loss по всему батчу.

```
# эта функция считает triplet loss на одном триплете, учитывая расхождения в от
ветах аннотаторов
def triplet_loss_on_single_triplet_by_votes(vec1, vec2, vec3, votes1, votes2, v
otes3, delta):
  #tf.print(vec1)
  loss1 = triplet_loss_on_single_triplet(vec1, vec2, vec3, delta)
  loss2 = triplet_loss_on_single_triplet(vec2, vec1, vec3, delta)
  loss3 = triplet_loss_on_single_triplet(vec3, vec1, vec2, delta)
  return votes1*loss1 + votes2*loss2 + votes3*loss3

# эта функция считает triplet loss на целом батче, состоящем из нескольких трип
летов
delta = 0.5
def triplet_loss(votes, features):
  features = tf.split(features, batch_size, axis=0)
  votes = tf.split(votes, batch_size, axis=0)
  losses = [triplet_loss_on_single_triplet_by_votes(f[0], f[1], f[2], v[0], v[1
], v[2], delta)
            for f, v in zip(features, votes)]
  loss = tf.math.reduce_mean(losses)
  return loss
```

Задание 12.2

Выберите верные утверждения:

☐ Функция `triplet_loss` считает **сумму** loss'ов по каждому триплету в батче и возвращает одно число.

☐ Если все векторы нулевые, то triplet loss равен нулю.

☐ Triplet loss может быть меньше нуля.

☑ Сумма votes1 + votes2 + votes3 всегда равна единице.

☑ Функция `triplet_loss_on_single_triplet_by_votes` считает взвешенное среднее.

☐ Если в евклидовом пространстве I1 и I2 находятся близко, а I3 находится очень далеко от них (например, на расстоянии 10*delta), то градиент triplet loss по вектору I3 будет равен нулю.

Решение:

1. Функция triplet_loss считает среднее по каждому триплету в батче и возвращает это среднее, то есть одно число.

2. Если все векторы нулевые, то triplet loss равен 2*delta (можно посчитать по формуле).

3. В формуле для triplet loss два слагаемых, каждое из них имеет вид max(0, ...), а значит не может быть меньше нуля.

4. Числа votes1, votes2, votes3 означают долю голосов, отданных за каждое изображение в тройке, поэтому их сумма всегда равна единице.

5. Функция triplet_loss_on_single_triplet_by_votes считает взвешенное среднее, используя votes1, votes2, votes3 как веса. За счет этого мы учитываем несогласие среди аннотаторов при разметке датасета.

6. В каждом из двух слагаемых triplet loss мы считаем разность квадратов двух расстояний. В описанном случае в каждом из двух слагаемых разность будет существенно меньше нуля, и даже прибавление дельта не сделает ее больше нуля, поэтому каждое слагаемое будет равно нулю. Функция max(0, A) имеет ненулевой градиент по A только если A > 0, поэтому градиент будет равен нулю. Это говорит о том, что если вектор I3 и так находится далеко от векторов I1, I2, то минимизация triplet loss не будет способствовать его еще большему отдалению. Если описанное выше не до конца понятно, то попробуйте нарисовать картинку, где I1, I2, I3 - вершины треугольника, а расстояния между ними - его ребра. Затем сопоставьте формулу для triplet loss с картинкой.

Кроме функции потерь хотелось бы ввести метрику качества. Мы будем считать точность, то есть долю верных ответов. Мы будем считать ответ модели верным, если в треугольнике с вершинами I1, I2, I3 ребро I1-I2 самое короткое. Примеры, в которых аннотаторы не согласны между собой (максимальная доля голосов не более 0.5) не будем учитывать.

Смысл triplet accuracy в следующем: для каждого триплета она равна единице только тогда, когда по выходным данным модели (векторам признаков) можно верно определить, какое из изображений сильнее отличается от остальных.

```
def triplet_accuracy_on_single_triplet(vec1, vec2, vec3, votes1, votes2, votes3
):
```

```
  vectors = tf.stack([vec1, vec2, vec3])
  votes = tf.stack([votes1, votes2, votes3])
  indices = tf.argsort(votes, direction='DESCENDING')
  votes = tf.gather(votes, indices, axis=0)
  vectors = tf.gather(vectors, indices, axis=0)
  if votes[0] <= 0.5:
    return np.nan
  far, near1, near2 = vectors[0], vectors[1], vectors[2]
  dist_n1n2 = tf.math.reduce_euclidean_norm(near1 - near2)**2
  dist_n1f = tf.math.reduce_euclidean_norm(near1 - far)**2
  dist_n2f = tf.math.reduce_euclidean_norm(near2 - far)**2
  is_answer_correct = (dist_n1f > dist_n1n2) & (dist_n2f > dist_n1n2)
  return tf.cast(is_answer_correct, tf.float32)

def triplet_accuracy(votes, features):
  features = tf.split(features, batch_size, axis=0)
  votes = tf.squeeze(votes)
  votes = tf.split(votes, batch_size, axis=0)
  accuracies = [triplet_accuracy_on_single_triplet(f[0], f[1], f[2], v[0], v[1]
, v[2])
                for f, v in zip(features, votes)]
  accuracy = tf.experimental.numpy.nanmean(accuracies)
  if tf.math.is_nan(accuracy):
    return 0.5
  return accuracy
```

Модель

Одна из самых современных сверточных архитектур – EfficientNetV2.

```
import tensorflow_hub as hub
def get_conv_net():
  # YOUR CODE STARTS
  return ...
  # YOUR CODE ENDS

# проверка
conv_net = get_conv_net()
assert 'trainable_weights' in dir(conv_net)
assert conv_net.trainable
# проверяем, что загружена верная модель
assert 'efficientnetv2-b0-21k/feature-
vector' in conv_net.get_config()['handle']
```

Задание 12.3

Изучите этот репозиторий и загрузите модель EfficientNetV2 B0, предобученную на датасете ImageNet21K (`efficientnetv2-b0-21k`), из Tensorflow Hub в виде слоя Keras (`KerasLayer`). Используйте тип «`feature-vector`» и параметр `trainiable=True`.

☑ У меня получилось!

Решение:

```
def get_conv_net():
  return hub.KerasLayer('gs://cloud-tpu-
checkpoints/efficientnet/v2/hub/efficientnetv2-b0-21k/feature-
vector', trainable=True)
```

Мы будем использовать L2-нормализацию выходных векторов (называемую еще «проекцией на гиперсферу») и умножение результата на константу, как предложено данной статье.

Согласно документации, для модели `efficientnetv2-b0-21k` требуется нормализация входного изображения от -1 до 1.

В результате архитектура модели будет выглядеть таким образом:

```
from tensorflow import keras
from tensorflow.keras import Sequential, layers

def get_model(embedding_size=16, scale=1, dropout=0.4):
  model = Sequential([
    layers.Lambda(lambda x: x/127.5 - 1), #нормализация
    get_conv_net(), #сверточная сеть EfficientNetV2
    layers.Dropout(dropout),
    layers.Dense(embedding_size), #выходной слой размера embedding_size
    layers.Lambda(lambda x: scale*tf.math.l2_normalize(x, axis=1)) #L2-
нормализация
  ])
  return model
```

Задание 12.4

Функция `get_model()` возвращает модель `Sequential`, включающую 5 слоев (один из слоев является вложенной моделью). Сколько из этих 5 слоев имеют обучаемые веса?

○ 1

◉ 2

○ 3

○ 4

Решение:

Обучаемыми являются слои `get_conv_net()` и `Dense`. Остальные слои являются stateless-слоями и не имеют обучаемых параметров.

Датасет имеет большой размер, поэтому эпоха будет длиться долгое время. Поэтому мы сократим длину эпохи, указав при обучении параметр `steps_per_epoch`. Также укажем этот параметр для валидации. Тем самым один проход по обучающему датасету займет несколько эпох.

В недавно вышедшей статье (видео) авторы сравнивают различные оптимизаторы. По результатам тестов AdaBound и AMSBound превосходят Adam в сверточных сетях. Поэтому мы попробуем использовать AdaBound.

```
!pip install keras-adabound -q
from keras_adabound import AdaBound
def train(model, epochs):
  model.compile(
      loss=triplet_loss,
      optimizer=AdaBound(learning_rate=1e-3, final_lr=0.1),
      metrics=triplet_accuracy
  )
  model.fit(train_dataset.repeat(), steps_per_epoch=200, validation_data=test_d
ataset, validation_steps=200, epochs=epochs)

train(get_model(), epochs=1)
```

Мы будем обучать модель в течение только одного полного прохода по обучающему датасету (для экономии времени). Поэтому давайте не использовать аугментации, они только ухудшат результат.

Мы не знаем заранее какие параметры `embedding_size` и `scale` (см. функцию `get_model()`) являются наилучшими. Предположим, что эти параметры влияют на результат независимо, и переберем несколько значений каждого из этих параметров, обучив модель на 200 батчах обучающего датасета и сделав валидацию на 200 батчах валидационного датасета.

Задание 12.5

Зафиксируйте некое значение `embedding_size` (например 16) и переберите несколько разных значений `scale` (на ваш выбор), сравнив точность на валидации после первой эпохи. Затем выберите наилучшее значение scale и переберите несколько разных значений `embedding_size`.

☑ У меня получилось!

Решение:

```
for scale in [1/4, 1/2, 1, 2, 4, 8]:
   print(scale)
   train(get_model(16, scale), epochs=1)

best_scale = 1
for dim in [8, 16, 32, 128, 1024]:
   print(dim)
   train(get_model(dim, best_scale), epochs=1)
```

У вас могли получиться значения `embedding_size=128`, `scale=1`. А могли получиться и другие. Один эксперимент с небольшой длительностью обучения достаточно ненадежен, но он лучше, чем ничего. Для надежности можно было обучать дольше и по нескольку раз. Кроме того параметры `embedding_size`, `scale` и `delta` явно связаны друг с другом и влияют на результат не независимо.

Теперь обучим нашу модель в течение 25 эпох. Вы можете выбрать и другое число эпох.

```
model = get_model(embedding_size=128, scale=1)
train(model, epochs=25)

import matplotlib.pyplot as plt
plt.plot(model.history.history['triplet_accuracy'])
plt.plot(model.history.history['val_triplet_accuracy'])
plt.show()
```

Измерим точность нашей модели на всем валидационном датасете.

```
model.evaluate(test_dataset, verbose=1);
```

ПОЗДРАВЛЯЕМ!

Вы завершили обучающий курс по TensorFlow & Keras!