```python
#!/usr/bin/python3.6

### Kevin Sheng
### ECE471 Selected Topics in Machine Learning - Midterm Project

# "Learning Sparse Neural Networks through L_0 Regularization"
#       by Christos Louizos, Max Welling, Diederik P. Kingma
#   https://arxiv.org/pdf/1712.01312.pdf
#   https://github.com/AMLab-Amsterdam/L0_regularization
#
# Reproducing part of Table 1: using L0 regularization to prune LeNet-5-Caffe
#   Pruning the original 20-50-800-500 architecture to about 9-18-65-25 with 99% accuracy.
#   The important part is the level of shrinkage achieved in the computationally expensive
#   fully connected layers.
#
# Results:
#   Deterministic pruned architecture after 110001 global steps: 14-19-36-21
#   Test accuracy: 0.9872999787330627
#   Test loss: 0.30946531891822815
#
# Example of pruning at train time (one arbitrary step):
#   112599/1100000 [15:38<2:17:10, 119.97it/s, epoch=204,
#           neurons=[14.0, 19.0, 35.0, 21.0], t_acc=0.999, t_loss=0.223, v_acc=0.986]

import os
import argparse
from tqdm import tqdm
import numpy as np
import tensorflow as tf
from tensorflow.contrib.learn.python.learn.datasets import mnist
import blocks

class Model():
    def __init__(self, data, train=False, save=False, load=False):
        epochs = 2000
        learning_rate = .001
        weight_decay = .0005
        batch_size = 100
        early_stop = False
        num_train = 55000

        f = [5, 5]
        k = [20, 50, 800, 500]

        X = tf.placeholder(tf.float32, [None, 28, 28, 1])
        y = tf.placeholder(tf.float32, [None, 10])

        lambdas = [10, .5, .1, .1, .1]
        # lambdas = [1 / num_train for l in lambdas]
        # lambdas = [1., 1., 1., 1., 1.]

        # conv1
        conv1 = blocks.L0Conv2d(
                'conv1',
                [f[0], f[0], 1, k[0]],
                weight_decay=weight_decay,
                lambd=lambdas[0]
        )

        # conv2
        conv2 = blocks.L0Conv2d(
                'conv2',
                [f[1], f[1], k[0], k[1]],
                weight_decay=weight_decay,
                lambd=lambdas[1]
        )

        # fc1, after 2 maxpools
```

```
        fc1 = blocks.L0Dense(
                'fc1',
                [7*7*k[1], k[2]],
                weight_decay=weight_decay,
                lambd=lambdas[2]
        )

        # fc2
        fc2 = blocks.L0Dense(
                'fc2',
                [k[2], k[3]],
                weight_decay=weight_decay,
                lambd=lambdas[3]
        )

        # output layer
        w_out = blocks.weight('w_out', [k[3], 10])
        b_out = blocks.bias('b_out', [10])

        layers = (conv1, conv2, fc1, fc2)

        global_step = tf.train.get_or_create_global_step()

        # Convolutional layers have feature map sparsity
        # FC layers have neuron sparsity

        # during training, the authors disable the bias as that kills any sparsitydd
        if train:
            # The goal here for convolutional layers is output feature map sparsity
            w1 = conv1.sample_weights()
            X_ = blocks.conv(X, w1, 1, None)
            X_ = blocks.relu(X_)
            X_ = blocks.pool(X_, 2, 2)

            w2 = conv2.sample_weights()
            X_ = blocks.conv(X_, w2, 1, None)
            X_ = blocks.relu(X_)
            X_ = blocks.pool(X_, 2, 2)

            # for fully connected layers we instead prune inputs in order to reduce
            # MAC operations at train time, thus the paper measures input neurons
            w3 = fc1.sample_weights()
            X_ = blocks.dense(X_, w3, None)

            w4 = fc2.sample_weights()
            X_ = blocks.dense(X_, w4, None)

            # count the number of neurons in the pruned architecture
            neurons = []
            neurons.append(tf.count_nonzero(tf.reduce_sum(w1, axis=[0, 1, 2]),
dtype=tf.float32))
            neurons.append(tf.count_nonzero(tf.reduce_sum(w2, axis=[0, 1, 2]),
dtype=tf.float32))
            neurons.append(tf.count_nonzero(tf.reduce_sum(w3, axis=[1]), dtype=tf.float32))
            neurons.append(tf.count_nonzero(tf.reduce_sum(w4, axis=[1]), dtype=tf.float32))

        else:
            # at test time use deterministic weights
            X_ = blocks.conv(X, conv1.weights, 1, conv1.bias)
            z1 = conv1.sample_z(tf.shape(X_)[0])
            X_ = X_ * z1
            X_ = blocks.relu(X_)
            X_ = blocks.pool(X_, 2, 2)

            X_ = blocks.conv(X_, conv2.weights, 1, conv2.bias)
            z2 = conv2.sample_z(tf.shape(X_)[0])
            X_ = X_ * z2
            X_ = blocks.relu(X_)
```

```
            X_ = blocks.pool(X_, 2, 2)

            z3 = fc1.sample_z(10000)
            X_ = tf.layers.flatten(X_) * z3
            X_ = blocks.dense(X_, fc1.weights, fc1.bias)

            z4 = fc2.sample_z(10000)
            X_ = X_ * z4
            X_ = blocks.dense(X_, fc2.weights, fc2.bias)

            # count the number of neurons in the pruned architecture
            neurons = []
            neurons.append(tf.count_nonzero(tf.reduce_sum(z1, axis=[0, 1, 2]),
dtype=tf.float32))
            neurons.append(tf.count_nonzero(tf.reduce_sum(z2, axis=[0, 1, 2]),
dtype=tf.float32))
            neurons.append(tf.count_nonzero(tf.reduce_sum(z3, axis=[0]), dtype=tf.float32))
            neurons.append(tf.count_nonzero(tf.reduce_sum(z4, axis=[0]), dtype=tf.float32))

        logits = blocks.dense(X_, w_out, b_out, activation=False)

        pred = tf.nn.softmax(logits)
        loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
        expected_l0 = [l.count_l0() for l in layers]
        reg = tf.reduce_sum([- (1/num_train) * l.regularization() for l in layers])
        loss = loss + reg


        correct_pred = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

        optim = tf.train.AdamOptimizer(learning_rate).minimize(loss, global_step=global_step)

        constrain = [l.constrain_parameters() for l in layers]

        saver = tf.train.Saver()
        checkpoint = 'checkpoints/model.ckpt'
        with tf.Session(config=config) as sess:
            sess.run(tf.global_variables_initializer())
            if load:
                try:
                    saver.restore(sess, 'checkpoints/model.ckpt.{}'.format(load))
                except:
                    pass

            if not train:
                a_test, l_test, n_test, g_test = sess.run([accuracy, loss, neurons,
global_step],
                feed_dict={X:data.test.images, y:data.test.labels})
                print('Deterministic pruned architecture after {} global steps:
{}'.format(g_test, '-'.join([str(int(n)) for n in n_test])))
                print('Test accuracy: {}'.format(a_test))
                print('Test loss: {}'.format(l_test))
                return

            best = 0
            current_epoch = 0
            step_in_epoch = 0
            a_total, l_total = 0, 0
            a_val, l_val = 0, 0
            with tqdm(total=epochs * num_train // batch_size) as t:
                t.update(0)
                while True:
                    # print(len([n.name for n in tf.get_default_graph().as_graph_def().node]))
                    data_train, labels_train = data.train.next_batch(batch_size)
                    a, l, o, s, n, expect, _ = sess.run([accuracy, loss, optim, global_step,
neurons, expected_l0, constrain],
                            feed_dict={X: data_train, y: labels_train})
```

```
                    epochs_completed = data.train.epochs_completed
                    total_epochs = s * batch_size // num_train
                    # grab the next batch of data
                    t.update(s - t.n)
                    a_total += a
                    l_total += l
                    step_in_epoch += 1

                    t.set_postfix(
                        epoch=total_epochs,
                        neurons=n,
                        t_acc=a_total / step_in_epoch,
                        t_loss=l_total / step_in_epoch,
                        v_acc=a_val
                    )

                    # check validation loss every complete epoch
                    if epochs_completed > current_epoch:
                        a_val, l_val = sess.run([accuracy, loss],
                        feed_dict={X: data.validation.images, y: data.validation.labels})
                        if save:
                            if a >= best:
                                saver.save(sess, 'checkpoints/model.ckpt.best')
                                best = a_val
                            if epochs_completed % 10 == 0:
                                saver.save(sess, 'checkpoints/model.ckpt.
{}'.format(epochs_completed))
                            saver.save(sess, checkpoint)
                        #  saver.save(sess, 'model.{}.ckpt'.format(current_epoch))
                        t.set_postfix(
                            epoch=total_epochs,
                            neurons=n,
                            t_acc=a_total / step_in_epoch,
                            t_loss=l_total / step_in_epoch,
                            v_acc=a_val
                        )

                        a_total = 0
                        l_total = 0
                        step_in_epoch = 0
                        current_epoch = epochs_completed

                        if total_epochs >= epochs:
                            break


if __name__ == '__main__':
    # some cuda issues
    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True

    parser = argparse.ArgumentParser()
    parser.add_argument('--train', action='store_true', help='')
    parser.add_argument('--save', action='store_true', help='')
    parser.add_argument('--load', default=None, help='')

    args = parser.parse_args()
    data = mnist.read_data_sets("data", one_hot=True, reshape=False)
    model = Model(data, train=args.train, save=args.save, load=args.load)
```

```
import numpy as np
import tensorflow as tf

LIMIT_A, LIMIT_B, EPSILON = -.1, 1.1, 1e-6

def weight(name, shape):
    return tf.get_variable(
            name=name,
            shape=shape,
            initializer=tf.contrib.layers.xavier_initializer()
    )

def bias(name, shape):
    return tf.get_variable(
            name=name,
            shape=shape,
            initializer=tf.constant_initializer(0.0)
    )

def conv(inputs, filter, stride, bias):
    if not bias:
        bias = 0
    return tf.nn.conv2d(
        input=inputs,
        filter=filter,
        strides=[1, stride, stride, 1],
        padding='SAME',
    ) + bias

def relu(inputs):
    return tf.nn.relu(inputs)

def pool(inputs, kernel_size, stride):
    return tf.nn.max_pool(
        value=inputs,
        ksize=[1, kernel_size, kernel_size, 1],
        strides=[1, stride, stride, 1],
        padding='SAME'
    )

def dense(inputs, filter, bias, activation=True):
    if not bias:
        bias = 0
    inputs = tf.layers.flatten(inputs)
    if activation:
        return tf.nn.relu(tf.matmul(inputs, filter) + bias)
    else:
        return tf.matmul(inputs, filter) + bias

class L0Conv2d():
    def __init__(self, scope, shape, droprate_init=.5, temperature=2./3., weight_decay=1.0,
lambd=1.0, train=True):
        self.shape = shape
        self.temperature = temperature
        self.weight_decay = weight_decay
        self.lambd = lambd
        self.droprate_init = droprate_init
        self.dim_z = shape[3]

        with tf.variable_scope(scope):
            self.weights = tf.get_variable(
                    name='w',
                    shape=shape,
                    initializer=tf.initializers.he_normal()
            )
            self.bias = tf.get_variable(
                    name='b',
                    shape=self.dim_z,
```

```
                    initializer=tf.initializers.constant(0.0)
            )
            self.log_a = tf.get_variable(
                    name='log_a',
                    shape=self.dim_z,
                    initializer=tf.initializers.random_normal(np.log(1 - droprate_init) -
np.log(droprate_init), 1e-2)
            )

    def constrain_parameters(self):
        return tf.clip_by_value(self.log_a, np.log(1e-2), np.log(1e2))

    def cdf_qz(self, x):
        xn = (x - LIMIT_A) / (LIMIT_B - LIMIT_A)
        logits = np.log(xn) - np.log(1 - xn)
        return tf.clip_by_value(tf.sigmoid(logits * self.temperature - self.log_a), EPSILON, 1
- EPSILON)

    def quantile_concrete(self, x):
        y = tf.sigmoid((tf.log(x) - tf.log(1 - x) + self.log_a) / self.temperature)
        return y * (LIMIT_B - LIMIT_A) + LIMIT_A

    def regularization(self):
        q0 = self.cdf_qz(0)
        logpw_col = tf.reduce_sum(- (.5 * self.weight_decay * tf.pow(self.weights, 2)) -
self.lambd, [2, 1, 0])
        logpw = tf.reduce_sum((1 - q0) * logpw_col)
        logpb = -tf.reduce_sum((1 - q0) * (.5 * self.weight_decay * tf.pow(self.bias, 2) -
self.lambd))
        return logpw + logpb

    def count_l0(self):
        ppos = tf.reduce_sum(1 - self.cdf_qz(0))
        n = self.shape[0] * self.shape[1] * self.shape[2]
        return ppos

    def get_eps(self, shape):
        return tf.random_uniform(shape, EPSILON, 1-EPSILON)
        # return tf.Variable(tf.random_uniform(shape, EPSILON, 1-EPSILON))

    def hard_tanh(self, x, min_val, max_val):
        return tf.minimum(max_val, tf.maximum(min_val, x))

    def sample_z(self, batch_size, train=False):
        if train:
            eps = self.get_eps([batch_size, self.dim_z])
            z = tf.reshape(self.quantile_concrete(eps), [batch_size, 1, 1, self.dim_z])
            return self.hard_tanh(z, min_val=0.0, max_val=1.0)

        else:  # mode
            pi = tf.reshape(tf.sigmoid(self.log_a), [1, 1, 1, self.dim_z])
            return self.hard_tanh(pi * (LIMIT_B - LIMIT_A) + LIMIT_A, min_val=0.0, max_val=1.0)

    def sample_weights(self):
        z = tf.reshape(self.quantile_concrete(self.get_eps([self.dim_z])), [1, 1, 1,
self.dim_z])
        return self.hard_tanh(z, min_val=0.0, max_val=1.0) * self.weights

class L0Dense():
    def __init__(self, scope, shape, droprate_init=.5, temperature=2./3., weight_decay=1.0,
lambd=1.0, train=True):
        self.shape = shape
        self.temperature = temperature
        self.weight_decay = weight_decay
        self.lambd = lambd
        self.droprate_init = droprate_init

        with tf.variable_scope(scope):
```

```
            self.weights = tf.get_variable(
                    name='w',
                    shape=shape,
                    initializer=tf.initializers.he_normal()
            )
            self.bias = tf.get_variable(
                    name='b',
                    shape=self.shape[1],
                    initializer=tf.initializers.constant(0.0)
            )
            self.log_a = tf.get_variable(
                    name='log_a',
                    shape=self.shape[0],
                    initializer=tf.initializers.random_normal(np.log(1 - droprate_init) -
np.log(droprate_init), 1e-2)
            )

    def constrain_parameters(self):
        return tf.clip_by_value(self.log_a, np.log(1e-2), np.log(1e2))

    def cdf_qz(self, x):
        xn = (x - LIMIT_A) / (LIMIT_B - LIMIT_A)
        logits = np.log(xn) - np.log(1 - xn)
        return tf.clip_by_value(tf.sigmoid(logits * self.temperature - self.log_a), EPSILON, 1
- EPSILON)

    def quantile_concrete(self, x):
        y = tf.sigmoid((tf.log(x) - tf.log(1 - x) + self.log_a) / self.temperature)
        return y * (LIMIT_B - LIMIT_A) + LIMIT_A

    def regularization(self):
        q0 = self.cdf_qz(0)
        logpw_col = tf.reduce_sum(- (.5 * self.weight_decay * tf.pow(self.weights, 2)) -
self.lambd, 1)
        logpw = tf.reduce_sum((1 - q0) * logpw_col)
        logpb = -tf.reduce_sum(.5 * self.weight_decay * tf.pow(self.bias, 2))
        return logpw + logpb

    def count_l0(self):
        ppos = tf.reduce_sum(1 - self.cdf_qz(0))
        n = self.shape[1]
        return ppos

    def get_eps(self, shape):
        return tf.random_uniform(shape, EPSILON, 1-EPSILON)
        # return tf.Variable(tf.random_uniform(shape, EPSILON, 1-EPSILON))

    def hard_tanh(self, x, min_val, max_val):
        return tf.minimum(max_val, tf.maximum(min_val, x))

    def sample_z(self, batch_size, train=False):
        if train:
            eps = self.get_eps([batch_size, self.shape[0]])
            z = self.quantile_concrete(eps)
            return self.hard_tanh(z, min_val=0.0, max_val=1.0)

        else:  # mode
            pi = tf.tile(tf.reshape(tf.sigmoid(self.log_a), [1, self.shape[0]]),
tf.constant([batch_size, 1]))
            return self.hard_tanh(pi * (LIMIT_B - LIMIT_A) + LIMIT_A, min_val=0.0, max_val=1.0)

    def sample_weights(self):
        z = self.quantile_concrete(self.get_eps([self.shape[0]]))
        return tf.reshape(self.hard_tanh(z, min_val=0.0, max_val=1.0), [self.shape[0], 1]) *
self.weights

    def _sample_weights(self):
        z = self.quantile_concrete(self.get_eps([self.shape[1]]))
```

```
        return tf.reshape(self.hard_tanh(z, min_val=0.0, max_val=1.0), [1, self.shape[1]]) *
self.weights
```