Korea Advanced Institute of Science and Technology
CS492 Spring 2021
Project 2

Jinwoo Hwang
jwhwang@casys.kaist.ac.kr

Due: April. 16 11:59 PM KST

Rules

- Do not copy from others. You will get huge penalty according to the University policy. Sharing of code between students is viewed as cheating.
- Every code was written and will be tested in the provided docker environment. You need to implement your code in the given environment.
- Read carefully not only this document but the comment in the skeleton code.
- This is the version 1.0 document which means not perfect. Any specifications or errors can be changed during the project period with announcement. Stay tuned to Piazza.

Introduction

This project is made to study the how the tensorflow is designed. To handle complicated computations in DNN, tensorflow adopts a *computational graph* model. The graph is composed of operation nodes that take input tensors and produce output tensors. This design is highly abstract so that users can convert their idea to the graph model easily. Also, it enables to track dependencies of input and output and to capture the parallelization point. In this project, you need to implement operation nodes. You will learn how the nodes are designed internally. A sample image is given, you can check your implementation with this input.

Project Sturcture

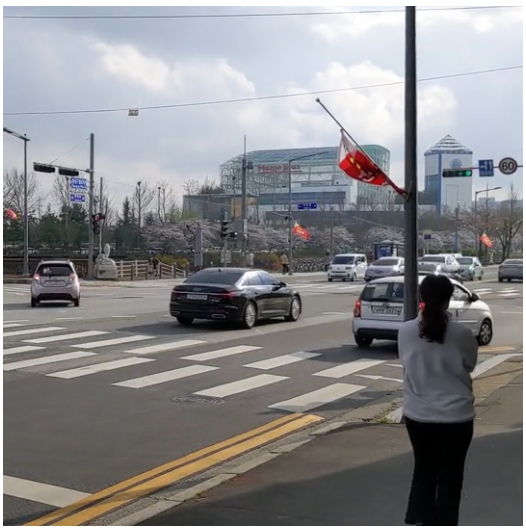
This project contains the following files:

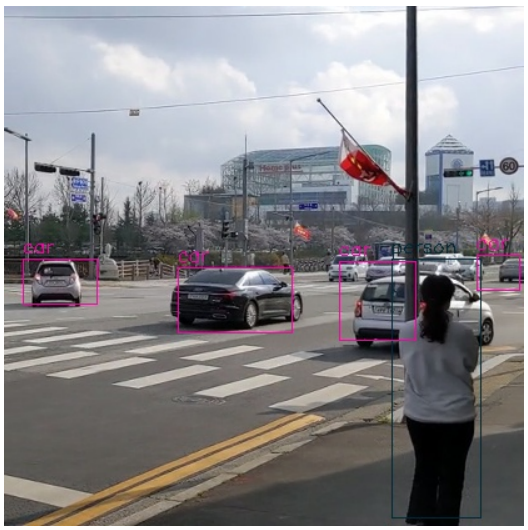
- `__init__.py`: Main source file. Following command will produce a labeled image:

  ```
  $ python3 __init__.py [path_to_input] [path_to_output]
  ```

- `yolov2tiny.py`: Source file of `YOLO_V2_TINY` class
- `dnn.py`: Source file of DNN inference engine
- `activation.py`: Supplementary file implementing activation functions.
- `sample.jpg`: Sample input image
- `output.jpg`: Sample output image
- `y2t_weights.onnx`: ONNX file for Tiny Yolo
- `test.py`: Sample test file to help you debug your implementation

Sample Input & Output





Implementation

In this project, everything is ready for you except internal computations in the nodes. Let me start with explaining the codes briefly. `__init__.py` takes an image input and inferences with `YOLO_V2_tiny` class to produce an output image. `yolov2tiny.py` builds a graph based on the YOLO v2 tiny algorithm with a DNN inferecne engine using `networkx` library. Code in `build_graph` will be similar to what you've done in the first project. There are tons of `create` methods which appends DNN nodes to the computational graph.

```
# An example of a DNN node
class Node(object):
def __init__(self, name, in_node):
    self.name = name
    self.in_node = in_node

    ... # initial setup for compute

def run(self):
    ...
    self.result ... # do computation
    ...
```

Now, look at the `dnn.py`. The class `DnnInferenceEngine` will take a graph of DNN nodes to produce its computation result. `DnnGraphBuilder` consists of several `create` functions that create and append a DNN node. We treat five types of DNN nodes; `Conv2d, BiasAdd, MaxPool2D, BatchNorm, LeakyReLU`. Your job is to implement actual computations in the DNN nodes. Every intermediate DNN node must contain `in_node` and `result`. The `in_node` is a previous node that feeds a tensor to the current node through `result`.

Lastly, the `test.py` is a sample test file to help you debug your implemenation. I have provided you two sample test case to test your convolution layer, but you may add other test cases to test other layers as well. Use the following command to run the tests,

```
python3 test.py
```

Graph Construction (20pt)

First, you need to implement a graph initializer for each node. (`__init__` method) The initializer does the followings:

- Print layer names line by line only when the construction succeed. Every layer should store its name in `self.name`. The names are automatically set at the start of creation by `DnnGraphBuilder`.
- Check dimension at constrution time. The initializing process must fail when it gets wrong arguments. In `BatchNorm`, for example, `mean`, `variance`, and `gamma` should be provided as many as the number of channels of previous `result` dimension.
- Handle `padding` option in `Conv2D` node. You may have seen padding options while using the tensorflow library. The argument must be one of `'SAME'`, `'VALID'`. You must reject abnormal padding option.
- Padding implementation for `MaxPool2D` node.

Five points per each item. Save any values in the class variables for the actual computation. The code will be graded by various graph building scenarios. Note that there will be no timeconsuming job in the initializer.

Computation (25pt)

Implement `run` method for the nodes. Five points per each node implementation. Take the result from previous node, do proper computation with class variables, and save the value at `self.result` for further computations. *Caution: The operations in this part must not contain vector operations, such as mutiplying two 2-dimensional array with * operation. Use scalar operations only.* (e.g. Instead of C = A * B, use C[i][j] = A[i][k] * B[k][j] with nested loops.)

Multiprocessing (5pt, bonus)

In the `run`, there must be nested loops which take really long time. Use `multiprocessing` library to improve the performance. You may parallelize the computation of the loops.

Handin

Your final outcome should be a single tar file that contains one python file `dnn.py`. The name of the file should be `[student_number].tar`. Upload your tar file to KLMS.