[2018년 여름]

Tensorflow를 이용한 Deep Learning의 기초

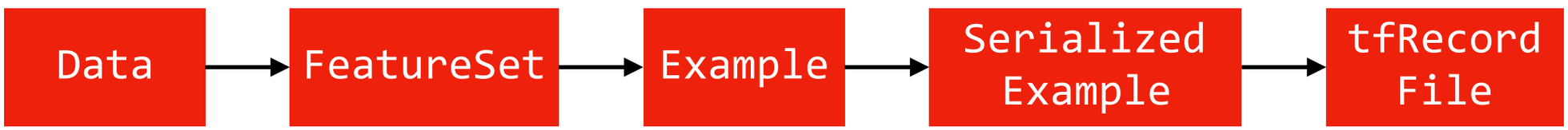
제5강: Data API

TFRecordDataset

- **Tensorflow의 권장 데이터 포맷**
- **Protocol Buffer 포맷으로 데이터를 직렬화하여 저장.** 프로토콜 버퍼는 구글이 개발한 구조화된 (structured) 데이터를 직렬화(serialize)하는 방식이다. 통신이나 데이터 저장을 위해 사용된다. (알기 쉬운 유사한 예로 XML의 binary version을 생각하면 된다.)

TFRecord File 만들기 절차

1. `tf.python_io.TFRecordWriter`를 이용하여 `tfrecord` 파일을 open한다.
2. `tfrecord`에 쓰기 전에 이미지와 라벨 데이터를 적절한 데이터 타입(byte, int, float)으로 변환한다.
3. 데이터를 `tf.train.Feature` 객체로 변환한다.
4. `tf.train.Example`을 이용하여 Example Protocol Buffer를 생성하고, `serialize()` 함수를 이용하여 직렬화한다.
5. 직렬화된 example을 쓴다.



Convert To TFRecord File

generate_tfrecords.py

```
from random import shuffle
import glob
import cv2
import numpy as np
import tensorflow as tf
import sys

ORI_IMG_HEIGHT = 160
ORI_IMG_WIDTH = 160
NUM_CHANNEL = 3

# read addresses and labels from the 'train' folder
img_file_path = '../animal_images/*/*.*'
addrs = glob.glob(img_file_path)
labels = []
for addr in addrs:
    if 'cat' in addr:
        label = 0
    elif 'cow' in addr:
        label = 1
    elif 'dog' in addr:
        label = 2
    elif 'pig' in addr:
        label = 3
    elif 'sheep' in addr:
        label = 4
    labels.append(label)
```

이미지 사이즈를 좀 크게 해보았다.

모든 이미지 파일 경로명의 리스트

라벨 리스트

```
# to shuffle data
c = list(zip(addrs, labels))
shuffle(c)
addrs, labels = zip(*c)

# Divide the data into 60% train, 20% validation, and 20% test
train_addrs = addrs[0:int(0.6 * len(addrs))]
train_labels = labels[0:int(0.6 * len(labels))]

val_addrs = addrs[int(0.6 * len(addrs)):int(0.8 * len(addrs))]
val_labels = labels[int(0.6 * len(addrs)):int(0.8 * len(addrs))]

test_addrs = addrs[int(0.8 * len(addrs)):]
test_labels = labels[int(0.8 * len(labels)):]

def load_image(addr):
    img = cv2.imread(addr)
    img = cv2.resize(img, (ORI_IMG_HEIGHT, ORI_IMG_WIDTH),
                                    interpolation=cv2.INTER_CUBIC)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = img.astype(np.float32)
    return img
```

shuffling한다.

6:2:2의 비율로 분할한다.

여기까지는 지금까지 여러 번 등장했던 코드이다.

Convert To TFRecord File
generate_tfrecords.py

- TFRecord 파일에 저장하기 위해서 데이터는 int64, float, 혹은 그냥 bytes 리스트로 변환해야 한다. 이 변환을 해주기 위한 함수들이다.
- int64, float, 혹은 byte 리스트로 변환된 데이터는 다시 Feature proto message로 변환한다.

```
def _int64_feature(value):
  return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
```

```
def _bytes_feature(value):
  return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))
```

Feature는 proto message인데, 그냥 int64_list, bytes_list,
혹은 float_list를 필드(field)로 가지는 클래스로 생각하면 된다.
즉 이 문장은 Int64List, BytesList, 혹은 FloatList 타입의 데이터를
저장하는 Feature객체를 생성하는 것으로 이해하면 된다.

Convert To TFRecord File
generate_tfrecords.py

이미지 경로 리스트 / 라벨 값 리스트 / 생성할 TFRecord 파일의 이름

```
def convert_and_save(feature_addrs, labels, filename):

    writer = tf.python_io.TFRecordWriter(filename)

    for i in range(len(feature_addrs)):

        img = load_image(feature_addrs[i])

        # Create a feature map
        feature = {'label': _int64_feature(labels[i]),
                   'image': _bytes_feature(img.tostring())}

        # Create an example protocol buffer
        example = tf.train.Example(features=tf.train.Features(feature=feature))

        # Serialize to string and write on the file
        writer.write(example.SerializeToString())

    writer.close()
```

TFRecord 파일을 생성

이미지 파일 로드

라벨과 이미지를 각각 Feature객체로 변환한 후 적절한 키(key)를 대응시켜 map을 만든다.

Convert To TFRecord File

generate_tfrecords.py

```
convert_and_save(train_addrs, train_labels, 'train.tfrecords')
convert_and_save(val_addrs, val_labels, 'validation.tfrecords')
convert_and_save(test_addrs, test_labels, 'test.tfrecords')
```

Importing Data using DataSet API

장점 혹은 필요성

- 데이터의 크기가 오직 파일 크기(즉 디스크 용량)에 의해서만 제한됨
- 많은 참조 코드들이 DataSet API를 이용해서 작성되어 있음
- Estimator등의 high-level API를 이해하기 위해서 필요

TFRecord 파일로부터 읽어올 **각각의** 데이터에 대해서 수행할 일들을 모아둔다.

```
def _parse_function(example_proto):

    features = {'label': tf.FixedLenFeature([], tf.int64),
                'image': tf.FixedLenFeature([], tf.string)}

    parsed_features = tf.parse_single_example(example_proto, features)

    label = tf.cast(parsed_features['label'], tf.int32)

    image = tf.decode_raw(parsed_features['image'], tf.float32)


    image = tf.reshape(image, [ORI_IMG_HEIGHT, ORI_IMG_WIDTH, NUM_CHANNEL])
    image = tf.image.resize_images(image, [IMG_HEIGHT, IMG_WIDTH])

    image = tf.image.per_image_standardization(image)

    return image, label
```

먼저 TFRecord 파일에 저장된 각 record들의 **구조**를 이렇게 명시한다.

record를 파싱한다.

int64로 저장되어 있으므로 int32로 변환한다.

byte sequence로 저장된 이미지를 np array로 복구

픽셀값들을 zero mean, unit norm 을 가지도록 변환 (일종의 정규화이며 해야하는 일은 아니고 그냥 해보는 것임)

handle_dataset.py

- 학습 데이터가 부족할 때 흔히 사용하는 방법의 하나는 각각의 이미지를 random하게 distort해서 새로운 이미지를 만들어 내는 것이다. 가장 간단한 방법은 이미지의 일부분을 random하게 crop하는 것이다.

```
def _distorted_parse_function(example_proto):

    features = {'label': tf.FixedLenFeature([], tf.int64),
                'image': tf.FixedLenFeature([], tf.string)}
    parsed_features = tf.parse_single_example(example_proto, features)
    label = tf.cast(parsed_features['label'], tf.int32)
    image = tf.decode_raw(parsed_features['image'], tf.float32)
    image = tf.reshape(image, [ORI_IMG_HEIGHT, ORI_IMG_WIDTH, NUM_CHANNEL])

    image = tf.random_crop(image, [IMG_HEIGHT, IMG_WIDTH, NUM_CHANNEL])

    image = tf.image.per_image_standardization(image)

    return image, label
```

IMG_HEIGHT와 IMG_WIDTH는 128로 가정하였다.

handle_dataset.py

읽어올 TRRecord 파일명의 리스트. 즉 파일이 하나 이상 이어도 된다는 의미이다.

```
def get_input(train_filenames, valid_filenames, test_filenames):

    dataset_train = tf.data.TFRecordDataset(train_filenames)

    dataset_train = dataset_train.map(_distorted_parse_function)

    dataset_train = dataset_train.shuffle(buffer_size=10000)

    dataset_train = dataset_train.batch(50)

    dataset_train = dataset_train.repeat()
```

DataSet을 만든다.

DataSet의 각각의 record들은 _distorted_parse_function에 의해서 처리된다.

record들은 10,000개씩 버퍼에 읽혀져 온 후 shuffle된다.

record들은 50개씩 batch로 묶여진다.

repeat 횟수를 지정할 수 있다. 지정하지 않으면 무한 반복된다.

- 우리는 이 DataSet에 대한 Iterator를 만들 것이다. 이 Iterator에 대해서 next를 수행할 때마다 실제 데이터가 반환된다.
- 즉 DataSet은 그 자체가 데이터 집합이라기 보다는 이렇게 우리가 지정한 규칙에 따라 데이터를 반환해주는 기능을 가진 어떤 객체이다.

handle_dataset.py

- Validation 데이터와 test 데이터에 대해서도 유사하게 DataSet 객체를 생성하고 설정한다.

```
dataset_valid = tf.data.TFRecordDataset(valid_filenames)
dataset_valid = dataset_valid.map(_parse_function)
dataset_valid = dataset_valid.batch(50)
dataset_valid = dataset_valid.repeat()
```

shuffling은 불필요하므로 하지 않는다.

```
dataset_test = tf.data.TFRecordDataset(test_filenames)
dataset_test = dataset_test.map(_parse_function)
dataset_test = dataset_test.batch(50)
dataset_test = dataset_test.repeat()
```

Test와 Validation 데이터는 distort할 필요가 없으므로 그냥 _parse_function을 적용한다.

handle_dataset.py

- DataSet에 대해서 만들 수 있는 Iterator는 one-shot, initializable, reinitializable, 그리고 feedable 타입의 4 유형이다.
- 우리처럼 training 중간 중간에 validation을 하려면 reinitializable iterator가 적당하다. 다른 iterator의 사용법은 여기를 참조.

Reinitializable iterator는 이렇게 DataSet의 structure만을 이용해서 만든다.

```
iterator = tf.contrib.data.Iterator.from_structure(dataset_train.output_types,
                                                   dataset_train.output_shapes)
```

```
image_op, labels_op = iterator.get_next()
```

image_op와 labels_op는 이미지와 라벨 값이 아니다. iterator에 대해서 get_next를 수행하는 op이다. 즉 tensorflow graph의 노드이다. 나중에 session을 만들고 이 op들을 run하고 그 결과를 fetch하면 이미지와 라벨 값이 반환된다.

```
training_init_op = iterator.make_initializer(dataset_train)
valid_init_op = iterator.make_initializer(dataset_valid)
test_init_op = iterator.make_initializer(dataset_test)
```

iterator가 return할 값의 source를 변경할 수 있게 해주는 initializer들이다.

```
return image_op, labels_op, training_init_op, valid_init_op, test_init_op
```

hyperparameters.py

- Hyperparameter들을 하나의 파일로 모아 두었다.

```
ORI_IMG_HEIGHT = 180
ORI_IMG_WIDTH = 180
IMG_HEIGHT = 128
IMG_WIDTH = 128
NUM_CHANNEL = 3
NUM_CLASS = 5
IMAGE_DIR_BASE = '../animal_images'
LOG_DIR = 'logs'
BATCH_SIZE = 50
NUM_EPOCH = 50
BATCH_PER_EPOCH = 12

NUM_VAL_BATCH = 4
NUM_TEST_BATCH = 4
```

Build Network: mycnn.py

- cnn01.py과 동일한 CNN을 단지 Python Class로 구현해보았다.

```
import tensorflow as tf
from hyperparameters import *
import os

class MyNet:
    def __init__(self, images_batch, labels_batch, keep_prob, num_class):
        self.build(images_batch, labels_batch, keep_prob, num_class)
        self.keep_prob = keep_prob

    def con_layer(self, input_tensor, filter_size, in_channels, out_channels, layer_name):
        with tf.variable_scope(layer_name):
            filt = tf.get_variable(name="filter", shape=[filter_size, filter_size,
                                   in_channels, out_channels], dtype=tf.float32)
            bias = tf.get_variable(name='bias', shape=[out_channels], dtype=tf.float32)
            pre_activate = tf.nn.conv2d(input_tensor, filt, [1, 1, 1, 1], padding='SAME') + bias
            activations = tf.nn.relu(pre_activate)
            return activations

    def max_pool_2x2(self, x):
        return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

    def dense_layer(self, input_tensor, input_dim, output_dim, layer_name, act=True):
        with tf.variable_scope(layer_name):
            weights = tf.get_variable(name="weight", shape=[input_dim, output_dim])
            biases = tf.get_variable(name='bias', shape=[output_dim])
            preactivate = tf.matmul(input_tensor, weights) + biases
            if act:
                return tf.nn.relu(preactivate)
            return preactivate
```

Build Network: mycnn.py

```
def build(self, images_batch, labels_batch, keep_prob, num_class):
    input_tensor_shape = images_batch.get_shape()[1:]
    conv1 = self.con_layer(images_batch, 5, input_tensor_shape[2], 32, 'con_layer1')
    h_pool1 = self.max_pool_2x2(conv1)
    conv2 = self.con_layer(h_pool1, 5, 32, 64, 'con_layer2')
    h_pool2 = self.max_pool_2x2(conv2)
    fc_size = input_tensor_shape[0]//4*input_tensor_shape[1]//4*64
    h_pool2_flat = tf.reshape(h_pool2, [-1, fc_size])
    fc1 = self.dense_layer(h_pool2_flat, fc_size, 1024, 'dense1')
    h_fc1_drop = tf.nn.dropout(fc1, keep_prob)
    y_pred = self.dense_layer(h_fc1_drop, 1024, num_class, 'dense2', act=False)
    self.loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
                        logits=y_pred, labels=labels_batch))
    tf.summary.scalar('loss', self.loss)
    self.prediction = tf.argmax(y_pred, 1, output_type=tf.int32, name='prediction')
    correct_prediction = tf.equal(self.prediction,  labels_batch)
    self.accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    tf.summary.scalar('accuracy', self.accuracy)
    self.train_op = tf.train.AdamOptimizer().minimize(self.loss)
    self.summary = tf.summary.merge_all()
```

Train: cnn02.py

```
import tensorflow as tf
from hyperparameters import *
import mycnn
import handle_dataset
import os

best_val_acc = 0.0
save_path = LOG_DIR + os.sep + 'model.ckpt'


def train():
    step = 0
    for epoch in range(NUM_EPOCH):
        sess.run(training_init_op)

        for _ in range(BATCH_PER_EPOCH):
            _, loss_val, accuracy_val, _summary = sess.run([mynet.train_op,
             mynet.loss, mynet.accuracy, mynet.summary], feed_dict={keep_prob: 0.5})
            step += 1
            summary_writer.add_summary(_summary, step)

        print('Train Loss and Accurary after {}-th epoch: {} {}'.format(epoch,
              loss_val, accuracy_val))
        validate(epoch)
```

epoch단위로 반복한다.

데이터를 읽어주는 iterator의 source를 training data로 설정한다.

Validate and Test: cnn02.py

```
def validate(epoch):
    global best_val_acc

    sess.run(vaid_init_op)

    sum_accuracy = 0.0
    for _ in range(NUM_VAL_BATCH):
        accuracy_val = sess.run(mynet.accuracy, feed_dict={keep_prob: 1.0})
        sum_accuracy += accuracy_val
    val_accuracy = sum_accuracy / NUM_VAL_BATCH
    print('Validation Accuracy after {}-th epoch is {}'.format(epoch, val_accuracy))
    if val_accuracy > best_val_acc:
        best_val_acc = val_accuracy
        saver.save(sess, save_path, global_step=epoch)
        print('Weights are saved to ' + save_path)
```

데이터를 읽어주는 iterator의 source를 validation data로 설정한다.

```
def test():

    sess.run(test_init_op)

    sum_accuracy = 0.0
    for _ in range(NUM_TEST_BATCH):
        accuracy_test = sess.run(mynet.accuracy, feed_dict={keep_prob: 1.0})
        sum_accuracy += accuracy_test
    test_accuracy = sum_accuracy / NUM_TEST_BATCH
    print('Test Accuracy  is {}'.format(test_accuracy))
```

데이터를 읽어주는 iterator의 source를 test data로 설정한다.

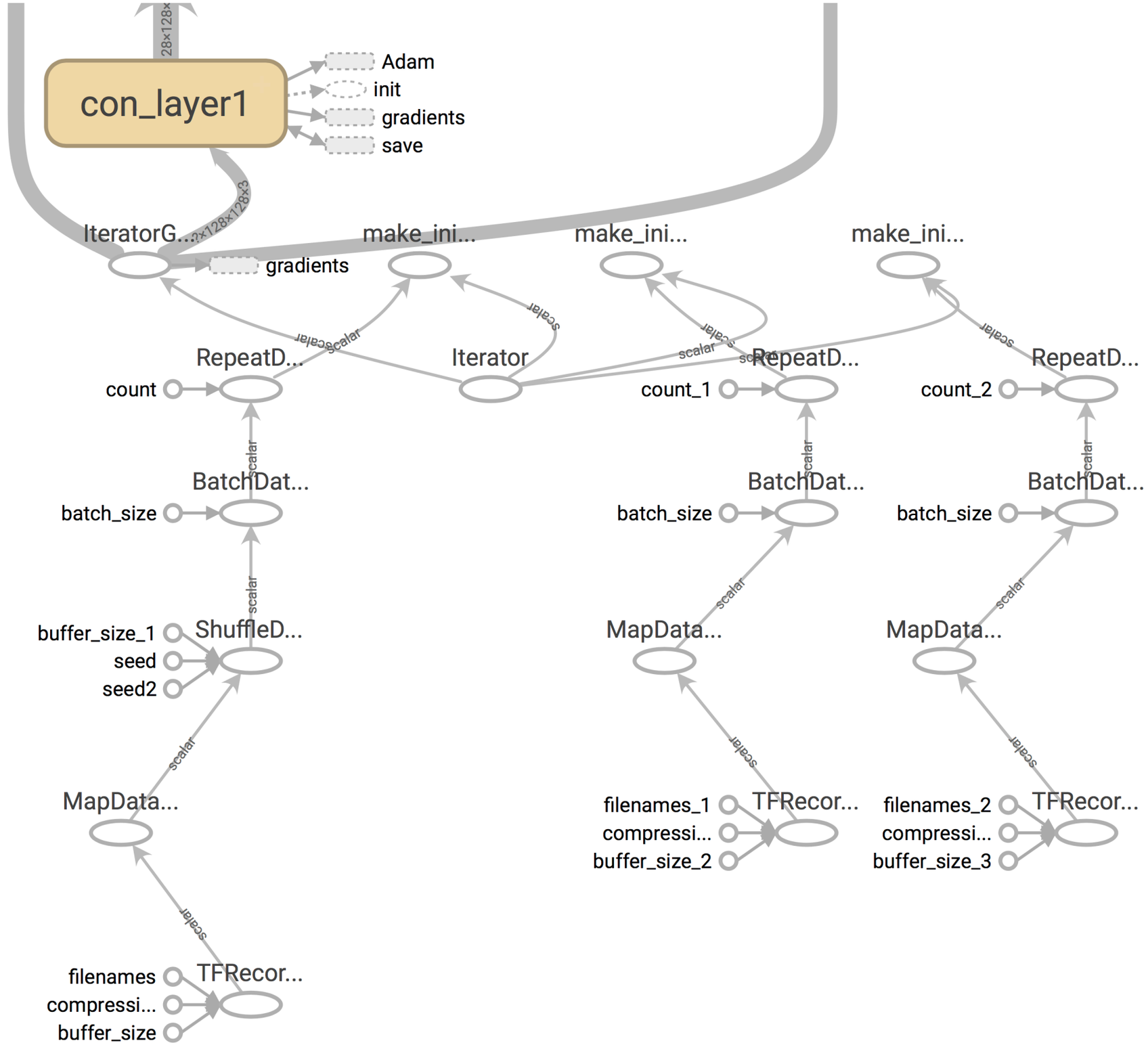
```
train_filenames = ['train.tfrecords']
valid_filenames = ['validation.tfrecords']
test_filenames = ['test.tfrecords']

images_batch, labels_batch, training_init_op, vaid_init_op, test_init_op =
          handle_dataset.get_input(train_filenames, valid_filenames, test_filenames)
keep_prob = tf.placeholder(dtype=tf.float32, name='keep_prob')

mynet = mycnn.MyNet(images_batch, labels_batch, keep_prob, NUM_CLASS)

saver = tf.train.Saver()
sess = tf.Session()
summary_writer = tf.summary.FileWriter(LOG_DIR, sess.graph)
sess.run(tf.global_variables_initializer())

train()
print('Training Finished....')
summary_writer.close()
test()
```


28×128×
con_layer1
Adam
init
gradients
save
IteratorG...
2×128×128×3
gradients
make_ini...
make_ini...
make_ini...
scalar
scalar
scalar
scalar
scalar
scalar
RepeatD...
Iterator
RepeatD...
RepeatD...
count
count_1
count_2
scalar
scalar
scalar
BatchDat...
BatchDat...
BatchDat...
batch_size
batch_size
batch_size
scalar
scalar
scalar
buffer_size_1
seed
seed2
ShuffleD...
MapData...
MapData...
scalar
scalar
scalar
MapData...
filenames
compressi...
buffer_size
TFRecor...
filenames_1
compressi...
buffer_size_2
TFRecor...
filenames_2
compressi...
buffer_size_3
TFRecor...