mnist_data

February 28, 2026

1 load_mnist.py

[1]:
```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy
import gzip
```

[2]:
```
def _read32(bytestream):
  dt = numpy.dtype(numpy.uint32).newbyteorder(">")
  return numpy.frombuffer(bytestream.read(4), dtype=dt)[0]
```

[3]:
```
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
```

[4]:
```
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
```

[5]:
```
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
```

[6]:
```
def read_data_sets(
  train_dir="mnist_data",
  one_hot=False,
  reshape=True,
  validation_size=5000,
  seed=None
):
  """Reads and parses examples from MNIST data files."""

  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 = train_dir + "/" + TRAIN_IMAGES

  with open(local_file, "rb") as f:
    train_images = extract_images(f)

  local_file = train_dir + "/" + TRAIN_LABELS
  with open(local_file, "rb") as f:
    train_labels = extract_labels(f, one_hot=one_hot)

  local_file = train_dir + "/" + TEST_IMAGES
  with open(local_file, "rb") as f:
    test_images = extract_images(f)

  local_file = train_dir + "/" + 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:]

  return train_images, train_labels, validation_images, validation_labels,␣
 ↪test_images, test_labels
```

2 vis_utils.py

[7]:
```
import numpy as np
from math import sqrt, ceil
```

[8]:
```
def visualize_grid(Xs, ubound=255.0, padding=1):
  """
  Reshape a 4D tensor of image data to a grid for easy visualization.

  Inputs:
  - Xs: Data of shape (N, H, W, C)
  - ubound: Output grid will have values scaled to the range [0, ubound]
  - padding: The number of blank pixels between elements of the grid
  """

  (N, H, W, C) = Xs.shape

  grid_size = int(ceil(sqrt(N)))
  grid_height = H * grid_size + padding * (grid_size - 1)
  grid_width = W * grid_size + padding * (grid_size - 1)
  grid = np.zeros((grid_height, grid_width, C))

  next_idx = 0
  y0, y1 = 0, H
  for y in range(grid_size):
    x0, x1 = 0, W
    for x in range(grid_size):
      if next_idx < N:
        img = Xs[next_idx]
        low, high = np.min(img), np.max(img)

        grid[y0:y1, x0:x1] = ubound * (img - low) / (high - low)

        next_idx += 1

      x0 += W + padding
      x1 += W + padding

    y0 += H + padding
    y1 += H + padding

  return grid
```

[ ]:
```
def vis_grid(Xs):
  """ visualize a grid of images """

  (N, H, W, C) = Xs.shape
```

```
  A = int(ceil(sqrt(N)))
  G = np.ones(((A * H) + A, (A * W) + A, C), Xs.dtype)
  G *= np.min(Xs)
  n = 0

  for y in range(A):
    for x in range(A):
      if n < N:
        G[
          (y * H) + y: ((y + 1) * H) + y,
          (x * W) + x: ((x + 1) * W) + x,
          :
        ] = Xs[n, :, :, :]
        n += 1

  maxg = G.max()
  ming = G.min()

  G = (G - ming) / (maxg - ming)

  return G
```

[ ]:
```
def vis_nn(rows):
  """ visualize array of arrays of images """

  N = len(rows)
  D = len(rows[0])
  H, W, C = rows[0][0].shape
  Xs = rows[0][0]

  G = np.ones(((N * H) + N, (D * W) + D, C), Xs.dtype)
  for y in range(N):
    for x in range(D):
      G[
        (y * H) + y: ((y + 1) * H) + y,
        (x * W) + x: ((x + 1) * W) + x,
        :
      ] = rows[y][x]

  maxg = G.max()
  ming = G.min()

  G = (G - ming) / (maxg - ming)

  return G
```