TREINAMENTO YOLOV3 E TINY-YOLOV3

V 1.2

JOSE AMAT – SAS SOFTWARE & INNOVATION SYSTEMS ENGINEER

INNOVATION AND GROWTH INITIATIVES

SUMÁRIO

INTRODUÇÃO

Cada dia vemos mais soluções que fazem uso do Computer Vision para que estas se tornem mais inteligentes e automatizadas. Este tipo de tecnologia tem sido usada por diversas áreas da industria, desde empresas de beleza, até linhas de montagem.

Existem diversos métodos para o treinamento de modelos de Deep Learning que são capazes de reconhecer objetos personalizados. A escolha deles depende muito da aplicação que se deseja criar, existindo a possibilidade de perda de precisão, mas ganhando rapidez e eficácia na detecção.

O intuito deste documento é ser um guia rápido, que mostre passo a passo o treinamento de um modelo de Deep Learning que possa reconhecer objetos personalizados, e falando em objetos personalizados, é recomendável que você defina incialmente o que você quer reconhecer. Nada melhor do que ter um objetivo claro para colocar o aprendizado em prática.

Por tanto, se esta for sua primeira vez treinando uma rede neural, ou mesmo se você não souber o que ela é, não precisa entrar em desespero, iremos explicar conceitos e definições de maneira simples e resumida, para que no final deste guia você adquira conhecimento, além de ter treinado seu modelo.

OBJETIVOS

Nosso objetivo é mostrar todo o processo de treinamento de um modelo YOLO, que serve para detecção de objetos. A maneira de exemplo, treinaremos o modelo para fazer deteção de camisetas de equipes de futebol. Posteriormente o colocaremos em uma solução de promoções personalizadas de uma loja, correspondente ao time de futebol de cada cliente, que pode ser detectado com uma pequena câmera em tempo real.

Cabe ressaltar que nesta documentação explicaremos apenas a parte do Computer Vision, treinando um modelo que possa fazer a detecção das camisetas das seleções do Brasil, Argentina, Colômbia, e México.

O QUE É YOLO?

You Only Look Once ou, simplesmente YOLO, é um algoritmo caracterizado pela sua velocidade na detecção de objetos. O uso do YOLO é recomendado para detecções de objetos em tempo real, sem se importar muito com sua precisão. YOLO trabalha olhando para a imagem apenas uma vez, fazendo com que a detecção fique mais rápida.

Para sermos mais exatos, YOLO divide a imagem em 13x13 células, as quais são analisadas uma a uma, até achar regiões com maior probabilidade de conter um objeto (qualquer objeto), uma vez feito isto, YOLO descarta as caixas com baixa probabilidade de conter um objeto, ficando apenas com as de maior probabilidade.

Dessa maneira, o algoritmo analisa as caixas que ficaram, usando um processo de classificação de imagem, para predizer a que tipo de classe pertencente àquele objeto.

O QUE É DARKNET?

Darknet é um framework de rede neural para código aberto que foi escrito em C e CUDA. Segundo a documentação original, Darknet tem suporte para YOLO: Real-Time Object Detection, ImageNet Classification, Train a Classifier on CIFAR-10, entre outros.

PRE-REQUISITOS

- Git
- Darknet
- Ambiente Linux com GPU (EC2 p2.xlarge)
    - Deep Learning AMI (Ubuntu 16.04) Version 25.3 (ami-01320196d47bc6dfa)
- CUDA e CUDNN instalados

Cabe ressaltar que o ambiente Linux com GPU, não precisa ser exatamente o mesmo. Ele pode ser a sua própria workstation, como também pode ser algum outro servidor de algum outro fornecedor cloud (GCP, Azure, IBM, etc). O importante aqui é que o modelo da sua GPU seja a partir da NVIDIA GTX 10.

Em termos de padronização deste guia, iremos explicar o processo nos baseando na instância da AWS com o ambiente Linux especificado acima.

Se você, possui algum outro ambiente com GPU, no qual queira replicar o treinamento, saiba que existe a possibilidade de alguma ou outra coisa mudar.

Sendo assim, vá por sua própria conta e risco, que com paciência você conseguirá chegar com sucesso no final!

COMPILANDO DARKNET

CLONANDO O PROJETO

Se você estiver usando um servidor da AWS com Ubuntu, é garantido que a rota */home/ubuntu* existe.

Primeiro vamos clonar nosso projeto:

```
sudo su
cd /home/ubuntu
git clone https://github.com/pjreddie/darknet.git
cd darknet
```

CONFIGURANDO O PROJETO

Agora vamos editar nosso arquivo *Makefile*:

```
vi /home/ubuntu/darknet/Makefile
```

```
1) GPU=1
2) CUDNN=1
```

É necessário passar a rota correta do paramento *NVCC*, dentro do *Makefile*, neste ambiente é:

```
24) NVCC=/usr/local/cuda-10.0/bin/nvcc
```

Confira também que os caminhos da versão do CUDA que você irá usar estão corretos nas seguintes linhas. No nosso caso os caminhos são:

```
50) COMMON+= -DGPU -I/usr/local/cuda-10.0/include/
51) CFLAGS+= -DGPU
52) LDFLAGS+= -L/usr/local/cuda-10.0/lib64 -lcuda -lcudart -lcublas -lcurand
```

Pela configuração original, Darknet só salva o treinamento a cada 100 iterações, até chegar na iteração número 900, a partir daí ele começa a salvar a cada 10000 iterações. Isto é ruim se você precisar parar o treinamento em alguma parte do processo, sem perder o que já foi feito. Para isso nós vamos editar o arquivo *detector.c*:

```
vi /home/ubuntu/darknet/examples/detector.c
```

Vá para a linha 138 e edite *if(i%10000==0 || (i < 1000 && i%100 == 0)){* para:

```
138)                    if(i < 10000 && i%400 == 0){
```

Isto fará com que salve a cada 400 iterações, até chegar na iteração 10000.

Uma vez editado vamos no caminho:

```
cd /home/ubuntu/darknet/
```

E compilamos o projeto com o seguinte comando:

```
make
```

Nosso output deverá ser parecido com esse:

```
mkdir -p obj
gcc -I/usr/local/cuda/include/  -Wall -Wfatal-errors  -Ofast....
gcc -I/usr/local/cuda/include/  -Wall -Wfatal-errors  -Ofast....
gcc -I/usr/local/cuda/include/  -Wall -Wfatal-errors  -Ofast....
```

```
.....
gcc -I/usr/local/cuda/include/  -Wall -Wfatal-errors  -Ofast -lm....
```

Se seu output foi esse, quer dizer que você está no caminho correto. Se seu output não foi esse, coloque o comando "make clean" e tente debugar o problema.

EXPORTANDO AS VARIÁVEIS DO PROJETO

Agora você precisa exportar as seguintes variáveis de ambiente:

```
export PATH=/usr/local/cuda-10.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-
10.0/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

Confira que os caminhos existem na sua máquina. Perceba que até agora nós usamos a versão 10.0 do CUDA.

TESTANDO O PROJETO

Para testar o sucesso da sua compilação coloque o seguinte comando:

```
./darknet
```

Seu output deve ser:

```
usage: ./darknet <function>
```

Pronto, sua compilação foi feita com sucesso!

CRIANDO O ARQUIVO DE CONFIGURAÇÃO

Nesta etapa podemos optar tanto pelo modelo tiny, quanto pelo modelo normal. O tiny-YOLO é um modelo simplificado, ideal para demonstrações. Já o YOLO normal possui uma demanda de processamento maior, aumentando sua precisão nas detecções.

YOLO-V3 & TINY YOLO-V3

Criaremos um arquivo *.cfg* chamado *yolo-obj.cfg* (esse nome é arbitrário), e copiaremos nele o conteúdo do arquivo *yolov3.cfg* ou *yolov3-tiny-obj.cfg*, dependendo da rede

escolhida por você. Para isso, estando dentro da pasta *darknet,* coloque os seguintes comandos:

Para YOLO-V3:

```
cd /home/ubuntu/darknet
cp cfg/yolov3.cfg cfg/yolo-obj.cfg
```

Para Tiny YOLO-V3:

```
cd /home/ubuntu/darknet
cp cfg/yolov3-tiny.cfg cfg/yolov3-tiny-obj.cfg
```

Edite seu arquivo *yolo-obj.cfg* ou *yolov3-tiny-obj.cfg* criado para editar os parâmetros *batch* e *subdivisions*.

Para YOLO-V3:

```
vi cfg/yolov3-tiny.cfg
```

Para Tiny YOLO-V3:

```
vi cfg/yolov3-tiny-obj.cfg
```

Coloque os valores *32* e *8* nas seguintes linhas, independemente da quantidade de objetos que for detectar.

```
6) batch=32
7) subdivisions=8
```

Editando os *batch* e *subdivisions* conseguimos reduzir o consumo de memória tanto na hora de treinamento quanto na hora de teste.

Agora vamos editar o número de objetos. Edite o parâmetro colocando o valor do número de objetos que você quer detectar. Neste caso queremos detectar 4 objetos:

```
610) classes=4
696) classes=4
783) classes=4
```

Edite o parâmetro *max_batches* colocando o valor do produto do número de objetos vezes 2000 *(classes * 2000)*. Neste caso o valor é *4 * 2000 = 8000*:

```
20) max_batches = 8000
```

Obs: O valor mínimo a colocar é 4000, ou seja, se você for detectar apenas 1 objeto coloque *max_batches = 4000*, para 2 objetos *max_batches = 4000*, para 3 objetos *max_batches = 6000*, para 4 objetos *max_batches = 8000*, ...

Atribua o valor do *80%* e *90%* do seu *max_batches* no parâmetro steps:

```
22)steps=6400,7200
```

Altere o parâmetro *filters* colocando o valor de *(classes + 5)*3* nas seguintes linhas. Neste caso o valor é *(4 + 5)*3 = 27*:

```
603) filters=27
689) filters=27
776) filters=27
```

TREINAMENTO DE OBJETOS

CRIANDO ARQUIVOS .NAMES E .DATA

Dentro da pasta *data*, crie um arquivo de texto com o nome *obj.names*:

```
cd /home/ubuntu/darknet/data
vi obj.names
```

Dentro dele insira os nomes dos seus objetos a reconhecer:

```
Brasil
Argentina
Colombia
Mexico
```

Cada linha representa um objeto diferente, começando do *0* até o *n-1* objetos (n = número de objetos/classes).

Agora, dentro da pasta *./data*, crie um outro arquivo de texto chamado *obj.data*.

```
cd /home/ubuntu/darknet/data
vi obj.data
```

Nele, insira as seguintes informações:

```
Classes = 4
train  = data/train.txt
```

```
valid  = data/test.txt
names = data/obj.names
backup = backup/
```

Obs: Está vendo os arquivos *train.txt* e *test.txt*? Não se preocupe, iremos falar deles mais na frente.

Substitua o valor *classes* pelo seu respectivo valor (*classes* corresponde ao número de objetos).

PREPARAÇÃO DO BANCO DE IMAGENS

EXPLICANDO OS LABELS

Antes de iniciar o treinamento precisamos preparar as images com seus respectivos labels.

Os labels são arquivos txt que contem as coordenadas do nosso objeto dentro da imagem.

Exemplo:

```
1)1 0.241406 0.377778 0.084375 0.241667
2)0 0.320703 0.339583 0.078906 0.284722
3)0 0.397266 0.404861 0.128906 0.320833
4)3 0.549219 0.377083 0.154687 0.276389
5)1 0.684375 0.395139 0.168750 0.326389
6)2 0.732031 0.304167 0.129688 0.327778
```

No exemplo acima encontramos 6 linhas, onde cada uma representa um objeto e sua posição, e dentro de cada linha encontramos 5 valores, onde o primeiro (na cor vermelha) corresponde ao número do objeto. Neste caso temos 4 objetos para reconhecer: 0, 1, 2 e 3) e os 4 restantes (na cor preta) são as coordenadas *X* e *Y* do plano cartesiano que correspondem ao quadrilátero que contem nosso objeto, junto com sua altura e comprimento.

Aqui um link que explica melhor o significado das labels:

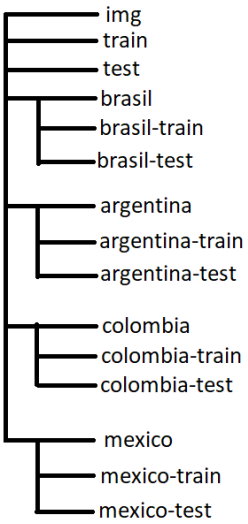
https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b

ORGANIZAÇÃO DE PASTAS

Neste treinamento faremos o reconhecimento de 4 camisetas das seleções de futebol (*Brasil, Argentina, Colombia, e Mexico*). Para isso, criaremos primeiramente 3 pastas: *img*, *train*, e *test*.

Depois criaremos uma pasta para cada objeto, e dentro de cada uma delas criaremos mais 2 pastas chamadas *<nome do objeto>*-train e *<nome do objeto>*-test.

Nossa hierarquia de pastas teria que ficar assim:



COLETA DE IMAGENS

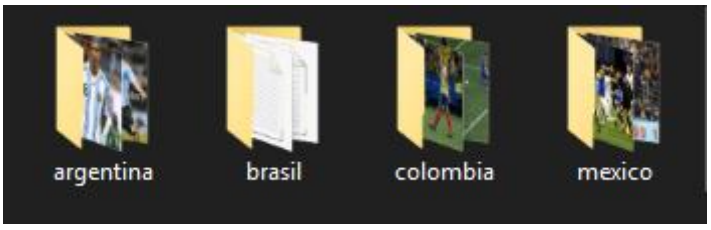
Nesta etapa do processo começaremos com a coleta de imagens. Surge uma questão importante: onde coletá-las? Acontece que fazer um modelo de computer vision para camisetas de times de futebol é um caso muito específico, fato que dificulta a procura de um dataset de imagens prontas na internet.

Neste caso nós optamos por usar a ferramenta chamada *Google Images Download*, a qual auxilia no download de imagens en grandes quantidades.

Você pode aprender mais sobre a ferramenta no seguinte link:

https://google-images-download.readthedocs.io/en/latest/

Nós baixamos em média 750 imagens por objeto, e separamos elas em pastas diferentes:



Renomeie cada imagem começando pelo 1.jpg. Nós começaremos renomeando todas as imagens jpg correspondentes à pasta argentina (1.jpg, 2.jpg, ..., 726.jpg), depois repetimos o processo na pasta brasil, continuando a enumação feita na pasta anterior (727.jpg, 728.jpg, ..., 1434.jpg)

Garanta que tenham em media umas 750 imagnes por pasta de cada objeto, e que estas 750 imagens estejam divididas em 2 subpastas na proporção de 80 % de treinamento para 20% para teste.

Algumas observações:

É recomendável que, uma vez tendo todas as imagens baixadas, façamos uma limpeza excluindo as que não iremos usar. Por exemplo: Terão imagens coletadas que não contém nenhum dos nossos objetos a reconhecer.

Outra observação que cabe ressaltar é que não há problema se tivermos mais de um objeto dentro de uma imagem. Por exemplo: Nas imagens do jogo do Brasil x Argentina terão 2 objetos para reconhecer.

YOLO_LABEL

Os labels são parte importante deste processo. Existem diversos programas que nos auxiliam na criação deles. Nós usaremos o Yolo_Label. Você pode baixar esta ferramenta no link a seguir:

https://github.com/developer0hye/Yolo_Label

Nota: Agradecimentos especiais para *developer0hye* por sua contribuição à comunidade com esta ferramenta.

Uma vez baixada a ferramenta, iniciaremos o processo dos labels, pasta por pasta. Lembrando que para cada objeto (*Brasil, Argentina, Colombia,* e *Mexico*) temos mais 2 subpastas com os nomes <*nome do objeto*>-train e <*nome do objeto*>-test.

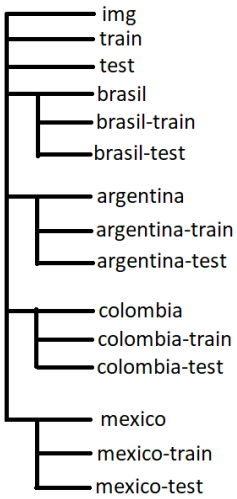
Você terá acabado quando todas as imagens do seu dataset tiverem seus respectivos labels.

É muito importante que o nome do label (arquivo txt) coincida com o nome da imagem que foi extraida (arquivo jpg).

Exemplo: Se a minha imagem for *dog.jpg*, seu label correspondente deverá se chamar *dog.txt*

ARQUIVOS TRAIN.TXT E TEST.TXT

Primeiro vamos relembrar como ficou nossa arquitetura de pastas:



Vamos copiar todas as imagens das subpastas de treinamento (*<nome do objeto>*-train) dos 4 objetos, para a pasta *train*, depois repetimos o mesmo processo com as imagens de teste (*<nome do objeto>*-test) de cada objeto, para a pasta *test*. Por último, copiaremos todas as imagens de treinamento e teste que estão em *train* e *test*, respectivamente, para a pasta *img*.

Este tipo de organização é importante para podermos diferenciar as imagens do nosso dataset final que serão usadas para treinamento, das que serão usadas para teste, para posteriormente poder listar seus caminhos nos arquivos txt: *train.txt* e *test.txt*.

Perceba que estamos apenas copiando e não movendo os arquivos. Se você mover os arquivos você corre risco de não ter um backup e ter que fazer os labels de novo!

TRAIN.TXT

No arquivo train.txt você precisa listar os caminhos das imagens, linha por linha. Você pode criar este arquivo do jeito que você preferir. Nós usaremos um script python para gerar este arquivo.

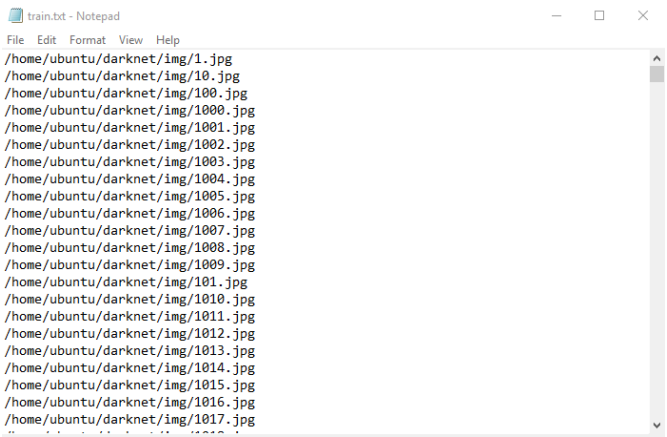
```
import os

# Function to rename multiple files
def main():

        fh = open('train.txt', 'w')
```

```
    #i = 1

        for filename in os.listdir("path-to-train-directory"):
                if filename.endswith(".jpg"):
                        src = 'path-to-train-directory\\'+ filename
                        fh.write("/path-to-img-
directory/{}\n".format(filename))
        fh.close()

# Driver Code
if __name__ == '__main__':

    # Calling main() function
    main()
```

Substitua os valores *path-to-train-directory* e *path-to-img-directory*, pelos seus valores correspondentes.

Nós estamos usando uma instância EC2 da AWS, por isso a rota de todas nossas imagens será

```
/home/ubuntu/darknet/img/nome-da-imagem.jpg
```



TEST.TXT

Para o arquivo *test.txt* o processo é similar. Trocamos o nome do arquivo dentro do código para *test.txt* e colocalos os respectivos caminhos.

```
import os

# Function to rename multiple files
def main():
```

```
        fh = open('test.txt', 'w')

    #i = 1

        for filename in os.listdir("path-to-test-directory"):
                if filename.endswith(".jpg"):
                        src = 'path-to-test-directory\\'+ filename
                        fh.write("/path-to-img
directory/{}\n".format(filename))
        fh.close()

# Driver Code
if __name__ == '__main__':

    # Calling main() function
    main()
```

TREINAMENTO

IMPORTANDO ARQUIVOS

Coloque os arquivos na sua máquina Linux, dentro do projeto darknet. O caminho ficará:

```
/home/ubuntu/darknet/img
```

Coloque seus arquivos *train.txt* e *test.txt* dentro da pasta darknet/data

```
/home/ubuntu/darknet/data/train.txt
/home/ubuntu/darknet/data/test.txt
```

Já que nós geramos todos estes arquivos *txt* dentro do sistema operacional Windows, e iremos trabalhar com eles num sistema operacional Linux (Ubuntu), é recomendável colocar os seguintes comandos:

```
cd /home/ubuntu/darknet
dos2unix ./img/*.txt
dos2unix ./data/*.txt
dos2unix ./data/obj.data
dos2unix ./data/obj.names
dos2unix ./cfg/*.cfg
```

MODELO PRE-TREINADO

Quase tudo pronto para iniciarmos o treinamento.

Vamos baixar agora um modelo pre-treinado que servirá de base para nosso treinamento.

Modelo para YOLO-V3:

```
cd /home/ubuntu/darknet
wget https://pjreddie.com/media/files/darknet53.conv.74
```

Obs: Note que este arquivo baixado será usado para treinamento.

Modelo para Tiny YOLO-V3:

```
cd /home/ubuntu/darknet
wget https://pjreddie.com/media/files/yolov3-tiny.weights
darknet.exe partial cfg/yolov3-tiny.cfg yolov3-tiny.weights yolov3-tiny.conv.15 15
```

Obs: Note que para o Tiny YOLO-V3 será gerado um arquivo chamado yolov3-tiny.conv.15 que será usado para o treinamento.

TMUX (OPCIONAL)

Nós usaremos o tmux, um programa que serve para criar sessões no terminal. Assim, se você iniciar seu treinamento e fechar o terminal, o treinamento continuará funcionando dentro da sessão tmux:

```
sudo apt install tmux
```

Agora criaremos uma nova de nome treino

```
tmux new -s treino
```

EXECUTANDO O TREINAMENTO

Pronto, agora coloquemos um dos seguintes comandos para inicar o treino.

Para YOLO-V3:

```
./darknet detector train data/obj.data cfg/yolo-obj.cfg darknet53.conv.74 | tee /home/ubuntu/darknet/train.log
```

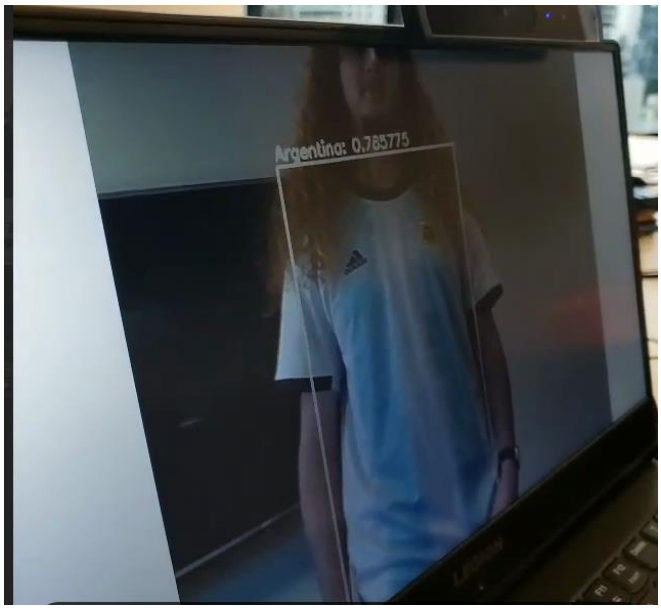
Para Tiny YOLO-V3:

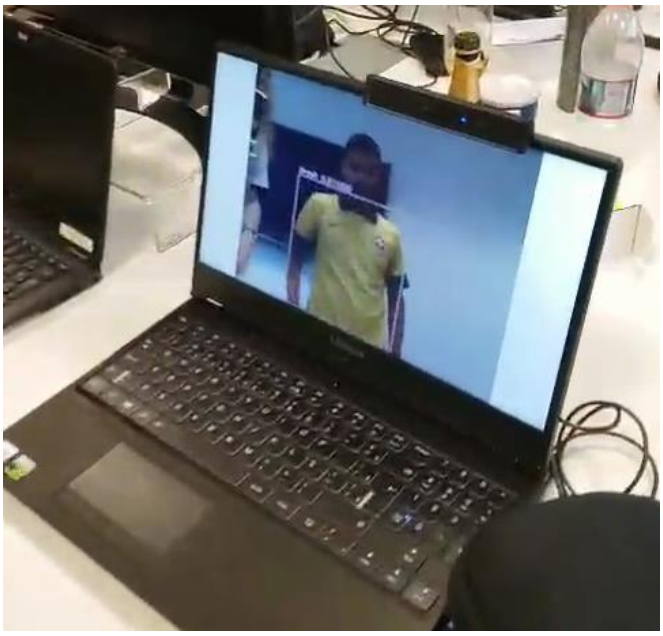
```
./darknet detector train data/obj.data cfg/yolov3-tiny-obj.cfg yolov3-tiny.conv.151 | tee /home/ubuntu/darknet/train.log
```

Seu treino irá começar, e no final dele irá criar um arquivo *train.log* no caminho */home/ubuntu/darknet* e o arquivo weights na pasta */darknet/backup*.

TESTANDO O MODELO TREINADO

Antes de testar seu modelo treinado, edite seu arquivo *.cfg* comentando as linhas **6)** e **7)**, e descomentando as linhas **3)** e **4)**:





DÚVIDAS

É natural que apareçam dúvidas no meio do caminho, sendo assim, sinta-se à vontade de me mandar um e-mail com suas perguntas, ou mesmo com suas sugestões para melhorar este documento.

E-mail: jose.amat@sas.com

Cel: +55 11 970558754

LinkedIn: https://www.linkedin.com/in/jose-amat-111225157/

Github: https://github.com/jose-amat

FONTES

https://pjreddie.com/darknet/install/

https://github.com/pjreddie/darknet

https://github.com/AlexeyAB/darknet/

https://www.learnopencv.com/training-yolov3-deep-learning-based-custom-object-detector/