УНИВЕРСИТЕТ
искусственного
интеллекта

Написание нейронных сетей с библиотекой Tensorflow

Часть 1

УНИВЕРСИТЕТ
искусственного интеллекта

Написание нейронных сетей с библиотекой Tensorflow

Часть 1

На данный момент библиотека Tensorflow стала настолько большой, что уже переросла в экосистему. Предоставляя сервисы по интеграции нейросетей (TF Serving), хранению моделей в облачных хранилищах (TensorFlow Hub), даже API по распознаванию объектов (Tensorflow Object Detection API), сама библиотека может использоваться для написания и обучения нейросетей на глубоком уровне.

Основное отличие от верхнеуровневого использования - это возможность модификации стандартных методов и функции, возможность создавать свои подходы и решения к задачам.

Нейронны сети с Tensorflow

Тип данных Тензор

Основной тип данных для работы называется `tf.Tensor`. Название Тензор не совсем связано с математическим определением в линейной алгебре wiki/Тензор, но имеет много общего.

Библиотека импортируется стандартной записью

```
import tensorflow as tf

# Создаем тензор
# Мы можем присвоить ему тип данных, например, float16
# Если этого не сделать, по умолчанию для целых чисел
будет использоваться int32

rank_0_tensor = tf.constant(4, dtype=tf.float16)

print('Это тензор, созданный нами. Вся информация о нем, в
том числе и тип данных: ')
print(rank_0_tensor)
```

```
Это тензор, созданный нами. Вся информация о нем, в том
числе и тип данных:
 tf.Tensor(4.0, shape=(), dtype=float16)
```

В этом коде мы создали константную величину, которая имеет особенность: её нельзя изменить без прямого указания на изменение. В типе `tf.constant` может храниться любой стандартный тип python.

Операции с тензорами

С тензорами возможны все стандартные математические действия: сложение, умножение, возведение в степень, матричное умножение. Примеры таких действий и результат записаны в коде:

Нейронны сети с Tensorflow

```
print(tf.add(1, 2)) # Сложение
tf.Tensor(3, shape=(), dtype=int32)

print(tf.add([1, 2], [3, 4])) # Сложить по элементам
 tf.Tensor([4 6], shape=(2,), dtype=int32)

 print(tf.square(5)) # Возведение в квадрат
 tf.Tensor(25, shape=(), dtype=int32)

 print(tf.reduce_sum([[1, 2, 3],[4, 5, 6]])) # Сложение
всех элементов
 tf.Tensor(21, shape=(), dtype=int32)

 print(tf.square(2) + tf.square(3)) # Перегрузка
операторов
 tf.Tensor(13, shape=(), dtype=int32)
```

Матричное умножение можно писать через оператор @

```
a = tf.constant([[1,2],[3,5]])
b = tf.constant([[2,1],[0,6]])

x = tf.matmul(a, b) # Матричное умножение
print(x)
print(x.shape)
print(x.dtype)
print(a@b)
```

Дополнительные методы вычисления указаны в коде ноутбука, дополнительная информация по ссылке Тензоры в TensorFlow

Нейронны сети с Tensorflow

Совместимость с NumPy

Работа с массивами в TensorFlow хорошо совместима с NumPy `ndarray`:

TensorFlow операции автоматически конвертируют NumPy массивы в тензоры.

NumPy операции автоматически конвертируют Tensors в NumPy массивы.

```
print('Один и тот же тензор, преобразованный в Numpy разными методами')
print(np.array(rank_3_tensor))
print('\n')
print(rank_3_tensor.numpy())
```

В итоге получатся одинаковые массивы с типом `ndarray`. Также недавно в библиотеке версии 2.4 добавили новые возможности для работы с Numpy, по ссылке дополнительная информация NumPy API на TensorFlow.

Формы тензоров

Одной из важных тем для работ с тензорами является понимание и отслеживание формы тензоров. Для этого есть несколько методов и функций.

Тензоры, как и Numpy массивы, имеют форму. Вот некоторые термины:

- **Shape**: количество элементов по каждому измерению тензора
- **Rank**: номер измерения тензора. Для числа rank = 0, для вектора rank = 1, а для матрицы rank = 2.
- **Axis** или **Dimension**: измерения тензоров
- **Size**: общее число элементов в тензоре. Вектор формы

Нейронны сети с Tensorflow

```
print(“Тензор:”, rank_3_tensor, “\n”)
Тензор: tf.Tensor(
[[[2 1]
  [2 1]]

 [[3 1]
  [3 1]]

 [[4 1]
  [4 1]]], shape=(3, 2, 2), dtype=int32)
print(“Число измерений:”, rank_3_tensor.ndim)
Число измерений: 3
print(“Тип каждого элемента:”, rank_3_tensor.dtype)
Тип каждого элемента: <dtype: ‘int32’>
print(“Форма тензора:”, rank_3_tensor.shape)
Форма тензора: (3, 2, 2)
print(“Элементы тензора на оси с индексом -1:”, rank_3_
tensor[0])
Элементы тензора на оси с индексом -1: tf.Tensor(
[[2 1]
 [2 1]], shape=(2, 2), dtype=int32)
print(“Элементы тензора на оси с индексом -1:”, rank_3_
tensor[-1])
Элементы тензора на оси с индексом -1: tf.Tensor(
[[4 1]
 [4 1]], shape=(2, 2), dtype=int32)
```

Тип данных `tf.Variable`

Tf.Variable представляет тензор, значение которого можно изменить, запустив на нем операции. Определенные операции позволяют вам читать и изменять значения этого тензора. Библиотеки более высокого уровня, такие как tf.keras, используют tf.Variable для хранения параметров модели.

Нейронны сети с Tensorflow

```
# Shape вернет форму тензора
var_x = tf.Variable(tf.constant([[1], [2], [3]]))
print(var_x.shape)
>>>(3, 1)
# Можно конвертировать форму в Python лист
print(var_x.shape.as_list())
>>>[3, 1]
#Имеется возможность поменять форму тензора
reshaped = tf.reshape(var_x, [1, 3])
print(var_x.shape)
>>>(3, 1)
print(reshaped.shape)
>>>(1, 3)
```

Broadcasting

Broadcasting (пакетные вычисления) - это полный набор операций над тензорами, заимствованный из NumPy, с поддержкой матричных вычислений над массивами разной формы и конвертирования между этими формами.

Самый простой и наиболее распространенный случай, когда вы пытаетесь умножить или добавить тензор в скаляр. В этом случае скаляр передается в той же форме, что и другой аргумент.

```
x = tf.constant([1, 2, 3])
y = tf.constant(2)
z = tf.constant([2, 2, 2])
# Разные методы одного и того же вычисления
print(tf.multiply(x, 2))
print(x * y)
print(x * z)

tf.Tensor([2 4 6], shape=(3,), dtype=int32)
tf.Tensor([2 4 6], shape=(3,), dtype=int32)
tf.Tensor([2 4 6], shape=(3,), dtype=int32)
```

Аналогично одномерные могут быть растянуты, чтобы соответствовать другим аргументам. Оба аргумента могут быть растянуты в одном и том же вычислении.

Нейронны сети с Tensorflow

Например, матрица 3x1, добавленная к матрице 1x3, становится добавлением двух матриц 3x3.

В большинстве случаев broadcasting является эффективным с точки зрения времени и пространства, поскольку эта операция никогда не материализует расширенные тензоры в памяти.
В отличие от математической операции, broadcast_to не делает ничего особенного для экономии памяти. Здесь вы материализуете тензор.

В этом разделе книги Джейка ВандерПласа “Python Data Science Handbook” показаны другие приемы broadcasting (опять же в NumPy).

Оборванный тензор

Тензор с переменным числом элементов в измерении называется «оборванным». Используйте tf.ragged.RaggedTensor для оборванных данных.

```
valid_tensor = tf.constant([[1, 2, 0], [2, 3, 4]])
print(valid_tensor)
tf.Tensor(
[[1 2 0]
 [2 3 4]], shape=(2, 3), dtype=int32)

# Здесь размеры неравномерны по 0-ой оси
# not_a_valid_base_tensor = tf.constant ([[1,2],
[2,3,4]])
# Вместо этого используйте `RaggedTensor`
ragged_tensor = tf.ragged.constant([[1, 2], [2, 3, 4]])
print(ragged_tensor)
<tf.RaggedTensor [[1, 2], [2, 3, 4]]>
```

Нейронны сети с Tensorflow

Строковый тензор

Как и NumPy, tf.string является dtype, то есть мы можем представлять данные в виде строк (байтовых массивов переменной длины) в тензорных числах.

Строки являются атомарными и не могут быть проиндексированы, как строки Python. Длина строки не является одним из измерений тензора. Смотрите tf.strings для функций, чтобы управлять ими. Некоторые основные функции со строками можно найти в tf.strings, включая tf.strings.split.

Хотя вы не можете использовать tf.cast, чтобы превратить строковый тензор в числовой, вы можете преобразовать его в байты, а затем в числа.

Тип tf.string dtype используется для всех необработанных байтовых данных в TensorFlow. Модуль tf.io содержит функции для преобразования данных в байты и из них, включая декодирование изображений и синтаксический анализ CSV.

GPU и размещение на устройстве

Проверка доступных списков устройств и размещение данных на конкретном модуле.

```
 print(“Доступные модули: “),
 print(tf.config.experimental.list_physical_devices())
 Доступные модули:
 [PhysicalDevice(name=’/physical_device:CPU:0’, device_
type=’CPU’), PhysicalDevice(name=’/physical_device:XLA_
CPU:0’, device_type=’XLA_CPU’), PhysicalDevice(name=’/
physical_device:XLA_GPU:0’, device_type=’XLA_GPU’),
PhysicalDevice(name=’/physical_device:GPU:0’, device_
type=’GPU’)]
 print(“Доступные модули GPU: “),
 print(tf.config.experimental.list_physical_devices(“GPU”))
 Доступные модули GPU:
 [PhysicalDevice(name=’/physical_device:GPU:0’, device_
type=’GPU’)]
```

Нейронны сети с Tensorflow

```
x = tf.random.uniform([3, 3])
print("Прогоняется ли тензор на GPU #0 ?:  ")
print(x.device.endswith('GPU:0'))
Прогоняется ли тензор на GPU #0 ?:
True
```

Датасеты

Раздел tf.data.Dataset API предлагает построить готовый конвейер для обучения моделей:

- tf.data.Dataset.from_tensor_slices() - данные из массивов
- tf.data.Dataset.from_generator() - данные через генераторы
- tf.data.TFRecordDataset() - специальный формат данных для ускорения работы
- tf.data.TextLineDataset() - данные из текстовых файлов
- tf.data.Dataset.list_files() - список файлов с данными

Примеры создания данных описаны в ноутбуке занятия.

Создание и обучение модели

Решение проблемы машинного обучения обычно включает следующие шаги:

1.Получите данные для обучения.

2. Определите модель.
3. Определите функцию потерь.
4. Просмотрите данные обучения, вычисляя потери от идеального значения.
5. Вычислите градиенты для этой потери и используйте оптимизатор, чтобы настроить переменные в соответствии с данными.
6. Оцените свои результаты.

Все эти шаги реализованы в коде в ноутбуке занятия.