Justyna Kloc
232999

"Gdzie jest BoJack?"- aplikacja oparta na sieci neuronowej służąca do rozpoznawania obiektu

Spis treści

0.1. Wstęp

W tym sprawozdaniu przedstawię proces uczenia sieci neuronowej za pomocą interfejsu API Tensorflow Object Detection API i korzystania ze skryptu zbudowanego wokół niego w celu znalezienia BoJacka. Składa się z następujących dwóch najważniejszych kroków:

1. Przygotowanie zestawu danych poprzez utworzenie oznakowanych obrazów szkoleniowych, w których etykiety reprezentują xy lokalizacji BoJacka na obrazie.
2. Pobieranie i konfigurowanie modelu do użycia z interfejsem API Tensorflow Object Detection

0.2. Przygotowanie zbioru danych

Wartościami docelowymi dla najprostszych problemów z uczeniem maszynowym są zwykle skalary (jak w przypadku detektora cyfr) lub ciąg kategoryczny. Dane szkoleniowe interfejsu API Tensorflow Object Detection wykorzystują kombinację obu. Składa się z zestawu obrazów wraz z etykietami pożądanych obiektów i lokalizacji, w które pojawiają się na obrazie. Lokalizacje są zdefiniowane za pomocą dwóch punktów, ponieważ (w przestrzeni 2d) wystarczą dwa punkty, aby narysować prostokąt wokół obiektu.

Aby stworzyć idealny zestaw szkoleniowy, potrzebowałam zdjęć BoJacka - pobrane z Google Grafika, głównie kadry z serialu, jak i wymyślić zestaw zdjęć z łamigłówkami typu Where's Wally z lokalizacjami, w których pojawia się Wally, czyli w moim przypadku BoJack. W tym celu wykorzystałam sztukę współczesną "The Picure of Everything" autorstwa Horwarda Hallisa (http://www.howardhallis.com/tpoe/noflash.html).

Przygotowanie zbioru danych treningowych i testowych opierało się na kolejnych krokach.

1. Znalezienie i zapisanie obrazów w formacie .jpg lub .jpeg
2. Doklejenie sylwetki BoJacka do obrazów z dużą ilością postaci
3. Rysowanie i zapisanie parametrów bounding boxów dla wszystkich obrazów - aplikacja **BBox-Label-Tool**
4. Stworzenie pliku .csv z danymi- nazwa pliku, wymiary obrazu, klasa, obiektu, parametry bounding boxa
5. Konwersja zdjęć i pliku .csv to rekordów binarnych, osobno dla treningowych i testowych
   Ostatecznie zbiór moich danych wyglądał następująco: 141 treningowych i 24 obrazów testowych

0.3. Przygotowanie plików konfiguracyjnych

Interfejs API Tensorflow Object Detection zapewnia zestaw wstępnie przeszkolonych modeli o różnych osiągach (zwykle kompromis dokładności prędkości) przeszkolonych na kilku publicznych zbiorach danych. Chociaż model można wyszkolić od zera, zaczynając od losowo zainicjowanych wag sieciowych, proces ten prawdopodobnie potrwa tygodnie. Zamiast tego zastosowałam metodę transer-learningu.

Polega ona na wzięciu modelu zwykle przeszkolonego do rozwiązania jakiegoś ogólnego problemu i przekwalifikowaniu go, aby rozwiązał mój. Ideą takiego uczenia jest to, że zamiast odkrywać koło od nowa, szkoląc nasz model od zera, możemy wykorzystać wiedzę uzyskaną we wcześniej wyszkolonym modelu i przenieść go na nowy.

Użyłam modelu: SSD Mobilenet v1 z 11.06.2017 wyszkolonym na zbiorze danych COCO wraz z plikiem konfiguracyjnym. Model zawiera plik .ckpt punktu kontrolnego, którego mogłam użyć do rozpoczęcia szkolenia. Należało skonfigurować ścieżki, które prowadziły między innymi do lokalizacji plików binarnych.

W pliku konfiguracyjnym trzeba było wyszukać wszystkie punkty PATH_TO_BE_CONFIGURED i zmienić je, a także zmodyfikować rozmiar batcha. W wyjściowym pliku konfiguracyjnym jest ustawiony na 24. Ustaliłam ten parametr na 8 (zgodnie z zasadą, aby była to potęga 2). Na koniec należało zmienić nazwę i ścieżkę punktu kontrolnego, num_classes na 1, num_examples na 12, oraz label_map_path.

```
# SSD with Mobilenet v1, configured for Oxford-IIIT Pets Dataset.
#Users should configure the fine_tune_checkpoint field in the train config as
# well as the label_map_path and input_path fields in the train_input_reader and
# eval_input_reader. Search for "PATH_TO_BE_CONFIGURED" to find the fields that
# should be configured.

        model {
        ssd {
        num_classes: 1
        box_coder {
        faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
        height_scale: 5.0
        width_scale: 5.0
        }
        }
        matcher {
        argmax_matcher {
        matched_threshold: 0.5
        unmatched_threshold: 0.5
        ignore_thresholds: false
        negatives_lower_than_unmatched: true
        force_match_for_each_row: true
        }
        }
        similarity_calculator {
        iou_similarity {
        }
        }
        anchor_generator {
        ssd_anchor_generator {
        num_layers: 6
        min_scale: 0.2
        max_scale: 0.95
        aspect_ratios: 1.0
        aspect_ratios: 2.0
        aspect_ratios: 0.5
        aspect_ratios: 3.0
        aspect_ratios: 0.3333
        }
        }
        image_resizer {
        fixed_shape_resizer {
        height: 300
        width: 300
        }
        }
        box_predictor {
        convolutional_box_predictor {
        min_depth: 0
        max_depth: 0
        num_layers_before_predictor: 0
        use_dropout: false
```

```
dropout_keep_probability: 0.8
kernel_size: 1
box_code_size: 4
apply_sigmoid_to_scores: false
conv_hyperparams {
activation: RELU_6,
regularizer {
l2_regularizer {
weight: 0.00004
}
}
initializer {
truncated_normal_initializer {
stddev: 0.03
mean: 0.0
}
}
batch_norm {
train: true,
scale: true,
center: true,
decay: 0.9997,
epsilon: 0.001,
}
}
}
}
feature_extractor {
type: 'ssd_mobilenet_v1'
min_depth: 16
depth_multiplier: 1.0
conv_hyperparams {
activation: RELU_6,
regularizer {
l2_regularizer {
weight: 0.00004
}
}
initializer {
truncated_normal_initializer {
stddev: 0.03
mean: 0.0
}
}
batch_norm {
train: true,
scale: true,
center: true,
decay: 0.9997,
epsilon: 0.001,
}
}
}
loss {
classification_loss {
weighted_sigmoid {
anchorwise_output: true
}
}
localization_loss {
weighted_smooth_l1 {
anchorwise_output: true
}
```

```
}
hard_example_miner {
num_hard_examples: 3000
iou_threshold: 0.99
loss_type: CLASSIFICATION
max_negatives_per_positive: 3
min_negatives_per_image: 0
}
classification_weight: 1.0
localization_weight: 1.0
}
normalize_loss_by_num_matches: true
post_processing {
batch_non_max_suppression {
score_threshold: 1e-8
iou_threshold: 0.6
max_detections_per_class: 100
max_total_detections: 100
}
score_converter: SIGMOID
}
}
}

train_config: {
batch_size: 8
optimizer {
rms_prop_optimizer: {
learning_rate: {
exponential_decay_learning_rate {
initial_learning_rate: 0.004
decay_steps: 800720
decay_factor: 0.95
}
}
momentum_optimizer_value: 0.9
decay: 0.9
epsilon: 1.0
}
}
fine_tune_checkpoint: "/home/justyna/Pulpit/gdziejestbojack/Object-Detection/
ssd_mobilenet_v1_coco_11_06_2017/model.ckpt"
from_detection_checkpoint: true
data_augmentation_options {
random_horizontal_flip {
}
}
data_augmentation_options {
ssd_random_crop {
}
}
}

train_input_reader: {
tf_record_input_reader {
input_path: "/home/justyna/Pulpit/gdziejestbojack/Object-Detection/
data/train.record"
}
label_map_path: "/home/justyna/Pulpit/gdziejestbojack/Object-Detection/
data/object-detection.pbtxt"
}

eval_config: {
```

```
num_examples: 40
}

eval_input_reader: {
tf_record_input_reader {
input_path: "/home/justyna/Pulpit/gdziejestbojack/Object-Detection/
data/test.record"
}
label_map_path: "/home/justyna/Pulpit/gdziejestbojack/Object-Detection/
data/object-detection.pbtxt"
shuffle: false
num_readers: 1
}
```

Ostatnim plikiem, który musiałam skonfigurować, jest plik labels.txt, który zawiera etykiety wszystkich obiektów. Ponieważ szukam tylko jednego typu obiektu, plik etykiet wygląda następująco:
item {
id: 1
name: 'BoJack'
}

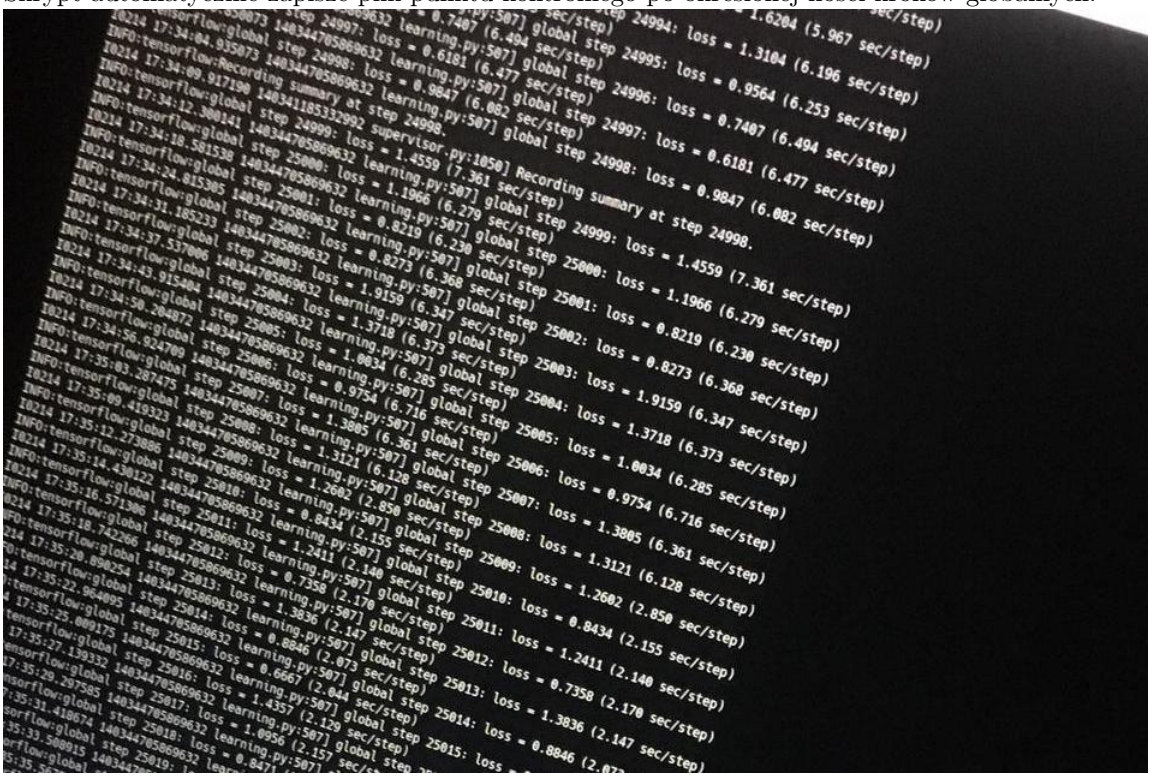
0.4. Trening

Interfejs API Tensorflow Object Detection zapewnia prosty w użyciu skrypt Pythona do lokalnego przeszkolenia naszego modelu.
python3 train.py - -logtostderr - -pipeline_config_path = PATH_TO_PIPELINE_CONFIG - -train_dir = PATH_TO_TRAIN_DIR

Gdzie PATH_TO_PIPELINE_CONFIG jest ścieżką do pliku konfiguracyjnego, a PATH_TO _TRAIN_DIR to nowo utworzony katalog, w którym będą przechowywane nowe punkty kontrolne i model.

Najważniejszymi informacjami, które należy obserwować w trakcie treningu jest strata. Jest to suma błędów popełnianych dla każdego przykładu w zestawach szkoleniowych lub walidacyjnych. Dążę do tego, aby ta wartość była jak najniższa.

Skrypt automatycznie zapisze plik punktu kontrolnego po określonej ilości kroków globalnych.

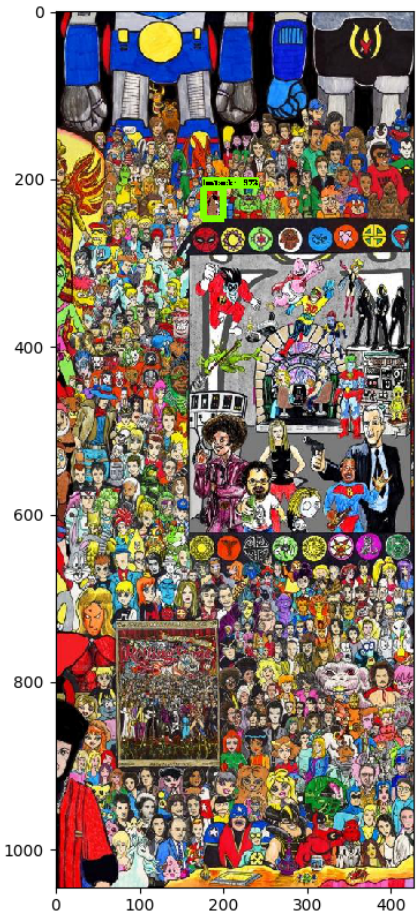


Wyuczony model został wygenerowany po 57934 globalnym kroku, ponieważ wtedy strata była poniżej 1.

0.5. Aplikacja znajdująca BoJacka

Do aplikacji wyszukującej BoJacka potrzebny jest jeszcze wyeksportowany model wytrenowanej sieci. Do tego celu użyłam gotowej funkcji z wykorzystywanego modułu tensorflow/models/research/object_detection/export_inference_graph.py

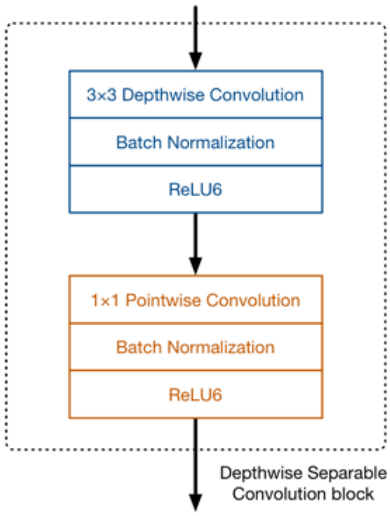
W pliku wykonywalnym **APP.py** używane są już zapisane rezultaty działania detektora, natomiast działanie w czasie rzeczywistym można sprawdzić funkcją **find_bojack.py** lub **find2.py**.



0.6. Dodatek teoretyczny: SSD Mobilenet v1

Ideą stojącą za MobileNet V1 jest to, że warstwy splotowe, które są niezbędne do zadań z zakresu wizji komputerowej, ale są dość drogie w obliczeniach, można zastąpić tak zwanymi splotami rozłącznymi.

Zadanie warstwy splotowej jest podzielone na dwa podzadania: najpierw jest głęboka warstwa splotowa, która filtruje dane wejściowe, a następnie warstwa splotowa 1×1 (lub punktowa), która łączy te filtrowane wartości w celu stworzenia nowych funkcji



Razem, głębokie i punktowe, tworzą razem blok splotów „dający się oddzielić". Robi to mniej więcej tak samo jak tradycyjny splot, ale jest znacznie szybszy.

Pełna architektura MobileNet V1 składa się z regularnego splotu 3×3 jako pierwszej warstwy, po której następuje 13-krotność powyższego bloku konstrukcyjnego. [MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications]

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Whole Network Architecture for MobileNet

Pomiędzy tymi głęboko oddzielnymi blokami nie ma warstw puli. Zamiast tego niektóre głębokie warstwy mają krok 2, aby zmniejszyć przestrzenne wymiary danych. Kiedy tak się dzieje, odpowiednia warstwa punktowa podwaja również liczbę kanałów wyjściowych. Jeśli obraz wejściowy to $224 \times 224 \times 3$, wówczas wyjściem sieci jest mapa obiektów $7 \times 7 \times 1024$.

Jak zwykle we współczesnych architekturach, po warstwach splotu następuje normalizacja wsadowa. Funkcją aktywacyjną używaną przez MobileNet jest ReLU6. Jest to podobne do znanego ReLU, ale zapobiega zbyt dużej aktywacji.

Autorzy artykułu MobileNet stwierdzili, że ReLU6 jest bardziej niezawodna niż zwykła ReLU przy użyciu obliczeń o niskiej precyzji. Sprawia również, że kształt funkcji wygląda bardziej jak sigmoid.

W klasyfikatorze opartym na MobileNet na końcu znajduje się zwykle globalna średnia warstwa puli, po której następuje w pełni połączona warstwa klasyfikacyjna lub równoważny splot 1×1 i softmax.

W rzeczywistości jest więcej niż jeden MobileNet. Został zaprojektowany jako rodzina architektur sieci neuronowych. Istnieje kilka hiperparametrów, które pozwalają grać z różnymi kompromisami architektury.

Najważniejszym z tych hiperparametrów jest mnożnik głębokości, myląco znany również jako „mnożnik szerokości". To zmienia liczbę kanałów w każdej warstwie. Zastosowanie mnożnika głębokości 0,5 zmniejszy o połowę liczbę kanałów używanych w każdej warstwie, co zmniejszy liczbę obliczeń o współczynnik 4 i liczbę możliwych do nauczenia się parametrów o współczynnik 3. Jest zatem znacznie szybszy niż pełny model, ale również mniej dokładne.

Dzięki innowacjom głęboko separowalnych zwojów MobileNet musi wykonać około 9 razy mniej pracy niż porównywalne sieci neuronowe z tą samą dokładnością. Ten rodzaj warstwy działa tak dobrze, że udało mi się uzyskać modele z ponad 200 warstwami do działania w czasie rzeczywistym, nawet na iPhonie 6s.

0.7. Dodatek teoretyczny: COCO

COCO to zestaw danych do wykrywania, segmentacji i podpisywania obiektów na dużą skalę. Ta wersja zawiera obrazy, ramki ograniczające i etykiety dla wersji 2014. COCO 2017 używa tych samych obrazów co 2014, ale różne podziały między train / val / test. Podział testu nie zawiera adnotacji (tylko obrazy). COCO definiuje 91 klas, ale dane używają tylko 80 klas.