GPU vs CPU

Если вы когда-нибудь пробовали собирать компьютер, то наверняка знаете, что внутри него находится множество компонентов. «Сердцем» любой вычислительной машины является центральный процессор (CPU) — небольшой чип, обычно спрятанный под охлаждающим вентилятором в центре материнской платы. Графический процессор (GPU) отвечает за обработку изображений. Иногда графическое ядро может быть интегрировано в CPU, поэтому не совсем корректно ставить знак равенства между GPU и видеокартой.

Если вы используете компьютер для запуска игр или тяжёлых программ, то, скорее всего, в нём есть дискретная видеокарта с GPU. Она занимает куда больше места и обычно требует отдельного охлаждения. Так как же все эти вещи используются в машинном обучении?

Рис. 1 - Компьютер изнутри

Одна из главных тем споров в IT-сфере — предпочтение оборудования или ПО того или иного производителя. Наверняка вы сталкивались с дебатами «текстовый редактор Vim против Emacs», «процессоры Intel против AMD»,

«видеокарты NVIDIA против Radeon». Изначально GPU создавались для рендеринга компьютерной графики, поэтому те, кто часто играет в компьютерные игры, наверняка могут высказать своё мнение о том или ином производителе графических карт. Мы в основном будем фокусироваться на GPU от NVIDIA ввиду их большей распространённости в области машинного обучения.

В чём же разница между CPU и GPU? И то, и другое — вычислительные единицы, выполняющие программы и произвольные инструкции. Но, тем не менее, они существенно отличаются друг от друга.

CPU состоят из небольшого числа ядер, в настоящее время обычно не превышающего десяти. Технология гиперпоточности (hyperthreading) позволяет каждому ядру работать в несколько потоков. Упрощённо говоря, CPU с 10-ю ядрами и 20-ю потоками может выполнять 20 задач одновременно. Это не так много, но на самом деле потоки CPU очень мощные, и одна их инструкция способна включать множество процессов.

С GPU дела обстоят иначе. Они содержат в себе тысячи вычислительных ядер, но каждое из них работает на гораздо меньшей тактовой частоте, чем CPU (тактовая частота показывает, сколько операций в секунду выполняет процессор). Ядра GPU не могут работать независимо друг от друга, в отличие от CPU. Они распараллеливают задачи и делают их одновременно. Поэтому сравнивать производительность GPU и CPU по числу ядер не имеет смысла — они предназначены для совершенно разных целей.

Новое поколение GPU выходит примерно раз в полтора года. Поэтому опытные исследователи активно переходят в облако и постоянно имеют доступ к новейшим графическим ускорителям. За их обновление, администрирование и поддержку отвечает провайдер, а клиенты могут сэкономить приличные суммы. Например, использование сервера с топовой на сегодняшний день GPU NVIDIA Tesla V100 в REG.RU выйдет дешевле, чем покупка аналогичной видеокарты в магазине, особенно если она нужна вам на ограниченный срок.

Ещё одно существенное отличие между CPU и GPU — использование памяти. CPU в основном расходует оперативную память системы (ОЗУ или RAM), размер которой достигает нескольких десятков гигабайт. GPU имеет собственную встроенную видеопамять (VRAM). В графическом ядре запись и чтение из памяти осуществляются последовательно — например, при обработке пикселей они будут считываться друг за другом, в то время как в CPU доступ к памяти организован более сложным образом.

GPU хорошо справляется с вычислениями, которые можно разбить на множество одновременно выполняющихся задач: например, обработка изображений или умножение матриц. CPU используется в самых разнообразных мощных процессах, но сильно проигрывает GPU в распараллеливании.

Как мы помним, в свёрточных архитектурах умножение матриц происходит постоянно, поэтому GPU — незаменимый инструмент для обучения нейросетей.

Для того, чтобы запускать код прямо на GPU, NVIDIA разработала ускорители CUDA. Написание CUDA-кода — достаточно сложный процесс, требующий глубокого погружения в архитектуру графических ускорителей. Поэтому для удобства можно использовать более высокоуровневые библиотеки: cuBLAS, cuFFT, cuDNN и другие. Также существует фреймворк openCL, оптимизированный для любых GPU (даже от AMD). Но он, как правило, показывает более медленные результаты.

Для задач глубокого обучения обычно достаточно готовых библиотек, поскольку написавшие их разработчики постарались максимально оптимизировать и упростить большинство необходимых операций.

Фреймворки для глубокого обучения

Число различных программ и библиотек для глубокого обучения растёт с каждым годом. К самым известным относятся TensorFlow, Caffe, PyTorch, также развиваются Paddle от Baidu, CNTK от Microsoft, MXNet от Amazon и

многие другие. И хотя каждая библиотека содержит свои отличительные функции, некоторые особенности присутствуют во всех фреймворках:

— в них легко строить большие вычислительные графы;

— легко вычислять градиенты для вычислительных графов;

— они эффективно используют GPU.

Варианты серверов с GPU NVIDIA

Тариф	Видеокарта	Диск	Процессор	Память	Цена за час	
GPU-1	Tesla V100 16GB	60 ГБ	2 ядра	4 ГБ	90 ₽/час	Заказать сервер
GPU-2	Tesla V100 16GB	80 ГБ	2 ядра	8 ГБ	100 ₽/час	Заказать сервер
GPU-3	Tesla V100 16GB	120 ГБ	2 ядра	16 ГБ	110 ₽/час	Заказать сервер
GPU-4	Tesla V100 16GB	150 ГБ	4 ядра	32 ГБ	120 ₽/час	Заказать сервер
GPU-5	Tesla V100 32GB	150 ГБ	2 ядра	32 ГБ	130 ₽/час	Заказать сервер

По этим причинам применять готовые библиотеки чаще всего гораздо продуктивнее и удобнее, чем писать собственный код. Попробуем заглянуть внутрь приложений и подробнее рассмотрим, для каких задач подходят фреймворки TensorFlow, PyTorch и Caffe.

TensorFlow

В качестве последующих примеров будем использовать двухслойную полносвязную нейросеть с функцией активации ReLU. Мы попробуем обучить её на случайных данных и вычислить потери L2. На самом деле наша нейросеть не будет делать ничего полезного, но её код поможет вам узнать о некоторых важных функциях TensorFlow.

Все вычислительные операции в TensorFlow проводятся в два больших этапа. Сначала необходимо определить вычислительный граф, а затем запустить многократный проход по нему с целью выполнения каких-либо действий с данными.

```
1	import numpy as np
2	import tensorflow as tf
3	
4	# Определяем наш вычислительный граф
5	N, D, H = 64, 1000, 100
6	# Входные узлы графа
7	x = tf.placeholder(tf.float32, shape = (N, D))
8	y = tf.placeholder(tf.float32, shape = (N, D))
9	w1 = tf.placeholder(tf.float32, shape = (D, H))
10	w2 = tf.placeholder(tf.float32, shape = (H, D))
11	
12	# Прямой проход: вычисляем прогнозируемые значения и
13	потери
14	h = tf.maximum(tf.matmul(x, w1), 0)
15	y_pred = tf.matmul(h, w2)
16	diff = y_pred - y
17	loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
18	
19	# Считаем потери градиентов
20	grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
21	
22	# Запускаем граф множество раз
23	with tf.Session() as sess:
24	    values = {x: np.random.randn(N, D),
25	              w1: np.random.randn(D, H),
26	              w2: np.random.randn(H, D),
27	              y: np.random.randn(N, D),}
```

```
28          out = sess.run([loss, grad_w1, grad_w2],
29                         feed_dict=values)
            loss_val, grad_w1_val, grad_w2_val = out
```

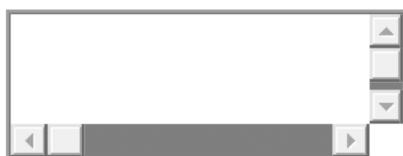
Объекты **tf.placeholder** используются для передачи входных данных в вычислительный граф, а метод **tf.maximum** вводит нелинейность ReLU.

Сначала мы выполняем матричное умножение переменных **x** и **w1** (данных и параметров), а затем вычисляем потери L2 между прогнозируемыми (**y_pred**) и истинными (**y**) значениями с помощью базовых тензорных операций. Как вы могли заметить, у нас пока нет никаких данных и на самом деле этот код ничего не делает.

Вычисление градиентных потерь выполняется с помощью всего одной магической строки. Это избавляет от необходимости писать собственный код для обратного прохода по графу.

При запуске графа мы генерируем данные с помощью **np.random**, чтобы передать конкретные значения в ранее созданные placeholders. Вызов метода **tf.Session().run** начинает выполнение вычислений. В качестве первого аргумента мы указываем параметры, которые хотим посчитать: потери **loss** и градиенты **grad_w1**, **grad_w2**. Второй аргумент **feed_dict** содержит в себе передаваемые в граф данные. После выполнения этой строки TensorFlow запустит граф и вычислит все необходимые значения.

Код пока не обучает нейросеть: для этого достаточно добавить всего несколько строк с реализацией градиентного спуска и обновлением весов:



```
1       loss_arr = []
2
3       # Запускаем граф множество раз
4       with tf.Session() as sess:
5           values = {x: np.random.randn(N, D),
```

```
6                       w1: np.random.randn(D, H),
7                       w2: np.random.randn(H, D),
8                       y: np.random.randn(N, D),}
9               learning_rate = 1e-5
10              for t in range(50):
11                  out = sess.run([loss, grad_w1, grad_w2],
12                              feed_dict=values)
13                  loss_val, grad_w1_val, grad_w2_val = out
14                  loss_arr.append(loss_val)
15                  values[w1] -= learning_rate * grad_w1_val
16                  values[w2] -= learning_rate * grad_w2_val
17
18          import matplotlib.pyplot as plt
19
20          x = [c for c in range(0, 50)]
21          _, ax = plt.subplots()
22          ax.scatter(x, loss_arr)
```

Построив график потерь, мы увидим, что нейросеть действительно обучается и потери уменьшаются:

Но в этом примере есть небольшая загвоздка. Дело в том, что мы каждый раз передаём в граф массивы NumPy, вычисляем градиенты и возвращаем их значения обратно. Если запускать код на CPU, то в этом нет особой проблемы, но при использовании GPU нам придётся каждый раз копировать данные из памяти CPU в память GPU. Поэтому если нейросеть будет очень большой, это существенно замедлит процесс её обучения.

К счастью, в TensorFlow есть готовое решение проблемы. Вместо того, чтобы использовать веса в виде **tf.placeholder**, мы объявим их как **tf.Variable**. Variable — это значение, которое находится внутри вычислительного графа и сохраняется при каждом его запуске.

```
1    w1 = tf.Variable(tf.float32, shape = (D, H))
2    w2 = tf.Variable(tf.float32, shape = (H, D))
```

Теперь нам не нужно каждый раз обновлять веса и градиенты, поскольку они уже находятся в графе. Мы отправляем на вход только данные и метки, а

на выходе получаем потери. Чтобы обновление выполнялось внутри графа, добавим оптимизатор **optimizer** для вычисления градиентов и будем минимизировать их с помощью переменной **updates**, которая использует метод **minimize** и передаётся в **tf.Session**:

```
1       # Прямой проход: вычисляем прогнозируемые значения и
2   потери
3       h = tf.maximum(tf.matmul(x, w1), 0)
4       y_pred = tf.matmul(h, w2)
5       diff = y_pred - y
6       loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
7
8       optimizer = tf.train.GradientDescentOptimizer(1e-5)
9       updates = optimizer.minimize(loss)
10
11      # Считаем потери градиентов
12      grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
13
14      learning_rate = 1e-5
15      new_w1 = w1.assign(w1 - learning_rate * grad_w1)
16      new_w2 = w2.assign(w2 - learning_rate * grad_w2)
17
18      loss_arr = []
19
20      # Запускаем граф множество раз
21      with tf.Session() as sess:
22          sess.run (tf.global_variables_initializer())
23          values = {x: np.random.randn(N, D),
```

```
24                      y: np.random.randn(N, D),}
25
26              for t in range(50):
27                  out = sess.run([loss, grad_w1, grad_w2],
28                              feed_dict=values)
29                  loss_val, _ = sess.run([loss, updates], feed_dict=values)
                    loss_arr.append(loss_val)
```

Дополнив код выше и построив график, вы увидите, что потери уменьшаются — значит, обучение нейросети снова прошло успешно, и на этот раз нам не пришлось копировать данные из одной памяти в другую.

Несколько слов о Keras

Keras — высокоуровневое API, которое обычно используется «поверх» Tensorflow и позволяет создавать и обучать нейросети с помощью простых и понятных команд. Он отлично подойдёт для построения базовых моделей и знакомства с Machine Learning. Если вам не терпится попробовать написать код для своей первой по-настоящему рабочей нейросети, рекомендуем ознакомиться с нашей статьёй **Как начать работу с Keras, Deep Learning и Python**.

PyTorch

В PyThorch существует понятие трёх уровней абстракции. Для каждой из них можно найти аналогичные объекты в TensorFlow:

Мы не будем подробно останавливаться на деталях и сразу рассмотрим код двухслойной нейросети с оптимизатором Adam, реализацию которой мы использовали выше. В PyTorch он будет выглядеть примерно следующим образом:

```
1       import torch
2       from torch.autograd import Variable
```

```
3
4           N, D_in, H, D_out = 64, 1000, 100, 10
5           x = Variable(torch.randn(N, D_in))
6           y = Variable(torch.randn(N, D_out), requires_grad=False)
7
8           # Определяем последовательность слоёв модели
9           model = torch.nn.Sequential(
10                torch.nn.Linear(D_in, H),
11                torch.nn.ReLU(),
12                torch.nn.Linear(H, D_out))
13          # Функция потерь
14          loss_fn = torch.nn.MSELoss(size_average=False)
15
16          # Устанавливаем скорость обучения и добавляем оптимизатор
17 Adam
18          learning_rate = 1e-4
19          optimizer          =          torch.optim.Adam(model.parameters(),
20 lr=learning_rate)
21
22          for t in range(500):
23             # Прямой проход: вычисляем прогноз и потери
24             y_pred = model(x)
25             loss = loss_fn(y_pred, y)
26
27             # Обратный проход: считаем градиенты
28             model.zero_grad()
29             loss.backward()
30
31             optimizer.step()
32
```

```
33              # Обновляем параметры
                for param in model.parameters():
                  param.data -= learning_rate * param.grad.data
```

Модуль **nn**, как вы могли догадаться, содержит в себе готовые функции для работы с нейросетями. Помимо него в библиотеке есть множество полезных инструментов: например, метод **DataLoader** позволяет удобно импортировать наборы данных, разбивать их на мини-пакеты, перемешивать и использовать собственные классы датасетов. А с дополнительной утилитой TorchVision можно быстро загружать предварительно обученные популярные модели: AlexNet, VGG16, ResNet и другие.

Совершить подробную и занимательную экскурсию по PyTorch можно, прочитав эту статью.

Caffe

Caffe отличается от других фреймворком тем, что в нём для обучения моделей иногда даже не нужно писать код. Можно просто взять существующие исходники, добавить в них кое-какие настройки и загрузить собственные данные, написав для этого несколько инструкций в текстовом файле специального формата **.prototxt**. Правда, обычно это не очень хорошо работает для больших архитектур. Caffe редко применяется в серьёзных исследованиях, но довольно широко распространён в практическом машинном обучении.

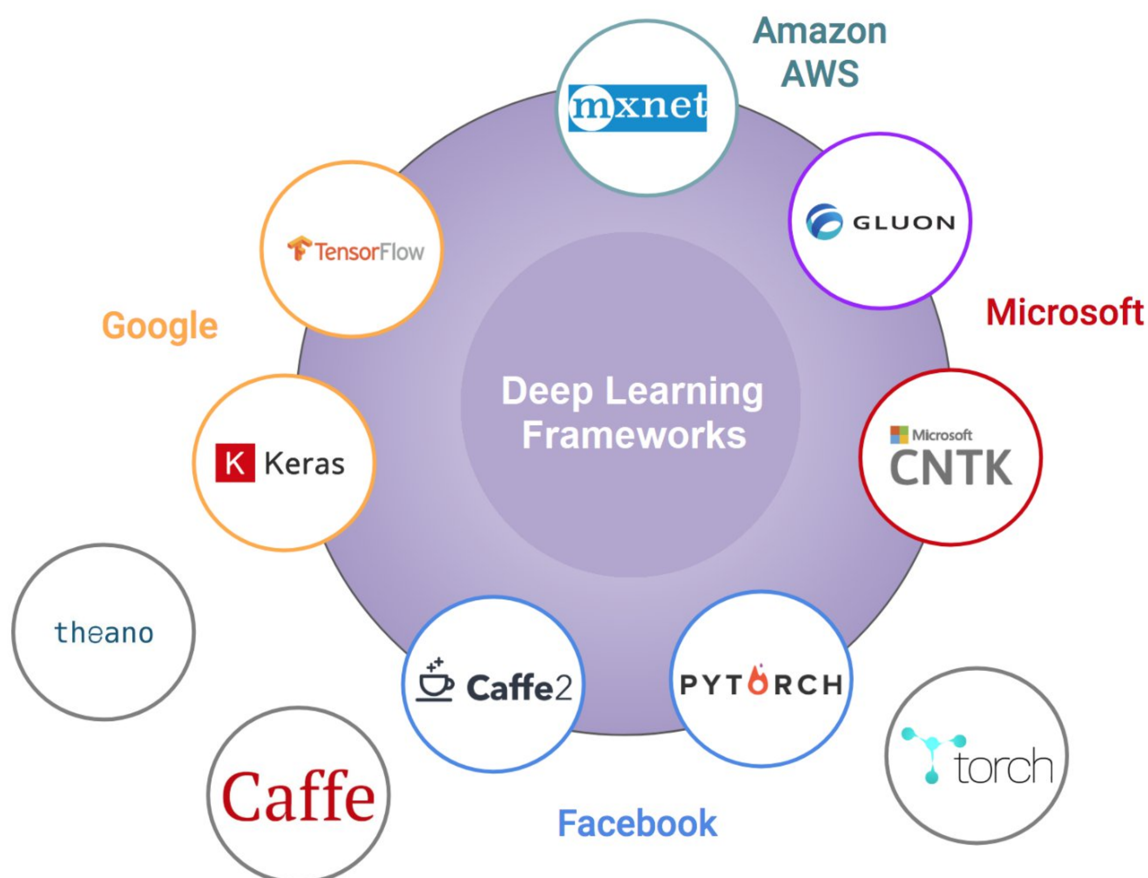
Весь алгоритм работы с Caffe можно охарактеризовать следующими шагами:

1. Сконвертируйте данные в формат HDF5 или LMDB с помощью готовых скриптов;
2. Задайте настройки для нейросети (отредактируйте prototxt);
3. Настройте solver (оптимизацию) (отредактируйте prototxt);
4. Запустите обучение (готовый скрипт).

```
1	./build/tools/caffe train \
2	 -gpu 0 \
3	 -model path/to/trainval.prototxt \
4	 -solver path/to/solver.prototxt \
5	 -weights path/to/pretrained_weights.caffemodel
```

Caffe доступен и в Python. Лучше всего использовать новую версию Caffe2, которая интегрирована в PyTorch: она хорошо подходит для работы с массивами NumPy, извлечения признаков из данных, настройки и обучения моделей, а также обеспечивает поддержку мобильных платформ iOS, Android и других.

Что же использовать?



На наш взгляд, **TensorFlow** — наилучший вариант для большинства проектов. Он не идеален, но имеет огромное сообщество и очень широко применяется в самых разных сферах. К тому же, его можно использовать с более высокоуровневой оболочкой (Keras, Sonnet и другие).

PyTorch лучше всего подходит для исследований, а **Caffe2** — для развёртывания готовых моделей и их адаптации на мобильных устройствах.