Traffic Sign Classification using Convolutional Neural Network

20194316 김동휘

1. Introduction

본 과제는 Caffe 라는 Machine Learning framework을 이용하여 Traffic Sign을 분류하는 것을 목적으로한다. Caffe에서 주로 사용하는 Network 는 Convolution Neural Network로 image분류에 적합한 형태를 지니고 있다. 필자는 docker를 이용하여 Caffe 설치를 마쳤으며, 이후 dataset을 Caffe에서 사용할 수 있도록 LMDB형태로 만들었다. 만들어진 DB를 이용하여 prototxt에 정의된 Network로 훈련하여, model을 만들고, 이후 inference하는 방법으로 진행되었다.

2. Background

2.1 Docker

여러 Task 가 서로다른 dependency를 가지고 실행된다면, 하나의 OS로는 충돌을 회피할 수 없다. docker는 이를 개선하고자 새로운 서버, Container 를 이용하여 새로운 가상환경을 구축하는 것을 제공한다. 흔히 Virtual Machine과 같은 원리로 움직이나, VM과의 차이점은 Guest OS를 완전히 구축하지 않고, 필요한 부분만 추려, Docker Engine의 형태로 구현되도록 한다. 이는 필요한 만큼의 resource를 사용하고 서로 영향을 주지 않아 독립적으로 프로그램이 실행되는 느낌을 준다.

Docker에서 중요한 점은 Image이다. Image는 Container 실행에 필요한 파일과 설정값 등을 포함하고 있는 것으로, 상태값을 가지지 않고 변하지 않는다. 즉 사용자 입장에서는, 필요한 프로그램에 필요한 docker Image파일을 받아, 그에 맞는 Container를 생성해주기만 하면 된다. 본 실험에서는 Caffe github에 docker install이 제시되어 있어, 그에 맞게 설치하였다.

2.2 Caffe

Tensorflow, Pytorch와 같이, Deep learning을 구현할 수 있는 framework이다. 각 layer는 cpp기반으로 형성되어 있고, 개발자의 입장에서는 구현되어 있는 layer를 prototxt의 layer parameter만을 이용하여 Network Architecture를 형성할 수 있다.

Prototxt layer의 parameter중 top, bottom 은 layer 하위/상위 layer와 이어주는 역할이다. 정확하게는 blob라는 data가 이동하는 경로를 나타내며, blob는 python에서 생각하는 numpy와 같은 matrix형태를 가진다. 중복되는 blob 이름이 있는데 이는, 새로운 blob를 만드는 것이 아닌, 기존 blob를 변형하는 효과를 가져온다.

2.2.1 Error handling

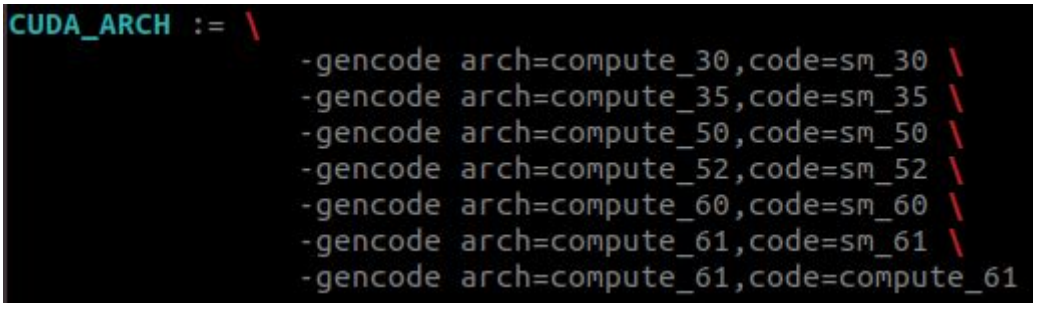
(1) can’t find hdf5.h when build caffe

command line에 **find . -type f -exec sed -i -e 's^"hdf5.h"^"hdf5/serial/hdf5.h"^g' -e**

's^"hdf5_hl.h"^"hdf5/serial/hdf5_hl.h"^g' '{}' \; 를 입력한다.

(2) NVCC src/caffe/solvers/rmsprop_solver.cu nvcc fatal : Unsupported gp architecture 'compute_20'

CUDA버전에 따른 gpu 사용에 대한 오류이다. 필자는 cuda 10.0, RTX2060으로 진행하였고, 이 경우 UDA_ARCH 에 compute_61 까지 추가하여야 한다. 또한, compute_20에 대해서는 un command 해주었다.



(3) check failed: error == cudaSuccess (2 vs. 0) out of memory batch normalize

GPU에 필요한 memory를 모두 채우지 못해 발생하는 에러이다. 이 경우에는 batch_size를 줄여 문제를 해결할 수 있었다. 기본 prototxt에서는 256 batch를 유지하였고, 64까지 줄였다.

(4) error: ImportError: No module named_tkinter

python2에서는 기본적으로 제공하지 않는 module이다. python3에서 진행하면 쉽게 해결되는 문제로 보여진다. docker에서는 기본적으로 python2를 지원하였기 때문에, tkinter이 필요한 plot에 대해서만 host OS에서 진행할 수 있도록 하였다. 간혹, matplotlib.use(“Agg”) 를 추가하면 된다는 글도 많이 보인다.

2.3 LMDB

Sysmas Lightning Memory-mapped Database 로 메모리급 효율의 데이터베이스이다. 메모리 매핑 파일을 사용하면 표준 디스크 기반 데이터베이스의 지속성을 유지하면서 순수 in-memory 데이터베이스의 읽기 성능을 가진다.

Caffe에서는 LMDB, LevelDB, hdf5 등등 여러 database를 사용하여 train data를 사용할 수 있다.

Caffe에서 LMDB를 만들 때, 조심해야할 부분이, 나중에 compute_mean을 하는 과정에서 image파일들의 경로를 command라인에서 적어야하는데, 여기에 있는 경로와 DB 속 Image 경로를 합친 경로로 진행될 수 있도록 LMDB를 만들어야 한다.

예를들면, 실제 image path를 folder/image/123.png 이라고 하자. 이 경우 LMDB 속 path 를 image/123.png라고 한다면, compute_mean의 image path는 folder 까지만 작성해야 한다.

command 명령어는 다음과 같다.
**GLOG_logtostderr=1 build/tools/convert_imageset **

**--resize_height=200 --resize_width=200 --shuffle **
/path/to/Images /path/to/label_txt /path/to/output_lmdb

3. Experiment

3.1 Data augmentation

training data에 채도, 밝기, 색감, 회전을 적용하여, 다양한 train dataset을 만드는 것이다. Machine Learning은 기본적으로 dataset의 성질을 이용하여 학습하기 때문에, 양질의 dataset은 곧 성능과 밀접하게 관련이 있다.

3.2 Batch Normalization

Network를 train 시킬 경우 각 layer를 지나면서, 학습이 불안정하게 진행된다. 그 이유는 각 layer에서 input distribution이 달라지게 되기 때문이다. 이 근본적인 원인을 잡기 위해서 각 layer에서 feature 각각에 대해서만 평균을 0, 표준편차를 1이 되도록 Normalize한 후, scale factor와 shift factor를 더해주고 이 변수들을 다시 train 시켜주도록 한다. Normalize만 진행하는 것은 data 의 bias를 무시하는 경향이 있기 때문에, scale factor를 넣어주어야 한다.

이를 training data set 전체에 진행하는 것이 아니라 mini-batch 단위로 접근하여 계산한다. Neural Network 는 보통 mini-batch 단위로 데이터를 가져와 학습을 시킨다. 보통 Normalization layer는 Activation 함수 전에 넣어주는 형태로 사용된다.

Test에서는 train에서 사용한 data를 모두 본다는 느낌으로 input들의 이동평균 및 unbiased variance estimate 의 이동평균을 계산하여 이 값을 사용한다. 추가로 구해진 scale 과 shift를 사용하여 계산되어진다.

```
layer {
  name: "norm5"
  type: "BatchNorm"
  bottom: "conv5"
  top: "norm5"
  batch_norm_param {
    use_global_stats : true
  }
  include {
    phase : TEST
  }
  param { lr_mult: 0 }
  param { lr_mult: 0 }
  param { lr_mult: 0 }
}
```

<prototxt batch norm code test>

```
layer {
  name: "norm5"
  type: "BatchNorm"
  bottom: "conv5"
  top: "norm5"
  batch_norm_param {
    use_global_stats : false
  }
  include {
    phase : TRAIN
  }
  param { lr_mult: 0 }
  param { lr_mult: 0 }
  param { lr_mult: 0 }
}
```

<prototxt batch norm code train>

```
layer {
  bottom: "norm5"
  top: "scale5"
  name: "scale5"
  type: "Scale",
  scale_param {
    bias_term: true
  }
}
```

<prototxt scale code>

3.3 Layer

bvlc_reference_caffenet 의 layer를 보면, layer의 top/bottom이 그물망 처럼 엮여 있는 것을 볼 수 있다. 이를 조금 더 직관적으로 표현하기 위해 하나의 CNN cycle에 대해 CNN-BN-relu-POOL을 거치도록 하였다.

4. Result

train을 진행하면서, loss값과 accuracy값을 받기 위해 **2>$1 | tee output.log** 를 train_caffenet.sh 의 끝에 넣어준다. 이렇게 얻어진 log 파일은 tools/extra의 shell script를 사용하여 여러 figure를 그릴 수 있다.

./plot_training_log.py.example 6 output.png input.log

plot_training_log.py.example 의 plot parameter는 아래와 같다.
 # Supported chart types:
 # 0: Test accuracy vs. Iters
 # 1: Test accuracy vs. Seconds
 # 2: Test loss vs. Iters
 # 3: Test loss vs. Seconds
 # 4: Train learning rate vs. Iters
 # 5: Train learning rate vs. Seconds
 # 6: Train loss vs. Iters
 # 7: Train loss vs. Seconds

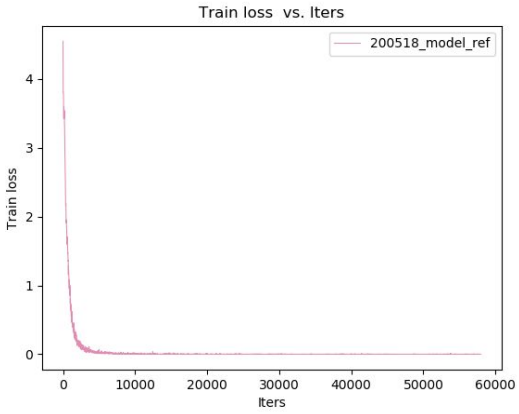
prototxt에는 3개의 file이 있는데, deploy.prototxt, train_val.prototxt, solver.prototxt 3가지 이다.

deploy.prototxt의 경우 backpropagation이 없는, 오직 inference만을 위한 forward network를 구성한다. 이 경우 train 에서 보다 더 빠른 running time을 가질 수 있다.

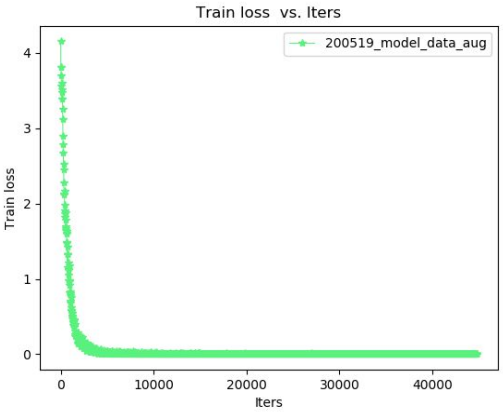
train_val.prototxt에서는 batch size와 data set의 경로 지정을 할 수 있다. 세부적인 train network 모두를 설정할 수 있다. train 속에서 layer마다 특정 parameter를 지정하여 구체적인 코딩이 가능하다. CNN의 경우 channel 수, kernel 수, Pooling 의 경우 mode, stride, FC에서는 input/output 크기 등이 있다.

solver.prototxt에서는 iteration 횟수, learning rate, GPU/CPU 등 Neural Network 라면 가지는 parameter를 가지고 있다. 학습된 weight를 저장할 수 도 있다.

기본으로 제공된 bvlc_reference_caffenet을 이용하여 학습시킨 결과와 data_augmentation을 적용시킨 결과이다. 둘다 train loss 에 대해서는 큰 차이가 없으며, test accuracy 또한 0.72, 0.73으로 유사한 결과를 가진다.



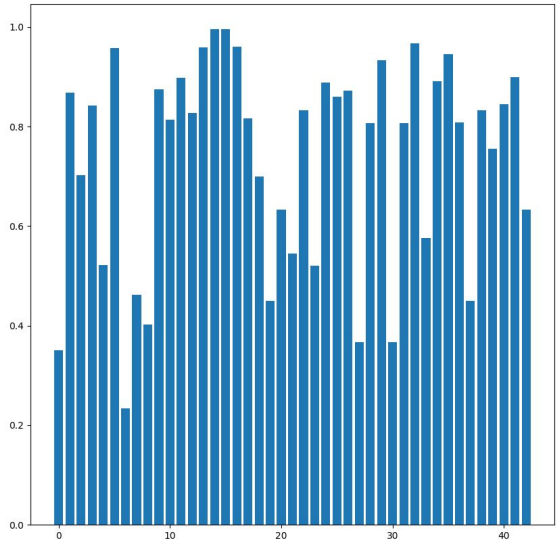
<Train loss vs Iters reference Network>



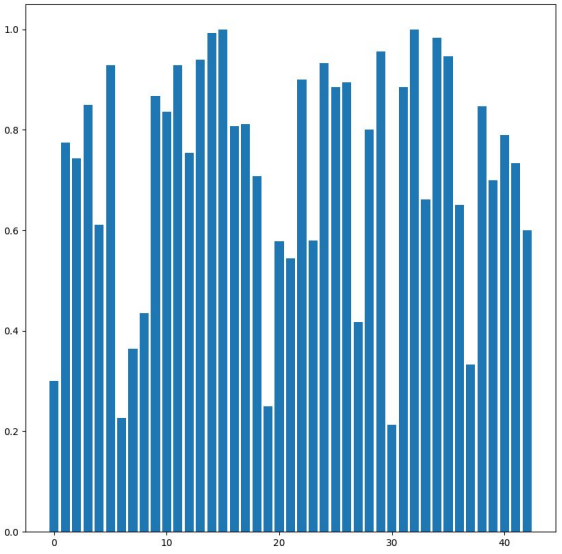
<Train loss vs Iters data augmentation>

학습된 model을 이용해 inference하는 code는 따로 만들어 주어야한다. matlab이 Ubuntu에서 없어 필자는 Caffe python inference파일을 만들어 진행하였다.

inference.py를 짤 때 조심해야하는 점은, RGB image에서, BGR로 변경해야하는 점이다. 이후 deploy 마지막 layer에서 구해지는 'prob' 을 이용하여, softmax 방법으로 추가해 image 분류를 정확히 하였는지 count하였다. 이후 총 image중 정확히 분류한 image를 세어 class마다 막대그래프를 그리도록 하였다.



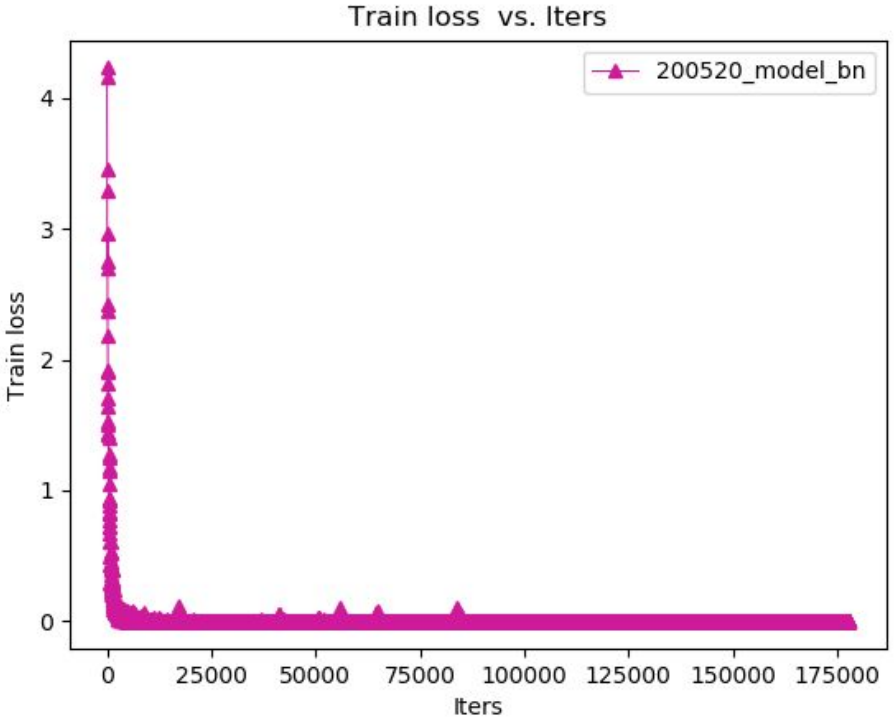
<Accuracy Per Class reference_caffenet>



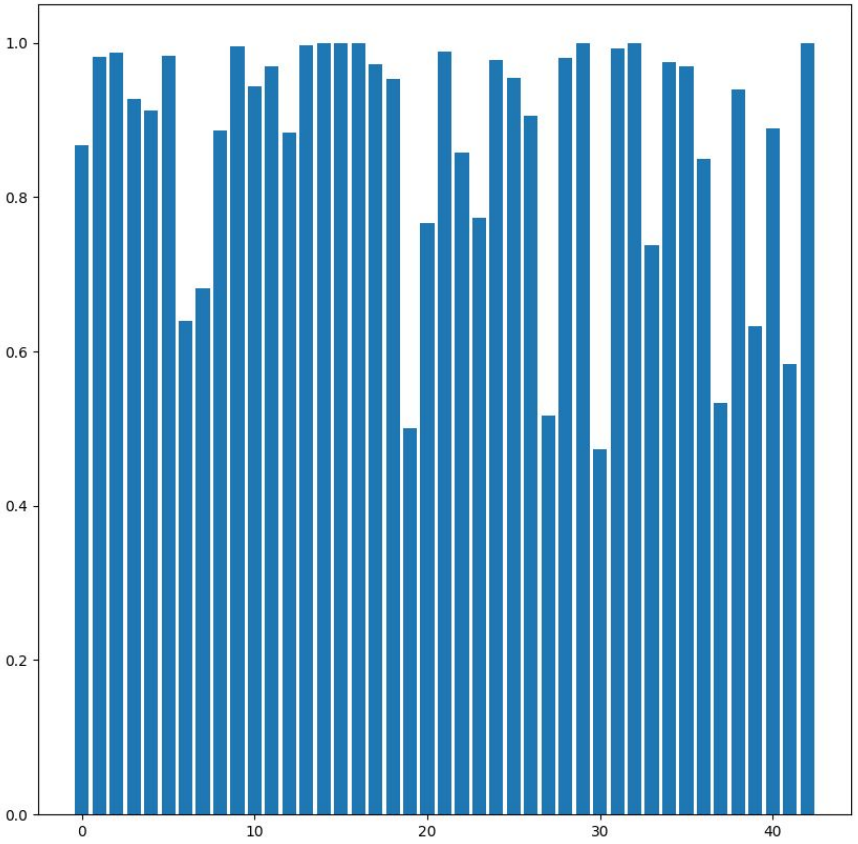
<Accuracy Per Class data_augmentation>

data augmentation 에 대해서는 큰 변화가 없는 것으로 결과가 그려진다.

data augmentation 과 CNN-BN-relu-POOL 의 cycle로 변화한 Network로 학습시킨 결과는 다음과 같다. data augmentation 과 reference Network로 73% 정도의 정확도를 가졌다면, BN 를 CNN 과 relu사이에 넣은 Network 의 정확도는 86.9% 까지 올라가는 모습을 보인다.



<Train loss vs Iters BN network>



<Accuracy Per Class BN network>

5. Test in KAIST

KI옆의 오리횡당보도 쪽 간판을 TEST data로 10장 사용하여 진행해보았다. Caffe data로 사용하기 위해 LMDB형태로 만들어주었다. 그러기 위해서는 Raw image 에서 Crop하여 아래 사진과 같이 나타내어 LMDB와 mean 값을 구하였다. 또한 image와 label을 연결해주는 test.txt 파일도 만들어주었다.

deploy에 필요한 과정을 모두 거친 후, inference를 진행해 보았다. 그 결과로는 test data 10장에 대해서 모두 class 1번으로 분류되었고, 정확도 100%를 보였다.

6. Conclusion

본 실험을 통해 Caffe를 사용하는 방법에 대해 알 수 있었다. Caffe에 맞는 LMDB를 만드는 방법도 알 수 있었고 직접 Network를 수정하여 다양한 Network를 학습할 수 있게 되었다. 또한 src파일 안에 data_augmented.cpp 파일을 넣어 내부 코드를 수정하여 Caffe에 새로운 parameter를 추가하는 방법에 대해서도 알게 되었다. 최종적으로 학습된 model을 불러와 inference하여 결과를 도출할 수 있었다. Caffe는 cpp기반으로 작성되어 다른 framework보다 빠르다고 하여 좋은 경험이 될 것이라 생각한다.

반면에 기존 Neural Network의 pytorch와 tensorflow보다 직관적으로 쉽게 Network를 구성할 수 있다는 장점이 있을 수 도 있지만, Caffe를 설치하는 것이 굉장히 힘들었던 실험 이었던 거 같다. 이 참에 docker에 대해서도 조금 더 자세히 배울 수 있게되는 자리였던거 같다.

7. Reference

[1]. https://github.com/kevinhuang06/Caffe-Data-Augmentation
[2]. https://caffe.berkeleyvision.org/tutorial/layers/batchnorm.html
[3]. https://github.com/BVLC/caffe