Keras FAQ: Часто задаваемые Вопросы по Keras

Как мне процитировать Keras?

Пожалуйста, цитируйте Keras в своих публикациях, если это поможет вашему исследованию. Вот пример записи BibTeX:

```
@misc{chollet2015keras,
  title={Keras},
  author={Chollet, Fran\c{c}ois and others},
  year={2015},
  howpublished={\url{https://keras.io}},
}
```

Как запустить Keras на GPU?

Если вы работаете на бэкэндах TensorFlow или CNTK, ваш код будет автоматически запущен на GPU при обнаружении любого доступного GPU.

Если Вы работаете на бэкэнде Theano, Вы можете использовать один из следующих методов:

Метод 1: используйте флаги Theano.

```
THEANO_FLAGS=device=gpu,floatX=float32 python my_keras_script.py
```

Имя 'gpu' может быть изменено в зависимости от идентификатора Вашего устройства (например, gpu0, gpu1 и т.д.).

Метод 2: настройте ваш .theanorc: Инструкции

Метод 3: вручную установите theano.config.device, theano.config.floatX в начале кода:

import theano

theano.config.device = ‘gpu’

theano.config.floatX = ‘float32’

Как запустить модель Keras на нескольких GPU?

Мы рекомендуем сделать это с помощью бэкэнда TensorFlow. Есть два способа запустить одну модель на нескольких GPU: параллелизм данных и параллелизм устройств.

В большинстве случаев, то, что вам нужно, это, скорее всего, параллелизм данных.

Параллелизм данных

Параллелизм данных состоит в том, чтобы реплицировать целевую модель один раз на каждое устройство и использовать каждую реплику для обработки разной доли входных данных. Keras имеет встроенную утилиту keras.utils.multi_gpu_model, которая может создать параллельную версию данных для любой модели, и достигает квазилинейного ускорения на 8 GPU.

Подробнее см. документацию по модели multi_gpu_model. Приведем краткий пример:

from keras.utils **import** multi_gpu_model

 # Реплицирует «model» на 8 GPU.# Это предполагает, что на вашей машине есть 8 доступных GPU.

parallel_model = multi_gpu_model(model, gpus=8)

parallel_model.compile(loss=’categorical_crossentropy’,

```
                optimizer='rmsprop')

# Этот вызов «fit» будет распространяться на 8 GPU.# Так как размер пакета 256, каждый GPU будет обрабатывать 32 сэмпла.

parallel_model.fit(x, y, epochs=20, batch_size=256)
```

Параллелизм устройств

Параллелизм устройств состоит в том, что на разных устройствах работают разные части одной и той же модели. Лучше всего он работает для моделей, имеющих параллельную архитектуру, например, модель с двумя ветвями.

Этого можно добиться с помощью диапазонов устройств TensorFlow. Приведем небольшой пример:

```
# Модель, где общий LSTM используется для кодирования двух различных последовательностей параллельно.

input_a = keras.Input(shape=(140, 256))

input_b = keras.Input(shape=(140, 256))

shared_lstm = keras.layers.LSTM(64)

# Обрабатываем первую последовательность на одном GPU

with tf.device_scope('/gpu:0'):

encoded_a = shared_lstm(tweet_a)# Обрабатываем следующую последовательность на другом GPU

with tf.device_scope('/gpu:1'):

  encoded_b = shared_lstm(tweet_b)

# Связываем результаты на процессоре

with tf.device_scope('/cpu:0'):

  merged_vector = keras.layers.concatenate([encoded_a, encoded_b],

                                           axis=-1)
```

Что означает «сэмпл», «партия», «эпоха»?

Ниже приведены некоторые общие определения, которые необходимо знать и понимать, чтобы правильно использовать Keras:

Сэмпл: один элемент набора данных.

Пример: одно изображение — это образец в конвулуационной сети.

Пример: один звуковой файл является образцом для модели распознавания речи.

Партия: набор N образцов. Образцы в партии обрабатываются независимо, параллельно. В случае обучения, партия приводит только к одному обновлению модели.

Как правило, партия аппроксимирует распределение входных данных лучше, чем один вход. Чем больше партия, тем лучше аппроксимация; однако верно и то, что обработка партии займет больше времени и все равно приведет только к одному обновлению. Для вывода (оценка/предсказание) рекомендуется выбрать такой размер пакета, который будет как можно больше, не выходя из памяти (так как большие партии обычно приводят к более быстрому выводу/предсказанию).

Epoch: произвольное отсечение, обычно определяемое как «один проход по всему набору данных», используемое для разделения обучения на отдельные фазы, что полезно для протоколирования и периодической оценки.

При использовании данных validation_data или validation_split с методом подгонки моделей Keras, оценка будет выполняться в конце каждой эпохи.

В Keras есть возможность добавить обратные вызовы, специально разработанные для запуска в конце эпохи. Примерами таких возможностей являются изменения скорости обучения и проверка (сохранение) модели.

Как сохранить модель Keras?

Сохранение/загрузка целых моделей (архитектура + вес + состояние оптимизатора)

Не рекомендуется использовать pickle или cPickle для сохранения модели Keras.

Вы можете использовать model.save(filepath), чтобы сохранить модель Keras в одном файле HDF5, который будет содержать:

архитектуру модели, позволяющую воссоздать ее заново.

вес модели

конфигурация обучения (потеря, оптимизатор)

состояние оптимизатора, позволяющее возобновить тренировки именно там, где вы остановились.

Затем вы можете использовать keras.models.load_model(filepath) для переустановки модели. load_model также позаботится о компиляции модели, используя сохраненную конфигурацию обучения (если только модель никогда не компилировалась).

Пример:

from keras.models **import** load_model

model.save('my_model.h5') *# создает файл HDF5 'my_model.h5'* **del** model *# удаляет существующую модель*

 # возвращает скомпилированную модель #, идентичную предыдущей.

model = load_model('my_model.h5')

Смотрите Как установить HDF5 или h5py для сохранения моделей в Keras? для получения инструкций по установке h5py.

Сохранение/загрузка только архитектуры модели

Если вам нужно только сохранить архитектуру модели, а не ее вес или тренировочную конфигурацию, то вы можете сделать следующее:

 # сохранить как JSON

json_string = model.to_json()

 # сохранить как YAML

yaml_string = model.to_yaml()

Генерируемые JSON / YAML файлы читаются человеком и при необходимости могут быть отредактированы вручную.

Из этих данных можно построить свежую модель:

*# реконструкция модели из JSON:***from** keras.models **import** model_from_json

model = model_from_json(json_string)

*# реконструкция модели из YAML:***from** keras.models **import** model_from_yaml

model = model_from_yaml(yaml_string)

Сохранение/загрузка только весов модели.

If you need to save the **weights of a model**, you can do so in HDF5 with the code below:

model.save_weights('my_model_weights.h5')

Если предположить, что у вас есть код для инстанцирования вашей модели, то вы можете загрузить сохраненные веса в модель с той же архитектурой:

model.load_weights('my_model_weights.h5')

Если вам нужно загрузить веса в другую архитектуру (с некоторыми общими слоями), например, для тонкой настройки или переноса-обучения, вы можете загрузить их по имени слоя:

model.load_weights('my_model_weights.h5', by_name=**True**)

Пример:

«»»

Assuming the original model looks like this:

```
model = Sequential()
```

```
  model.add(Dense(2, input_dim=3, name='dense_1'))
  model.add(Dense(3, name='dense_2'))
  ...
  model.save_weights(fname)
"""
# новая модель
model = Sequential()
model.add(Dense(2, input_dim=3, name='dense_1'))  # будет загружена
model.add(Dense(10, name='new_dense'))  # не будет загружена
# загрузить веса первой модели; относится только к последнему слою, dense_1.
model.load_weights(fname, by_name=True)
```

Также смотрите Как установить HDF5 или h5py для сохранения моделей Keras? Для получения инструкций по установке h5py

Обработка пользовательских слоев (или других пользовательских объектов) в сохраненных моделях.

Если модель, которую вы хотите загрузить, включает пользовательские слои или другие пользовательские классы или функции, вы можете передать их в механизм загрузки через аргумент custom_objects:

```
from keras.models import load_model# Предполагая, что ваша модель включает в себя использование класса «AttentionLayer».
model = load_model('my_model.h5', custom_objects={'AttentionLayer': AttentionLayer})
```

В качестве альтернативы можно использовать пользовательский диапазон объектов:

```
from keras.utils import CustomObjectScope
```

with CustomObjectScope({'AttentionLayer': AttentionLayer}):

  model = load_model('my_model.h5')

Работа с пользовательскими объектами работает точно так же
for load_model, model_from_json, model_from_yaml:

from keras.models **import** model_from_json

model = model_from_json(json_string, custom_objects={'AttentionLayer':
AttentionLayer})

Почему потери при обучении намного выше, чем потери при тестировании?

Модель Keras имеет два режима: обучение и тестирование. Механизмы регуляризации, такие как отсев и регуляризация веса L1/L2, отключаются во время тестирования.

Кроме того, потери от тренировок — это среднее значение потерь по каждой партии тренировочных данных. Поскольку ваша модель меняется со временем, потери по первым партиям эпохи обычно выше, чем по последним партиям. С другой стороны, потери при тестировании для эпохи вычисляются с использованием модели, как она есть в конце эпохи, что приводит к более низким потерям.

Как я могу получить выход промежуточного слоя?

Одним из простых способов является создание новой Модели, которая будет выводить интересующие вас слои:

from keras.models **import** Model

model = ... *#создать оригинальную модель*

layer_name = 'my_layer'

```
intermediate_layer_model = Model(inputs=model.input,
                                 outputs=model.get_layer(layer_name).output)
intermediate_output = intermediate_layer_model.predict(data)
```

В качестве альтернативы можно построить функцию Keras, которая будет возвращать вывод определенного слоя, например, при определенном входе:

```
from keras import backend as K

# с последовательной моделью
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
layer_output = get_3rd_layer_output([x])[0]
```

Точно так же можно построить функцию Theano и TensorFlow непосредственно.

Обратите внимание, что если ваша модель имеет различное поведение на этапе обучения и тестирования (например, если она использует Dropout, BatchNormalization и т.д.), вам нужно будет передать маркер фазы обучения в вашу функцию:

```
get_3rd_layer_output = K.function([model.layers[0].input, K.learning_phase()],
                                  [model.layers[3].output])

# вывод в тестовом режиме = 0
layer_output = get_3rd_layer_output([x, 0])[0]

# выход в режиме обучения = 1
layer_output = get_3rd_layer_output([x, 1])[0]
```

Как использовать Keras с наборами данных, которые не помещаются в памяти?

Вы можете проводить пакетное обучение, используя model.train_on_batch(x, y) и model.test_on_batch(x, y). См. документацию по моделям.

В качестве альтернативы можно написать генератор, который выдает партии обучающих данных и использовать метод model.fit_generator(data_generator, steps_per_epoch, epochs).

Пример обучения партиями можно посмотреть в нашем примере CIFAR10.

Как я могу прервать обучение, когда потеря валидации больше не уменьшается?

Вы можете использовать обратный вызов EarlyStopping:

```
from keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(monitor='val_loss', patience=2)

model.fit(x, y, validation_split=0.2, callbacks=[early_stopping])
```

Узнайте больше в документации по обратным вызовам.

Как рассчитывается разделение валидации?

Если вы установите аргумент validation_split в model.fit, например, в 0.1, то используемые данные валидации будут последними 10% данных. Если вы установите значение 0.25, то это будут последние 25% данных и т.д. Обратите внимание, что данные не перемешиваются перед извлечением дроби валидации, поэтому валидация — это буквально последние x% выборок во входных данных, которые вы передали.

Один и тот же набор валидации используется для всех эпох (в пределах одного и того же вызова fit).

Данные перемешиваются во время обучения?

Да, если аргумент shuffle в model.fit установлен в значение True (по умолчанию), то тренировочные данные будут случайным образом перетасовываться в каждую эпоху.

Данные проверки никогда не перетасовываются.

Как я могу записать данные обучения/потерь/точности в каждую эпоху?

Метод model.fit возвращает обратный вызов истории, который имеет атрибут истории, содержащий списки последовательных потерь и другие метрики.

hist = model.fit(x, y, validation_split=0.2)

print(hist.history)

Как я могу «заморозить» слои Keras?

Заморозить» слой означает исключить его из тренировки, т.е. его вес никогда не будет обновляться. Это полезно в контексте тонкой настройки модели или использования фиксированных встраиваний для ввода текста.

Можно передать обучаемый аргумент (булев) в конструктор слоя, чтобы установить слой как не обучаемый:

frozen_layer = Dense(32, trainable=**False**)

Кроме того, можно установить обучаемое свойство слоя в значение True или False после инстанцирования. Для того, чтобы это действие вступило в силу, вам нужно будет вызвать компилятор (compile()) на вашей модели после изменения обучаемого свойства. Вот пример:

x = Input(shape=(32,))

layer = Dense(32)

layer.trainable = **False**

y = layer(x)

frozen_model = Model(x, y)*# в модели ниже, веса `слоя` не будут обновляться во время обучения*

frozen_model.compile(optimizer='rmsprop', loss='mse')

layer.trainable = **True**

trainable_model = Model(x, y)*# с этой моделью веса слоя будут обновляться во время обучения# (что также повлияет на вышеуказанную модель, так как она использует тот же самый экземпляр слоя)*

trainable_model.compile(optimizer='rmsprop', loss='mse')

frozen_model.fit(data, labels) *# это НЕ обновляет веса `слоя`.*

trainable_model.fit(data, labels) *# это обновляет веса `слоя`*

Как я могу использовать RNN состояния?

Создание RNN stateful означает, что состояния для образцов каждой партии будут повторно использованы в качестве начальных состояний для образцов следующей партии.

Поэтому при использовании RNN с контролем состояния предполагается, что:

все партии имеют одинаковое количество отсчетов

Если x1 и x2 — последовательные партии образцов, то x2[i] — это последующая последовательность к x1[i], для каждого i.

Чтобы использовать статистику в RNN, необходимо:

явно указать размер используемой партии, передав аргумент batch_size на первый уровень вашей модели. Например, параметр batch_size=32 для партии из 32-х выборок, состоящей из последовательностей 10 разрядов по 16 разрядов.

Установите stateful=True на уровне RNN.

обозначьте shuffle=False при вызове функции fit().

Для сброса накопленных состояний:

используйте model.reset_states() для сброса состояний всех слоев модели.

используйте layer.reset_states() для сброса состояний конкретного RNN слоя с учетом состояния.

Пример:

x *# это наши входные данные, формы (32, 21, 16)# мы передадим их нашей модели в последовательностях длиной 10*

model = Sequential()

model.add(LSTM(32, input_shape=(10, 16), batch_size=32, stateful=**True**))

model.add(Dense(16, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

 # мы тренируем сеть, чтобы предсказать 11-ый раз, учитывая первые 10:

model.train_on_batch(x[:, :10, :], np.reshape(x[:, 10, :], (32, 16)))

 # состояние сети изменилось. Мы можем добавлять последующие последовательности:

model.train_on_batch(x[:, 10:20, :], np.reshape(x[:, 20, :], (32, 16)))

 # давайте сбросим состояния слоя LSTM:

model.reset_states()

 # другой способ сделать это в данном случае:

model.layers[0].reset_states()

Обратите внимание, что методы predict, fit, train_on_batch, preict_classes и т.д. будут обновлять слои состояния в модели. Это позволяет не только проводить обучение с учетом состояния, но и предсказывать его.

Как удалить слой из последовательной модели?

Последний добавленный слой в последовательной модели можно удалить с помощью вызова .pop():

```
model = Sequential()

model.add(Dense(32, activation='relu', input_dim=784))

model.add(Dense(32, activation='relu'))

print(len(model.layers))  # «2»

model.pop()

print(len(model.layers))  # «1»
```

Как использовать предварительно обученные модели в Keras?

Коды и предварительно подготовленные веса доступны для следующих моделей классификации изображений:

- Xception
- VGG16
- VGG19
- ResNet
- ResNet v2
- ResNeXt
- Inception v3
- Inception-ResNet v2
- MobileNet v1
- MobileNet v2
- DenseNet
- NASNet

Они могут быть импортированы из модуля keras.applications:

```
from keras.applications.xception import Xception
from keras.applications.vgg16 import VGG16
from keras.applications.vgg19 import VGG19
from keras.applications.resnet import ResNet50
from keras.applications.resnet import ResNet101
from keras.applications.resnet import ResNet152
from keras.applications.resnet_v2 import ResNet50V2
from keras.applications.resnet_v2 import ResNet101V2
from keras.applications.resnet_v2 import ResNet152V2
from keras.applications.resnext
```

import ResNeXt50**from** keras.applications.resnext **import** ResNeXt101**from** keras.applications.inception_v3 **import** InceptionV3**from** keras.applications.inception_resnet_v2 **import** InceptionResNetV2**from** keras.applications.mobilenet **import** MobileNet**from** keras.applications.mobilenet_v2 **import** MobileNetV2**from** keras.applications.densenet **import** DenseNet121**from** keras.applications.densenet **import** DenseNet169**from** keras.applications.densenet **import** DenseNet201**from** keras.applications.nasnet **import** NASNetLarge**from** keras.applications.nasnet **import** NASNetMobile

model = VGG16(weights='imagenet', include_top=**True**)

Несколько простых примеров использования см. в документации к модулю «Приложения».

Подробный пример использования такой предварительно подготовленной модели для извлечения функций или тонкой настройки см. в этом посте блога.

Модель VGG16 также является основой для нескольких примеров сценариев Keras:

- Передача стиля
- Визуализация функций
- Глубокий сон

Как я могу использовать входы HDF5 с Keras?

Вы можете использовать класс HDF5Matrix из keras.utils. Подробности см. в документации HDF5Matrix.

Вы также можете напрямую использовать набор данных HDF5:

import h5py**with** h5py.File('input/file.hdf5', 'r') **as** f:

  x_data = f['x_data']

  model.predict(x_data)

Смотрите Как установить HDF5 или h5py для сохранения моделей в Keras? для получения инструкций по установке h5py.

Где хранится файл конфигурации Keras?

Каталог по умолчанию, в котором хранятся все данные Keras:

$HOME/.keras/

Обратите внимание, что пользователи Windows должны заменить $HOME на %USERPROFILE%. Если Keras не может создать указанную выше директорию (например, из-за проблем с разрешениями), в качестве резервной копии используется /tmp/.keras/.

Конфигурационный файл Keras — это JSON-файл, который хранится в $HOME/.keras/keras.json. По умолчанию конфигурационный файл выглядит следующим образом:

```
{
    «image_data_format»: «channels_last»,
    «epsilon»: 1e-07,
    «floatx»: «float32»,
    «backend»: «tensorflow»
}
```

Он содержит следующие поля:

- Формат данных изображения, используемый по умолчанию слоями обработки изображений и утилитами (channels_last или channels_first).
- Эпсилоновый числовой коэффициент размытия, который будет использоваться для предотвращения деления на ноль в некоторых операциях.
- Тип данных с плавающей запятой по умолчанию.
- Бэкэнд по умолчанию. См. документацию по бэкэнду.

Аналогично, кэшированные файлы набора данных, например, загруженные функцией get_file(), по умолчанию хранятся в $HOME/.keras/datasets/.

Как я могу получить воспроизводимые результаты с помощью Keras во время разработки?

При разработке модели иногда полезно иметь возможность получать воспроизводимые результаты от пробега к прогону, чтобы определить, связано ли изменение производительности с фактическим изменением модели или данных, или просто с результатом новой случайной выборки.

Сначала необходимо установить переменную окружения PYTHONHASHSEED на 0 перед запуском программы (а не внутри самой программы). Это необходимо в дальнейшем на Python 3.2.3, чтобы иметь воспроизводимое поведение для определённых операций, основанных на хэшах (например, порядок следования элементов в наборе или диктате, см. документацию на Python или выпуск #2280 для более подробной информации). Одним из способов установки переменной окружения является запуск Python таким образом:

$ cat test_hash.py

print(hash(«keras»))**$** python3 test_hash.py *# невоспроизводимый хэш (Python 3.2.3+)*

-8127205062320133199**$** python3 test_hash.py *# невоспроизводимый хэш (Python 3.2.3+)*

320448064215646159 1**$** PYTHONHASHSEED=0 python3 test_hash.py *# воспроизводимый хэш*

488366495143474947 6**$** PYTHONHASHSEED=0 python3 test_hash.py *# воспроизводимый хэш*

488366495143474947 6

Более того, при использовании бэкэнда TensorFlow и работе на GPU некоторые операции имеют недетерминированные выходы, в частности tf.reduce_sum(). Это связано с тем, что GPU выполняет много операций параллельно, поэтому порядок выполнения не всегда гарантирован. Из-за ограниченной точности флотов даже сложение нескольких чисел вместе может дать несколько разных результатов в зависимости от того, в каком порядке вы их добавляете. Можно попытаться избежать недетерминированных операций, но некоторые из них могут быть созданы автоматически с помощью TensorFlow для вычисления градиентов, поэтому намного проще простого запускать код на CPU. Для этого можно,

например, установить переменную окружения CUDA_VISIBLE_DEVICES на пустую строку:

$ CUDA_VISIBLE_DEVICES=»» PYTHONHASHSEED=0 python your_program.py

Приведенный ниже фрагмент кода представляет собой пример того, как получить воспроизводимые результаты — он ориентирован на бэкэнд TensorFlow для среды Python 3:

```
import numpy as npimport tensorflow as tfimport random as rn

# Следующее необходимо для запуска Numpy сгенерированных случайных чисел# в четко определенном начальном состоянии.

np.random.seed(42)

# Ниже необходимо, чтобы запустить ядро Python, генерирующее случайные числа# в четко определенном состоянии.

rn.seed(12345)

# Заставить TensorFlow использовать один поток.# Многопоточность является потенциальным источником невоспроизводимых результатов.# Подробнее см. https://stackoverflow.com/questions/42022950/.

session_conf = tf.ConfigProto(intra_op_parallelism_threads=1,

                              inter_op_parallelism_threads=1)

from keras import backend as K

# Ниже tf.set_random_seed() сделает генерацию случайных чисел# в бэкенде TensorFlow хорошо определенным начальным состоянием.# Подробнее смотрите:# https://www.tensorflow.org/api_docs/python/tf/set_random_seed.

tf.set_random_seed(1234)

sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)

K.set_session(sess)

# Остаток кода следует ...
```

Как установить HDF5 или h5py для сохранения моделей в Keras?

Для сохранения моделей Keras в виде файлов HDF5, например, через keras.callbacks.ModelCheckpoint, Keras использует пакет h5py Python. Он является зависимой программой Keras и должен быть установлен по умолчанию. На дистрибутивах, основанных на Debian, вам нужно будет дополнительно установить libhdf5:

sudo apt-get **install** libhdf5-serial-dev

Если вы не уверены, установлен ли h5py, вы можете открыть оболочку Python и загрузить модуль с помощью команды

import h5py

Если он импортируется без ошибок, то в противном случае вы можете найти подробные инструкции по установке здесь: http://docs.h5py.org/en/latest/build.html.