You Only Look Once: From Darknet to Tensorflow

Trinh Hoang Trieu (thtrieu@apcs.vn)

July 29th, 2016

1. Intro

Hello, this is a friendly documentation of my work. It basically bridges Darknet, Tensorflow and iOS dev.

Regarding bridging Darknet and Tensorflow, there are currently some available repos online such as *this* and *this*. Unfortunately, they only provide hard-coded routines that allows translating full/small/tiny configurations from Darknet to Tensorflow, and only for testing (forward pass). The awaited training part is still not committed.

This is understandable since building the loss op of YOLO in Tensorflow is not a trivial task. Fortunately, the scripts provided in this work completed the training part and some more. Namely, we are now able to translate any configuration (old and new) specified in a Darknet-styled config file (very much alike the prototxt in Caffe) into Tensorflow graph. We are also able to train the graph in GPU/CPU mode and save the trained weights to a protobuf object that can be used in C++ interface.

Also note that the checkpoint method in Python API is not supported in C++ API, workarounds I found online give .pb files that is 2x larger than necessary, while this script produces exactly what is needed.

2. How to use it

2.1 Parsing the annotations

Skip this if you are not training or fine-tuning anything.

The first thing to do is specifying the classes you want to work with, write them down in the `labels.txt` file. For example, if you want to work with only 3 classes `tvmonitor`, `person`, `pottedplant`; edit `labels.txt` as follows

```
tvmonitor
person
pottedplant
```

Then run `clean.py` to parse xml files in the annotation folder (according to what has been specified in `labels.txt`)

```
python clean.py /path/to/annotation/folder
# the default path is ../pascal/VOCdevkit/ANN
```

This will print some stats on the parsed dataset to screen. Parsed bounding boxes and their associated classes is stored in `parsed.yolotf`.

2.2 Design the net

Skip this if you are working with one of the three original configurations since they are already there.

In this step you create a configuration `yolo-XX.cfg` and put it inside `./configs/`. Take a look at some of the available configs there to know the syntax.

Note that these files, besides being descriptions of the net structures, also store technical specifications that is read by Darknet framework. This Tensorflow source code, therefore, ignore these Darknet specifications. `yolo-3c.cfg` is an example without these redundant specifications.

2.2 Initialize weights

Skip this if you are working with one of the three original configurations since the `.weights` files are already there.

Now as you have already specified the new configuration, next step is to initialize the weights. In this step, it is reasonable to recollect a few first layers from some trained configuration before randomly initialize the rest. `makew.py` does exactly this.

```
# Recollect weights from yolo-tiny.weights to yolo-3c.weights
python makew.py tiny 3c
```

The script prints out which layers are recollected and which are randomly initialized. The recollected layers are a few first ones that are identical between two configurations. In case there is no such layer, all the new net will be randomly initialized.

After all this, `yolo-3c.weights` is created. Bear in mind that unlike `yolo-tiny.weights`, `yolo-3c.weights` is not yet trained.

2.3 Flowing the graph

From now on, all operations are performed by `tensor.py`.

```
# Have a look at its options
python tensor.py --h
# Forward all images in ./data using tiny yolo and 100% GPU usage
python tensor.py --test data --model tiny --gpu 1.0
# The results are stored in results/
```

Training the new configuration:

```
python tensor.py --train --model 3c --gpu 1.0
```

During training, the script will occasionally save intermediate results into two files, one is a Tensorflow checkpoint, stored in `./backup/`, one is a binary file in Darknet style, stored in `./binaries/`. Only the 20 most recent pairs are kept, you can change this number in the `keep` option, if `keep = 0`, no intermediate result is omitted.

To resume, use `--load` option, it essentially parse `./backup/checkpoint` to get the most recent save and load it before doing any next operation, either training or testing.

```
# To resume the most recent checkpoint for training
python tensor.py --train --model 3c --load
# To run testing with the most recent checkpoint
python tensor.py --notrain --model 3c --load
# Without the --load option, you will be using the untrained yolo-3c.weights
# Fine tuning tiny yolo from the original one
python tensor.py --train --model tiny --noload
```

2.4 Migrating the model to C++ and Objective-C++

Now this is the tricky part since there is no official support for loading variables in C++ API. Some suggest assigning the trained weights as constants into the graph and save it down as a `.pb` (protobuf) file *like this*. However this will double the necessary size of this file, which is very undesirable in, say, building mobile applications.

To avoid this, one would have to build the graph all over again with all Variables replaced by Constants (we wouldn't train any Deep Learning model on mobile device any time soon). Unfortunately, since there is no Variable in this new graph, Tensorflow does not allow the convenient checkpointing, so one would need to resort to `./binaries/` while building this constant graph.

In short, in order to produce a protobuf graph with optimal size for use in C++ and Objective-C++ API, one would need to use `.weights` files stored in `./binaries/`. They are **important**

```
## Saving the lastest checkpoint to protobuf file
python tensor.py --model 3c --load --savepb
```

For further usage of this protobuf file, please refer to the official documentation of Tensorflow on C++ API *here*. To run it on the iOS application, simply add the file to Bundle Resources and update the path to this file inside source code.

That's all!

3. Source code

In this part, I will discuss further into details and present design choices of the source code. Skip this if you are not concerned about improving the source code.

All python scripts are

```
clean.py # parsing xml annotations
makew.py # initialize weights
tensor.py # main script
./configs/process.py # .cfg parser
box.py # all geometry goes here
Drawer.py # pre-process, post-process images
Data_helper.py # Use parsed.yolotf to yield minibatches of placeholders
Yolo.py # Use the cfg parser to parse .weights files into raw YOLO obj
TFnet.py # Use the parsed .weights object to build the TF graph
```

In next parts, I will present the layout of each of these scripts.

3.1 `tensor.py`

This script essentially collects all options into object `FLAGS` and pass it on to other subroutines

3.2 `Yolo.py`

This script contains the class definition of `Yolo`, which calls `./configs/process.py` upon its initialization to parse the required `.cfg` config, so that it knows the structure of the corresponding `.weights` file. After that, it dissects this `.weights` file accordingly and store all the parameters into the `layer` attribute.

```
...
class YOLO(object):
    layers = [] # contains weights
    S = int() # grid size (originally 7)
    model = str() # model name
    def __init__(self, model):
        # parse 'labels.txt'
        ...
        # parse the config file into self.layers
        self.build(model)
        ...
        # load the .weights file according to information obtained in self.layers
        self.loadWeights(weight_file)
        ...

    def build(self, model):
        # import the cfg parser
        from configs.process import cfg_yielder
        # parse the config file into self.layers
        layers = cfg_yielder(cfg)
        ...

    def loadWeights(self, weight_path):
        # Read bytes array from the .weight file, using information
        # obtained in self.layers
        for i in range(self.layer_number):
            ...

        # Reshape these arrays into appropriate sized tensors
        # i.e. convolution kernel into 4D tensor
        # dense layer into 2D tensor
        for i in range(self.layer_number):
            ...
```

3.3 `Data_helpers.py`

Since Tensorflow does not support member assignment, calculating the loss is very difficult. One must first figure out what exactly is the tensorized operations that carry out the loss calculation. Then decide which among these tensorized operations should be implemented as `numpy tensors` (allow member assignmen) and which as `tensorflow tensors`. This script accounts for the `numpy` part, while the next section introduces the `tensorflow` part.

Basically, this script provide a yielder that does the following:

- Read `parsed.yolotf`, which essentially contains a list of objects, each represent information of a training example
- Shuffle the list, divide the list into minibatches
- For each minibatch, it runs through all objects and does the following:
  - Read the corresponding image, apply random scale/translation and convert the image into a tensor of size `448 x 448 x 3`.

  - Read the corresponding bounding boxes and their associated classes in this image, encode this information into 11 `numpy tensors` as material readily for the loss evaluation. The author believe this is the most efficient way to do it. Others may differ, this may be the part you want to consider improving first.

- Concatenate the image tensors into a tensor named `x_batch` of size `batchSize x 448 x 448 x 3`. It also concatenates all the 11-tensor groups element-wise into list `datum` of 11 tensors before yielding the pair `(x_batch, datum)`.

```
def shuffle(train_path, file, expectC, S, batch, epoch):
    ...
    for i in range(epoch):
        # shuffle the training data
        shuffle_idx = np.random.permutation(np.arange(size))
        ...
        # interate over each batch in one epoch
        for b in range(batch_per_epoch):
            ...
            # iterate over each image in the batch
            for j in range(start_idx, end_idx):
                ...
                # read image, apply scale/translation
                img, allobj = crop(path, allobj)
                ...

                # add this image to x_batch
                x_batch += img
                ...

                # 11 numpy tensors: material for L2-loss evaluation
                new = [
                    [probs], [confs1], [confs2], [coord],
                    [upleft], [botright],
                    [proid], [conid1], [conid2], [cooid1], [cooid2]
                ]
                # concatenate 11-tensor groups element-wise
                for i in range(len(datum)):
                    datum[i] = np.concatenate([datum[i], new[i]])
                ...

            # concatenate all images into a single tensor
            x_batch = np.concatenate(x_batch, 0)
            # yield the pair
            yield (x_batch, datum)
        ...
```

3.4 `TFnet.py`

This script contains a class definition of the class `SimpleNet`, it stores the tensorflow graph and methods operate on this graph, including the L2-loss evaluation.

```
class SimpleNet(object):
    def __init__(self, yolo, FLAGS):
```

```
        # obtain some parameter from yolo, an object of class Yolo
        # which stores the parsed .weights file
        self.model = yolo.model
        self.S = yolo.S
        self.labels = yolo.labels
        ...

        # build the graph layer by layer accordingly
        for i in range(yolo.layer_number):
            ...
            # Notice the difference when option
            # --savepb is used/not used:
            if FLAGS.savepb:
                b = tf.constant(l.biases)
                w = tf.constant(l.weights)
            else:
                b = tf.Variable(l.biases)
                w = tf.Variable(l.weights)
            ...

        # the output tensor
        self.out = now

    def setup_meta_ops(self, FLAGS):
        # Build meta-ops: including loss op and training op
        # (if option --train is used), create the Session with
        # specifications stored inside FLAGS, create the Saver,
        # load checkpoint / save protobuf file if necessary.
        if FLAGS.train: self.decode() # add loss and train op
        if FLAGS.savepb: self.savepb()
        ...

        self.sess.run(tf.initialize_all_variables())
        ...

    def savepb(self, name):
        # Save the constant graph into a protobuf file
        ...

    def to_constant(self, inc = 0):
        # Save the current graph into a .weights file
        ...

    def decode(self):
        # This function uses all 11 tensors yielded inside datum,
        # performs appropriate calculations with the output tensor
        # self.out to evaluate L2-loss of the current minibatch

        # Set up 11 place-holders
        self.true_class = tf.placeholder(tf.float32, [None, SS * self.C])
        ...
        # Calculate the loss op
        self.loss = tf.pow(self.out - true, 2) # L-2 Loss
```

```
        self.loss = tf.mul(self.loss, idtf) # Adding Weight terms
        self.loss = tf.reduce_sum(self.loss, 1) # Sum over all terms
        self.loss = .5 * tf.reduce_mean(self.loss) # Average over the minibatch
        ...
        # Set up the train op, RMSProp by default
        optimizer = tf.train.RMSPropOptimizer(self.learning_rate)
        gradients = optimizer.compute_gradients(self.loss)
        self.train_op = optimizer.apply_gradients(gradients)
        ...

    def train(self, train_set, annotates, batch_size, epoch_num):
        # import the mini-batch yielder
        from Data_helper import shuffle
        batches = shuffle(train_set, annotates, batch_size, epoch_num)
        for i, batch in enumerate(batches):
            ...
            # obtain x_batch and datum
            x_batch, datum = batch
            # feed x_batch as input
            # and datum as 11 placeholders' value
            feed_dict = {
                self.inp : x_batch,
                self.drop : .5,
                self.true_class : datum[0],
                self.confs1 : datum[1],
                self.confs2 : datum[2],
                self.true_coo : datum[3],
                self.upleft : datum[4],
                self.botright : datum[5],
                self.class_idtf : datum[6],
                self.conid1 : datum[7],
                self.conid2 : datum[8],
                self.cooid1 : datum[9],
                self.cooid2 : datum[10],
            }
            # Run the foward pass that includes train_op and loss op in the target
            _, loss = self.sess.run([self.train_op, self.loss], feed_dict)

            # Save checkpoint and binaries
            self.saver.save(self.sess, 'backup/model-{}'.format(self.step+i+1))
            self.to_constant()
            ...

    def predict(self, FLAGS):
        # Import the image processor, read images into a feed_dict
        from Drawer import crop, draw_predictions
        ...
        # Forward pass
        out = self.sess.run([self.out], feed_dict)
        ...
        # Draw bounding boxes
        for i, prediction in enumerate(out[0]):
            draw_predictions(prediction)
```