our_mnist.py

```python
# Copyright 2016 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS␣IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================

"""Functions␣for␣downloading␣and␣reading␣MNIST␣data."""

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import gzip

import numpy
from six.moves import xrange  # pylint: disable=redefined-builtin

from tensorflow.contrib.learn.python.learn.datasets import base
from tensorflow.python.framework import dtypes
from tensorflow.python.framework import random_seed

# CVDF mirror of http://yann.lecun.com/exdb/mnist/
m = 1500
SOURCE_URL = 'http://storage.googleapis.com/cvdf-datasets/mnist/'


def _read32(bytestream):
  dt = numpy.dtype(numpy.uint32).newbyteorder('>')
  return numpy.frombuffer(bytestream.read(4), dtype=dt)[0]


def extract_images(f):
  """Extract␣the␣images␣into␣a␣4D␣uint8␣numpy␣array␣[index,␣y,␣x,␣depth].

␣␣Args:
␣␣␣␣f:␣A␣file␣object␣that␣can␣be␣passed␣into␣a␣gzip␣reader.

␣␣Returns:
␣␣␣␣data:␣A␣4D␣uint8␣numpy␣array␣[index,␣y,␣x,␣depth].

␣␣Raises:
␣␣␣␣ValueError:␣If␣the␣bytestream␣does␣not␣start␣with␣2051.

```

```
␣␣"""
  #print('Extracting', f.name)
  with gzip.GzipFile(fileobj=f) as bytestream:
    magic = _read32(bytestream)
    if magic != 2051:
      raise ValueError('Invalid␣magic␣number␣%d␣in␣MNIST␣image␣file:␣%s' %
                       (magic, f.name))
    num_images = _read32(bytestream)
    rows = _read32(bytestream)
    cols = _read32(bytestream)
    buf = bytestream.read(rows * cols * num_images)
    data = numpy.frombuffer(buf, dtype=numpy.uint8)
    data = data.reshape(num_images, rows, cols, 1)
    return data


def dense_to_one_hot(labels_dense, num_classes):
  """Convert␣class␣labels␣from␣scalars␣to␣one-hot␣vectors."""
  num_labels = labels_dense.shape[0]
  index_offset = numpy.arange(num_labels) * num_classes
  labels_one_hot = numpy.zeros((num_labels, num_classes))
  labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
  return labels_one_hot


def extract_labels(f, one_hot=False, num_classes=10):
  """Extract␣the␣labels␣into␣a␣1D␣uint8␣numpy␣array␣[index].

␣␣Args:
␣␣␣␣f:␣A␣file␣object␣that␣can␣be␣passed␣into␣a␣gzip␣reader.
␣␣␣␣one_hot:␣Does␣one␣hot␣encoding␣for␣the␣result.
␣␣␣␣num_classes:␣Number␣of␣classes␣for␣the␣one␣hot␣encoding.

␣␣Returns:
␣␣␣␣labels:␣a␣1D␣uint8␣numpy␣array.

␣␣Raises:
␣␣␣␣ValueError:␣If␣the␣bystream␣doesn't␣start␣with␣2049.
␣␣"""
  #print('Extracting', f.name)
  with gzip.GzipFile(fileobj=f) as bytestream:
    magic = _read32(bytestream)
    if magic != 2049:
      raise ValueError('Invalid␣magic␣number␣%d␣in␣MNIST␣label␣file:␣%s' %
                       (magic, f.name))
    num_items = _read32(bytestream)
    buf = bytestream.read(num_items)
    labels = numpy.frombuffer(buf, dtype=numpy.uint8)
    if one_hot:
      return dense_to_one_hot(labels, num_classes)
    return labels


```

```
class DataSet(object):

  def __init__(self,
               images,
               labels,
               fake_data=False,
               one_hot=False,
               dtype=dtypes.float32,
               reshape=True,
               seed=None):
    """Construct␣a␣DataSet.
␣␣␣␣one_hot␣arg␣is␣used␣only␣if␣fake_data␣is␣true.␣␣`dtype`␣can␣be␣either
␣␣␣␣`uint8`␣to␣leave␣the␣input␣as␣`[0,␣255]`,␣or␣`float32`␣to␣rescale␣into
␣␣␣␣`[0,␣1]`.␣␣Seed␣arg␣provides␣for␣convenient␣deterministic␣testing.
␣␣␣␣"""
    seed1, seed2 = random_seed.get_seed(seed)
    # If op level seed is not set, use whatever graph level seed is returned
    numpy.random.seed(seed1 if seed is None else seed2)
    dtype = dtypes.as_dtype(dtype).base_dtype
    if dtype not in (dtypes.uint8, dtypes.float32):
      raise TypeError('Invalid␣image␣dtype␣%r,␣expected␣uint8␣or␣float32' %
                      dtype)
    if fake_data:
      self._num_examples = 10000
      self.one_hot = one_hot
    else:
      assert images.shape[0] == labels.shape[0], (
          'images.shape:␣%s␣labels.shape:␣%s' % (images.shape, labels.shape))
      self._num_examples = images.shape[0]

      # Convert shape from [num examples, rows, columns, depth]
      # to [num examples, rows*columns] (assuming depth == 1)
      if reshape:
        assert images.shape[3] == 1
        images = images.reshape(images.shape[0],
                                images.shape[1] * images.shape[2])
      if dtype == dtypes.float32:
        # Convert from [0, 255] -> [0.0, 1.0].
        images = images.astype(numpy.float32)
        images = numpy.multiply(images, 1.0 / 255.0)
    self._images = images
    self._labels = labels
    self._epochs_completed = 0
    self._index_in_epoch = 0

  @property
  def images(self):
    return self._images

  @property
  def labels(self):
    return self._labels

  @property
  def num_examples(self):
    return self._num_examples

  @property
  def epochs_completed(self):
    return self._epochs_completed

  def next_batch(self, batch_size, fake_data=False, shuffle=True):
    """Return␣the␣next␣`batch_size`␣examples␣from␣this␣data␣set."""
    if fake_data:
      fake_image = [1] * 784
      if self.one_hot:
        fake_label = [1] + [0] * 9
      else:
        fake_label = 0
      return [fake_image for _ in xrange(batch_size)], [
          fake_label for _ in xrange(batch_size)
      ]
    start = self._index_in_epoch
    # Shuffle for the first epoch
    if self._epochs_completed == 0 and start == 0 and shuffle:
      perm0 = numpy.arange(self._num_examples)
      numpy.random.shuffle(perm0)
      self._images = self.images[perm0]
      self._labels = self.labels[perm0]
    # Go to the next epoch
    if start + batch_size > self._num_examples:
      # Finished epoch
      self._epochs_completed += 1
      # Get the rest examples in this epoch
      rest_num_examples = self._num_examples - start
      images_rest_part = self._images[start:self._num_examples]
      labels_rest_part = self._labels[start:self._num_examples]
      # Shuffle the data
      if shuffle:
        perm = numpy.arange(self._num_examples)
        numpy.random.shuffle(perm)
        self._images = self.images[perm]
        self._labels = self.labels[perm]
      # Start next epoch
      start = 0
      self._index_in_epoch = batch_size - rest_num_examples
      end = self._index_in_epoch
      images_new_part = self._images[start:end]
      labels_new_part = self._labels[start:end]
      return numpy.concatenate((images_rest_part, images_new_part), axis=0) , numpy.co
    else:
      self._index_in_epoch += batch_size
      end = self._index_in_epoch
      return self._images[start:end], self._labels[start:end]


```

```
def read_data_sets(train_dir,
                   fake_data=False,
                   one_hot=False,
                   dtype=dtypes.float32,
                   reshape=True,
                   validation_size=int(m/10),
                   seed=None):
  if fake_data:

    def fake():
      return DataSet(
          [], [], fake_data=True, one_hot=one_hot, dtype=dtype, seed=seed)

    train = fake()
    validation = fake()
    test = fake()
    return base.Datasets(train=train, validation=validation, test=test)

  TRAIN_IMAGES = 'train-images-idx3-ubyte.gz'
  TRAIN_LABELS = 'train-labels-idx1-ubyte.gz'
  TEST_IMAGES = 't10k-images-idx3-ubyte.gz'
  TEST_LABELS = 't10k-labels-idx1-ubyte.gz'

  local_file = base.maybe_download(TRAIN_IMAGES, train_dir,
                                   SOURCE_URL + TRAIN_IMAGES)
  with open(local_file, 'rb') as f:
    train_images = extract_images(f)
    train_images = train_images[range(m),:,:,:]
    #print(train_images.shape)

  local_file = base.maybe_download(TRAIN_LABELS, train_dir,
                                   SOURCE_URL + TRAIN_LABELS)
  with open(local_file, 'rb') as f:
    train_labels = extract_labels(f, one_hot=one_hot)
    train_labels = train_labels[range(m)]
    #print(train_labels.shape)

  local_file = base.maybe_download(TEST_IMAGES, train_dir,
                                   SOURCE_URL + TEST_IMAGES)
  with open(local_file, 'rb') as f:
    test_images = extract_images(f)

  local_file = base.maybe_download(TEST_LABELS, train_dir,
                                   SOURCE_URL + TEST_LABELS)
  with open(local_file, 'rb') as f:
    test_labels = extract_labels(f, one_hot=one_hot)

  if not 0 <= validation_size <= len(train_images):
    raise ValueError(
        'Validation␣size␣should␣be␣between␣0␣and␣{}.␣Received:␣{}.'
        .format(len(train_images), validation_size))

  validation_images = train_images[:validation_size]
  validation_labels = train_labels[:validation_size]
  train_images = train_images[validation_size:]
  train_labels = train_labels[validation_size:]


  options = dict(dtype=dtype, reshape=reshape, seed=seed)

  train = DataSet(train_images, train_labels, **options)
  validation = DataSet(validation_images, validation_labels, **options)
  test = DataSet(test_images, test_labels, **options)

  return base.Datasets(train=train, validation=validation, test=test)


def load_mnist(train_dir=’MNIST-data’):
  return read_data_sets(train_dir)

if __name__ == "__main__":
    #print("test")
    load_mnist()
```