

PyTorch

Задание: Сделать *любой* тестовый пример использования PyTorch с базовым перцептроном. В качестве базы можно брать *любые* примеры и инструкции из интернета и инструкции изложенные в этом занятии. Результат необходимо представить в виде notebook или репозитория/папки и залить в свой репозиторий с заданиями. Если вы используете нестандартные зависимости, то указывайте их в документации.

Библиотека PyTorch предназначена для глубокого обучения. Глубокое обучение - это просто другое название крупномасштабной нейронной сети или сети многослойных перцептронов. В своей простейшей форме многослойные перцептроны представляют собой последовательность слоев, соединенных в тандем. В этом посте вы узнаете о простых компонентах, которые можно использовать для создания нейронных сетей и простых моделей глубокого обучения в PyTorch.

Начать свой проект в PyTorch можно с книги [Deep Learning with PyTorch](#). В ней вы найдете **самоучители с рабочим кодом**.

Изображение для примера использования, с которого можно начать.



Общее описание

Общее содержание - общие понятия

- Нейросетевые модели в PyTorch
- Входные данные модели

- Слои, активации и свойства слоев
- Функции потерь и оптимизаторы модели
- Обучение модели
- Исследование модели

Нейросетевые модели в PyTorch

PyTorch может делать множество вещей, но самый распространенный вариант использования - построение модели глубокого обучения. Простейшая модель может быть определена с помощью класса `Sequential`, который представляет собой линейный стек слоев, соединенных в тандем. Вы можете создать модель `Sequential` и определить все слои в один "заход";

Например:

```
import torch
import torch.nn as nn

model = nn.Sequential(...)
```

Все слои должны быть определены внутри круглых скобок в порядке обработки от входа к выходу.

Например:

```
model = nn.Sequential(
    nn.Linear(764, 100),
    nn.ReLU(),
    nn.Linear(100, 50),
    nn.ReLU(),
    nn.Linear(50, 10),
    nn.Sigmoid()
)
```

Другой способ использования `Sequential` - передать упорядоченный словарь, в котором вы можете присвоить имена каждому слою:

```
from collections import OrderedDict
import torch.nn as nn

model = nn.Sequential(OrderedDict([
    ('dense1', nn.Linear(764, 100)),
    ('act1', nn.ReLU()),
    ('dense2', nn.Linear(100, 50)),
    ('act2', nn.ReLU()),
    ('output', nn.Linear(50, 10)),
    ('outact', nn.Sigmoid()),
]))
```

Если вы хотите создавать слои по одному, а не делать все в один подход, вы можете сделать следующее:

```
model = nn.Sequential()
model.add_module("dense1", nn.Linear(8, 12))
model.add_module("act1", nn.ReLU())
model.add_module("dense2", nn.Linear(12, 8))
model.add_module("act2", nn.ReLU())
model.add_module("output", nn.Linear(8, 1))
model.add_module("outact", nn.Sigmoid())
```

Это пригодится вам в более сложных случаях, когда нужно построить модель на основе некоторых условий.

Входные данные модели

Первый слой в вашей модели намекает на форму входных данных. В примере выше в качестве первого слоя используется `nn.Linear(764, 100)`. В зависимости от типа слоя, который вы используете, аргументы могут иметь разное значение. Но в данном случае это `Linear` слой (также известный как плотный слой или слой с полным подключением), а два аргумента указывают размеры входа и выхода **этого слоя**.

Обратите внимание, что размер пакета является неявным. В данном примере вы должны передать в этот слой тензор PyTorch формы `(n, 764)` и ожидать в ответ тензор формы `(n, 100)`, где `n` - размер пакета.

Слои, активации и свойства слоев

В PyTorch существует множество видов слоев нейронных сетей. На самом деле, при желании можно легко определить свой собственный слой. Ниже приведены некоторые распространенные слои, которые вы можете часто встречать:

- `nn.Linear(input, output)`: Полностью связанный слой
- `nn.Conv2d(in_channel, out_channel, kernel_size)`: Слой двумерной свертки, популярный в сетях обработки изображений
- `nn.Dropout(probability)`: Слой выпадения, обычно добавляется в сеть для введения регуляризации
- `nn.Flatten()`: Преобразование тензора входных данных высокой размерности в одномерный (для каждого образца в партии).

Помимо слоев, существуют также функции активации. Это функции, применяемые к каждому элементу тензора. Обычно вы берете выход одного слоя и применяете активацию перед тем, как подать его на вход последующего слоя. Некоторые распространенные функции активации:

- `nn.ReLU()`: Выпрямленная линейная единица, наиболее распространенная активация в настоящее время
- `nn.Sigmoid()` и `nn.Tanh()`: Сигмоидная и гиперболическая касательные функции

- `nn.Softmax()`: Преобразование вектора в вероятностно-подобные значения; популярна в сетях классификации

Список всех различных слоев и функций активации можно найти в документации PyTorch.

Конструкция PyTorch очень модульная. Поэтому вам не придется много настраивать в каждом компоненте. Возьмем для примера этот слой `Linear`. Вы можете указать только форму входа и выхода, но не другие детали, например, как инициализировать веса. Однако почти все компоненты могут принимать два дополнительных аргумента: устройство и тип данных.

Устройство PyTorch определяет, где будет выполняться данный слой. Обычно вы выбираете между CPU и GPU или опускаете этот параметр и предоставляете PyTorch самому решать. Чтобы указать устройство, выполните следующие действия (CUDA означает поддерживаемый графический процессор nVidia):

```
nn.Linear(764, 100, device="cpu")
```

или

```
nn.Linear(764, 100, device="cuda:0")
```

Аргумент типа данных (`dtype`) указывает, с каким типом данных должен работать этот слой. Обычно это 32-битный float, и обычно вы не хотите его менять. Но если вам нужно указать другой тип, вы должны сделать это с помощью типов PyTorch, например,

```
nn.Linear(764, 100, dtype=torch.float16)
```

Функция потерь и оптимизаторы модели

Модель нейронной сети представляет собой последовательность матричных операций. Матрицы, которые не зависят от входных данных и хранятся внутри модели, называются весами. При обучении нейронной сети эти веса **оптимизируются** таким образом, чтобы они давали нужный вам выход. В глубоком обучении алгоритмом оптимизации этих весов является градиентный спуск.

Существует множество вариаций градиентного спуска. Вы можете сделать свой выбор, подготовив оптимизатор для своей модели. Он не является частью модели, но вы будете использовать его вместе с моделью во время обучения. Способ его использования включает в себя определение **функции потерь** и минимизацию функции потерь с помощью оптимизатора. Функция потерь выдает **оценку расстояния**, которая показывает, насколько далеко выход модели находится от желаемого выхода. Она сравнивает выходной тензор модели с ожидаемым тензором, который в другом контексте называется **ярлыком** или **земной истиной**. Поскольку она предоставляется как часть обучающего набора данных, модель нейронной сети является моделью обучения под наблюдением.

В PyTorch вы можете просто взять выходной тензор модели и манипулировать им для вычисления потерь. Но вы также можете использовать для этого функции, предоставляемые в PyTorch, например,

```
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(output, label)
```

В этом примере `loss_fn` - функция, а `loss` - тензор, поддерживающий автоматическое дифференцирование. Вы можете запустить дифференцирование, вызвав `loss.backward()`.

Ниже приведены некоторые распространенные функции потерь в PyTorch:

- `nn.MSELoss()`: Средняя квадратичная ошибка, полезная в задачах регрессии
- `nn.CrossEntropyLoss()`: Потеря перекрестной энтропии, полезная в задачах классификации
- `nn.BCELoss()`: Двоичные потери перекрестной энтропии, полезные в задачах двоичной классификации

Создание оптимизатора аналогично:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Все оптимизаторы требуют список всех параметров, которые им необходимо оптимизировать. Это связано с тем, что оптимизатор создается вне модели, и вам нужно указать ему, где искать параметры (т. е. веса модели). Затем оптимизатор возьмет градиент, вычисленный вызовом функции `backward()`, и применит его к параметрам в соответствии с алгоритмом оптимизации.

Вот список некоторых распространенных оптимизаторов:

- `torch.optim.Adam()`: Алгоритм Adam (адаптивная оценка моментов)
- `torch.optim.NAdam()`: Алгоритм Адама с нестеровским моментом
- `torch.optim.SGD()`: Стохастический градиентный спуск
- `torch.optim.RMSprop()`: Алгоритм RMSprop

Список всех предоставляемых функций потерь и оптимизаторов можно найти в документации PyTorch. О математической формуле каждого алгоритма оптимизации можно узнать на странице соответствующего оптимизатора в документации.

Обучение и вывод модели

В PyTorch нет специальной функции для обучения и оценки моделей. Определенная модель сама по себе похожа на функцию. Вы передаете на вход тензор и получаете на выходе тензор. Поэтому на вас лежит ответственность за написание цикла обучения. Минимальный цикл обучения выглядит следующим образом:

```
for n in range(num_epochs):
    y_pred = model(X)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```


Если у вас уже есть модель, вы можете просто взять `y_pred = model(X)` и использовать выходной тензор `y_pred` для других целей. Так вы используете модель для предсказания или вывода. Однако модель ожидает не один входной образец, а множество входных образцов в одном тензоре. Если модель должна принимать входной вектор (который является одномерным), вы должны предоставить ей двумерный тензор. Как правило, в случае с выводами вы намеренно создаете партию из одной выборки.

Исследование модели

Когда у вас есть модель, вы можете проверить, что она собой представляет:

```
print(model)
```

Это даст вам, например, следующее:

```
Sequential(
  (0): Linear(in_features=8, out_features=12, bias=True)
  (1): ReLU()
  (2): Linear(in_features=12, out_features=8, bias=True)
  (3): ReLU()
  (4): Linear(in_features=8, out_features=1, bias=True)
  (5): Sigmoid()
)
```

Если вы хотите сохранить модель, вы можете использовать библиотеку `pickle` из Python. Но вы также можете получить к ней доступ с помощью PyTorch:

```
torch.save(model, "my_model.pickle")
```

Таким образом, весь объект модели сохраняется в файле `pickle`. Вы можете извлечь модель с помощью:

```
model = torch.load("my_model.pickle")
```

Но рекомендуемый способ сохранения модели - оставить дизайн модели в коде и сохранить только веса. Это можно сделать с помощью:

```
torch.save(model.state_dict(), "my_model.pickle")
```

Функция `state_dict()` извлекает только состояния (т.е. веса в модели). Чтобы получить ее, вам нужно перестроить модель с нуля, а затем загрузить веса следующим образом:

```
model = nn.Sequential(...)
model.load_state_dict(torch.load("my_model.pickle"))
```

Ресурсы

Онлайн

- [torch.nn documentation](#)
- [torch.optim documentation](#)
- [PyTorch tutorials](#)

Как построить LSTM-модель с помощью Pytorch

Длительная кратковременная память (LSTM - Long short term memory) - это особый вид RNN (Recurrent Neural Network). Они доказали свою эффективность при решении различных задач в этой области, таких как распознавание речи, машинный перевод и другие. Они преодолевают ограничения наивных RNN, которые не справляются с долгосрочными зависимостями в последовательностях.

Имплементация модели LSTM в PyTorch

Мы можем построить LSTM-модель с помощью PyTorch, выполнив следующие шаги:

Шаг 1

Во-первых, мы импортируем библиотеку PyTorch в наш проект с помощью следующего фрагмента кода:

```
import torch
import torch.nn as nn
```

Шаг 2

Далее нужно подготовить и загрузить набор данных в проект.

Шаг 3

Теперь мы переходим к созданию модели LSTM и определению прямого прохода LSTM. Например:

```
class LSTMModel(nn.Module):
    def __init__(self, input_d, hidden_d, layer_d, output_d):
        super(LSTMModel, self).__init__()

        self.hidden_dim = hidden_d
        self.layer_dim = layer_d

        # LSTM model
        self.lstm = nn.LSTM(input_d, hidden_d, layer_d, batch_first=True)
```

```

        # batch_first=True (batch_dim, seq_dim, feature_dim)

        self.fc = nn.Linear(hidden_d, output_d)

    def forward(self, x):

        h0 = torch.zeros(self.layer_dim, x.size(0),
self.hidden_dim).requires_grad_()

        c0 = torch.zeros(self.layer_dim, x.size(0),
self.hidden_dim).requires_grad_()

        out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))

        out = self.fc(out[:, -1, :])
        return out

input_dim = 30
hidden_dim = 120
output_dim = 15
layer_dim = 1

model = LSTMModel(input_dim, hidden_dim, layer_dim, output_dim)

```

Пояснение кода выше

Строка 1: Мы наследуем `nn.Module` в классе **LSTM**.

Строка 2: `input_d` - количество ожидаемых признаков на входе. `hidden_d` - количество признаков в скрытом состоянии.

Строки 5-6: Определяем количество скрытых измерений и слоев.

Строка 12: мы определяем слой считывания с помощью функции `fc()`.

Строка 14: мы определяем функцию `forward` для создания прямого прохода для модели **LSTM**.

Строка 16: Мы инициализируем скрытое состояние нулями при помощи `zeros`.

Строка 18: Мы инициализируем состояние ячеек нулями при помощи `zeros`.

Строка 20: Этот шаг выполняется тридцать раз. Мы отсоединяемся (`detach`), когда усекаем (`truncate`) Backpropagation through time (BPTT - Градиентная методика обучения некоторых типов RNN). Если мы не производим `truncate`, то вернемся к началу.

Строки 25-30: Объявляются переменные и создается объект модели LSTM.

Шаг 4

После инстанцирования модели мы инстанцируем потери (instantiation), вычисляя кросс-энтропийные потери. Следующий фрагмент кода демонстрирует это:


```
error = nn.CrossEntropyLoss()
```

Шаг 5

Далее мы инстанцируем оптимизатор с помощью SGD-оптимизатора:

Шаг 6

Модель обучается и используется для прогнозирования.

Общее решение

```
# шаг 1: импорт библиотек
import torch
import torch.nn as nn

# шаг 3: создание модели
class LSTMModel(nn.Module):
    def __init__(self, input_d, hidden_d, layer_d, output_d):
        super(LSTMModel, self).__init__()

        self.hidden_dim = hidden_d
        self.layer_dim = layer_d

        # LSTM модель
        self.lstm = nn.LSTM(input_d, hidden_d, layer_d, batch_first=True)

        self.fc = nn.Linear(hidden_d, output_d)

    def forward(self, x):

        h0 = torch.zeros(self.layer_dim, x.size(0),
self.hidden_dim).requires_grad_()

        c0 = torch.zeros(self.layer_dim, x.size(0),
self.hidden_dim).requires_grad_()

        out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))

        out = self.fc(out[:, -1, :])
        return out

input_dim = 30
hidden_dim = 120
output_dim = 15
layer_dim = 1

model = LSTMModel(input_dim, hidden_dim, layer_dim, output_dim)
```

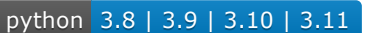
```
# шаг 4: расчет перекрестных энтропийных потерь (cross entropy loss)
error = nn.CrossEntropyLoss()

# шаг 5: оптимизация
learning_rate = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

PyTorch YOLO

Минимальная PyTorch-реализация YOLOv3, с поддержкой обучения, вывода и оценки.

Весы YOLOv4 и YOLOv7 также совместимы с этой реализацией.

Установка

Установка из исходного кода

Для обычного обучения и оценки мы рекомендуем устанавливать пакет из исходного кода, используя виртуальное окружение poetry.

```
git clone https://github.com/eriklindernoren/PyTorch-YOLOv3
cd PyTorch-YOLOv3/
pip3 install poetry --user
poetry install
```

Вам нужно присоединиться к виртуальной среде, запустив `poetry shell` в этой директории, прежде чем выполнять любую из следующих команд без префикса `poetry run`.

Также посмотрите на другой метод установки, если вы хотите использовать команды везде, не открывая poetry-shell.

Загрузка предварительно обученных весов

```
./weights/download_weights.sh
```

Загрузка COCO

```
./
```

Установка через pip

Этот способ установки рекомендуется, если вы хотите использовать этот пакет в качестве зависимости в другом python-проекте.

Этот метод включает только код, менее изолирован и может конфликтовать с другими пакетами.

Веса и набор данных COCO должны быть загружены, как указано выше.

Он также позволяет использовать инструменты CLI `yolo-detect`, `yolo-train` и `yolo-test` повсеместно без каких-либо дополнительных команд.

```
pip3 install pytorchyolo --user
```

Тест

Оценивает модель на тестовом наборе данных COCO.

Чтобы загрузить этот набор данных, а также веса, см. выше.

```
poetry run yolo-test --weights weights/yolov3.weights
```

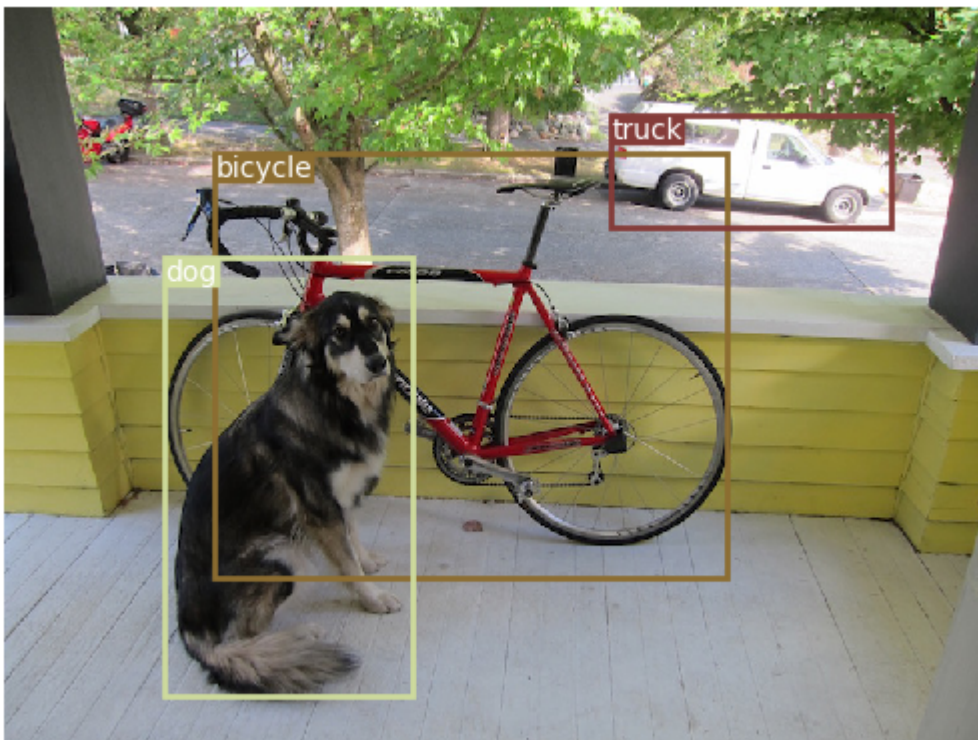
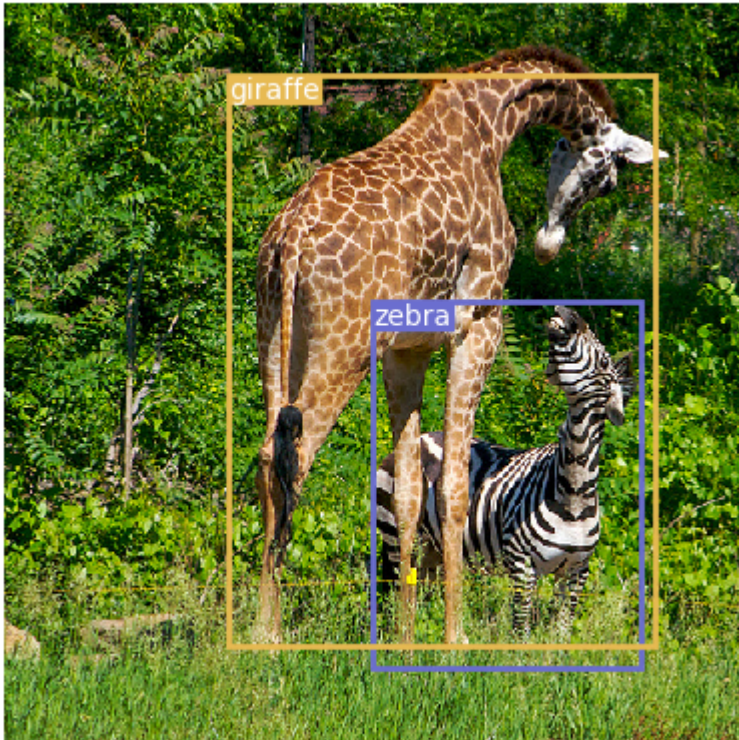
Model	mAP (min. 50 IoU)
YOLOv3 608 (paper)	57.9
YOLOv3 608 (this impl.)	57.3
YOLOv3 416 (paper)	55.3
YOLOv3 416 (this impl.)	55.5

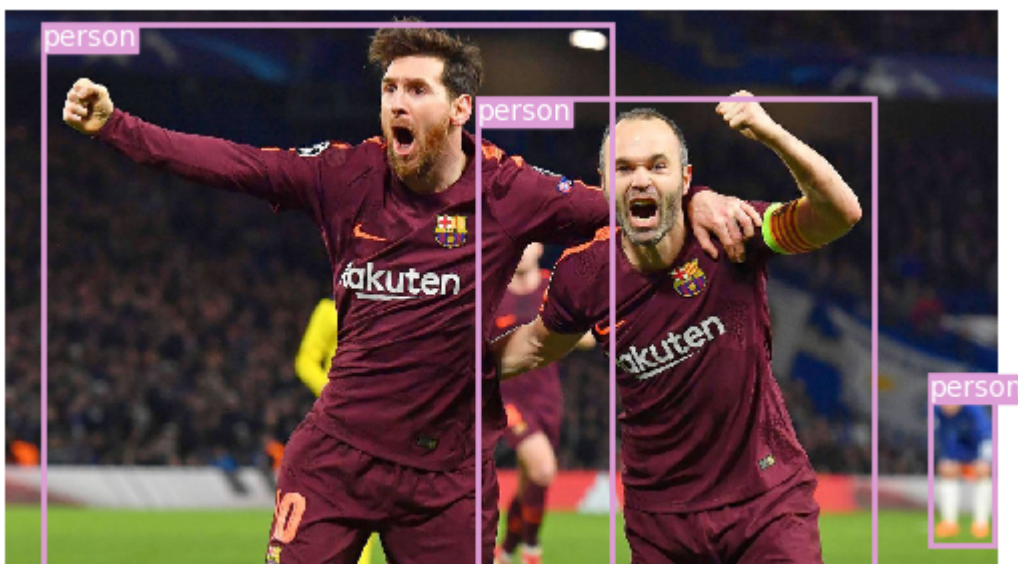
Inference

Использует предварительно обученные веса для прогнозирования изображений. Ниже приведена таблица, в которой показано время вычисления при использовании в качестве входных данных изображений, масштабированных до 256x256. Измерения ResNet backbone взяты из статьи YOLOv3. Отмеченное измерение Darknet-53 показывает время вычисления этой реализации на моей видеокарте.

Backbone	GPU	FPS
ResNet-101	Titan X	53
ResNet-152	Titan X	37
Darknet-53 (бумара)	Titan X	76
Darknet-53 (этот импл.)	1080ti	74

```
poetry run yolo-detect --images data/samples/
```





Тренировка

Для описания аргументов посмотрите `poetry run yolo-train --help`.

Пример (COCO)

Для обучения на COCO с использованием бэкенда Darknet-53, предварительно обученного на ImageNet, выполните команду:

```
poetry run yolo-train --data config/coco.data --pretrained_weights
weights/darknet53.conv.74
```

Tensorboard

Отслеживайте прогресс в обучении в Tensorboard:

- Инициализируйте обучение
- Выполните следующую команду
- Перейдите на <http://localhost:6006/>

```
poetry run tensorboard --logdir='logs' --port=6006
```

Хранение журналов на медленном диске может привести к значительному снижению скорости обучения.

Вы можете настроить каталог логов с помощью `--logdir <path>` при запуске `tensorboard` и `yolo-train`.

Обучение на пользовательском наборе данных

Пользовательская модель

Выполните приведенные ниже команды, чтобы создать определение пользовательской модели, заменив `<num-classes>` на количество классов в вашем наборе данных.

```
./config/create_custom_model.sh <num-classes> # Will create custom model 'yolov3-custom.cfg'
```

Классы

Добавьте имена классов в файл `data/custom/classes.names`. В этом файле должна быть одна строка для каждого имени класса.

Папка с изображениями

Переместите изображения вашего набора данных в папку `data/custom/images/`.

Папка аннотаций

Переместите аннотации в папку `data/custom/labels/`. Даталадер ожидает, что файл аннотаций, соответствующий изображению `data/custom/images/train.jpg`, имеет путь `data/custom/labels/train.txt`. Каждая строка в файле аннотации должна определять одно ограничивающее поле, используя синтаксис `label_idx x_center y_center width height`. Координаты должны быть масштабированы `[0, 1]`, а `label_idx` должен иметь нулевой индекс и соответствовать номеру строки имени класса в `data/custom/classes.names`.

Определите обучающий и проверочный наборы

В файлах `data/custom/train.txt` и `data/custom/valid.txt` добавьте пути к изображениям, которые будут использоваться в качестве обучающих и проверочных данных соответственно.

Обучение

Для обучения на пользовательском наборе данных выполните команду:

```
poetry run yolo-train --model config/yolov3-custom.cfg --data config/custom.data
```

Add `--pretrained_weights weights/darknet53.conv.74` to train using a backend pretrained on ImageNet.

API

Вы можете импортировать модули этого репо в свой собственный проект, если установите pip-пакет `pytorchyolo`.

Пример вызова предсказания из простого питон-скрипта OpenCV будет выглядеть следующим образом:

```
import cv2
from pytorchyolo import detect, models

# Load the YOLO model
model = models.load_model(
    "<PATH_TO_YOUR_CONFIG_FOLDER>/yolov3.cfg",
    "<PATH_TO_YOUR_WEIGHTS_FOLDER>/yolov3.weights")

# Load the image as a numpy array
img = cv2.imread("<PATH_TO_YOUR_IMAGE>")

# Convert OpenCV bgr to rgb
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Runs the YOLO model on the image
boxes = detect.detect_image(model, img)

print(boxes)
# Output will be a numpy array in the following format:
# [[x1, y1, x2, y2, confidence, class]]
```

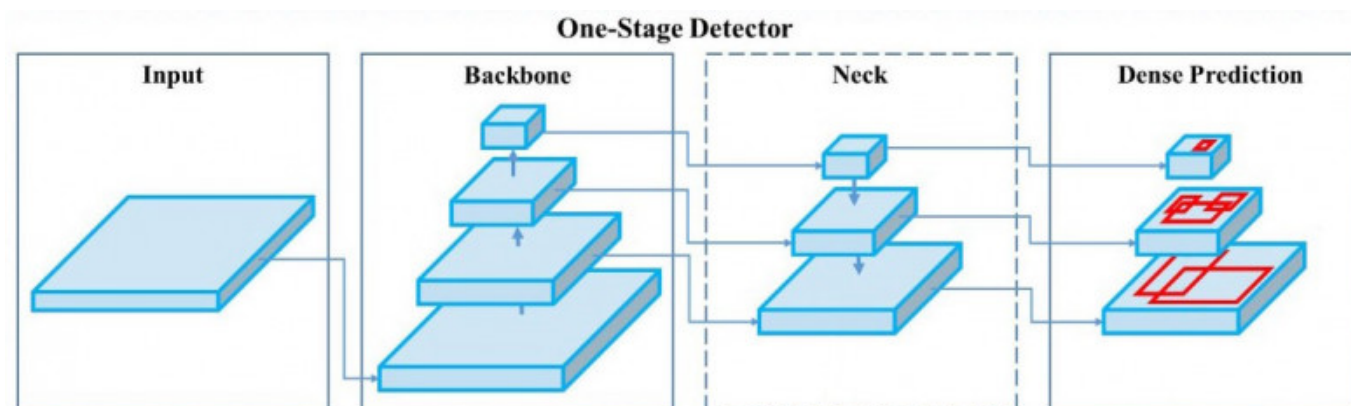
Детекция объектов с помощью YOLOv5

Введение

Иногда возникает необходимость быстро сделать кастомную модель для детекции, но разбираться в специфике компьютерного зрения и зоопарке моделей нет времени. Так появился краткий обзор проблемы детекции, и пошаговый туториал для обучения и инференса модели YOLOv5 в сервисе Google Colab.

Архитектура

YOLOv5 относится к архитектуре One-Stage detector - подход, который предсказывает координаты определённого количества *bounding box*'ов с результатами классификации и вероятности нахождения объекта, и в дальнейшем корректируя их местоположение. В целом такую архитектуру можно представить в следующем виде:



Сеть скейлит исходное изображение в несколько *feature map*'ов с использованием skip-connection и прочих архитектурных фишек. Полученные карты признаков приводятся в одно разрешение с помощью апсемплинга и конкатенируются. Затем предсказываются классы и **bounding box**'ы для объектов, далее для каждого объекта выбирается самый вероятный **bounding box** с помощью **Non-Maximum Suppression**. Подробнее можно почитать на [Medium](#).

Реализация

За основу взят репозиторий [ultralytics](#).

Установка YOLO

```
git clone https://github.com/ultralytics/yolov5
cd yolov5
pip install -r requirements.txt
```

Структура датасета

```
/dataset
--/images
--/train
  --/valid
--/labels
--/train
  --/valid
--dataset.yaml
```

Файлы изображений и меток разделены по разным папкам, в корне лежит файл конфигурации **yaml**, который имеет следующий формат:

```
train: ../dataset/images/train/
val: ../dataset/images/valid/

nc: 1 # количество классов
names: ['class_0'] # имена классов
```

Разметка

Разметка осуществляется в следующем формате:

Label_ID_1 X_CENTER_NORM Y_CENTER_NORM WIDTH_NORM HEIGHT_NORM

Для каждого изображения **<image_name>.jpg** создается файл **<image_name>.txt** с разметкой.

Значения для каждого ***bounding box***а вычисляются следующим образом:

```
X_CENTER_NORM = X_CENTER_ABS/IMAGE_WIDTH
Y_CENTER_NORM = Y_CENTER_ABS/IMAGE_HEIGHT
WIDTH_NORM = WIDTH_OF_LABEL_ABS/IMAGE_WIDTH
HEIGHT_NORM = HEIGHT_OF_LABEL_ABS/IMAGE_HEIGHT
```

Для разметки можно использовать программу [Labellmg](#).

Обучение YOLO

Обучение производится с помощью скрипта **train.py** с ключами:

--data, путь к датасету

--img, разрешение картинки

--epochs, количество эпох - рекомендуется 300-500

--cfg, конфиг размера s/m/l

--weights, стартовые веса

Существуют и другие настройки, о них можно почитать на [странице проекта](#).

После обучения можно сохранить лучшую модель, которая расположена в **../yolov5/runs/train/exp/weights/best.pt**

Инференс

Запустить модель можно несколькими способами:

- Скрипта `detect.py` указав веса модели и файл для детекции
- Torch Hub:

```
model_path = '/content/yolov5.pt'  
yolo_path = '/content/yolov5'  
model = torch.hub.load(yolo_path, 'custom', path=model_path, source='local')  
img_path = 'content/test.jpg'  
results = model([img_path])
```

- Кастомная функция `predict` на основе `detect.py`, которую можно найти в [репозитории](#).

```
from predict import predict  
pred_list = predict(  
    weights=weights,  
    source=image,  
    imgsz=[1280, 1280]  
)
```

Результаты

Пример распознавания:

Перечень жителей, подписавших вышеуказанное обращение

№№ п/п	Номер квартиры	Фамилия, имя и отчество	Подпись
Адрес: 123557, г. Москва, Средний Тишинский переулок, дом 3:			
1	87	Егорова В. Б.	
2.	84	Родневский И. П.	
3	86	Михайлов В. П.	
4	85	Шкалкова М. М.	
5.	82	Богарева М. С.	
6	77	Григорьев И. В.	
7.	72	Ромашенко А. Д.	
8.	71	Морозов Х. В.	
9.	74	Александровы	
10	90	Ломухов Г. И. С.	
11	89	Ратновская А. А.	
12	51	Зенкова Е. А.	
13	47	Дорожников И. И.	
14	43	Иванова Н. А.	
15	56	Стефановская И. В.	
16	55	ВАННЕР ХРИСТОФ	
17	73	ВАННЕР ХРИСТОФ	
18	82	Шуфутинский Марк	

Видно, что есть и пропуски, и ошибки распознавания, но сетка училась на сотне примеров из поисковой выдачи гугла, так что с ростом количества данных должно расти и качество предсказания.

Заключение

Код для запуска и пример датасета доступен в репозитории на [github](#).