```python
"""Datasets used in examples."""

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

import array
import gzip
import os
from os import path
import struct
import urllib2

import numpy as np


_DATA = "/tmp/jax_example_data/"


def _download(url, filename):
  """Download a url to a file in the JAX data temp directory."""
  if not path.exists(_DATA):
    os.makedirs(_DATA)
  out_file = path.join(_DATA, filename)
  if not path.isfile(out_file):
    with open(out_file, "wb") as f:
      f.write(urllib2.urlopen(url, out_file).read())
      print("downloaded {} to {}".format(url, _DATA))


def _partial_flatten(x):
  """Flatten all but the first dimension of an ndarray."""
  return np.reshape(x, (x.shape[0], -1))


def _one_hot(x, k, dtype=np.float32):
  """Create a one-hot encoding of x of size k."""
  return np.array(x[:, None] == np.arange(k), dtype)


def mnist_raw():
```

```python
    """Download and parse the raw MNIST dataset."""
    base_url = "http://yann.lecun.com/exdb/mnist/"

    def parse_labels(filename):
        with gzip.open(filename, "rb") as fh:
            _ = struct.unpack(">II", fh.read(8))
            return np.array(array.array("B", fh.read()), dtype=np.uint8)

    def parse_images(filename):
        with gzip.open(filename, "rb") as fh:
            _, num_data, rows, cols = struct.unpack(">IIII", fh.read(16))
            return np.array(array.array("B", fh.read()),
                                        dtype=np.uint8).reshape(num_data, rows, cols)

    for filename in ["train-images-idx3-ubyte.gz", "train-labels-idx1-ubyte.gz",
                            "t10k-images-idx3-ubyte.gz", "t10k-labels-idx1-ubyte.gz"]:
        _download(base_url + filename, filename)

    train_images = parse_images(path.join(_DATA, "train-images-idx3-ubyte.gz"))
    train_labels = parse_labels(path.join(_DATA, "train-labels-idx1-ubyte.gz"))
    test_images = parse_images(path.join(_DATA, "t10k-images-idx3-ubyte.gz"))
    test_labels = parse_labels(path.join(_DATA, "t10k-labels-idx1-ubyte.gz"))

    return train_images, train_labels, test_images, test_labels


def mnist(permute_train=False):
    """Download, parse and process MNIST data to unit scale and one-hot labels."""
    train_images, train_labels, test_images, test_labels = mnist_raw()

    train_images = _partial_flatten(train_images) / np.float32(255.)
    test_images = _partial_flatten(test_images) / np.float32(255.)
    train_labels = _one_hot(train_labels, 10)
    test_labels = _one_hot(test_labels, 10)

    if permute_train:
        perm = np.random.RandomState(0).permutation(train_images.shape[0])
        train_images = train_images[perm]
        train_labels = train_labels[perm]

    return train_images, train_labels, test_images, test_labels
```

```
"""A basic MNIST example using JAX together with the mini-libraries stax, for
neural network building, and minmax, for first-order stochastic optimization.
"""

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

import time
import itertools

import numpy.random as npr

import jax.numpy as np
from jax import jit, grad
from jax.experimental import minmax
from jax.experimental import stax
from jax.experimental.stax import Dense, Relu, LogSoftmax
import datasets


def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return -np.mean(preds * targets)

def accuracy(params, batch):
  inputs, targets = batch
  target_class = np.argmax(targets, axis=1)
  predicted_class = np.argmax(predict(params, inputs), axis=1)
  return np.mean(predicted_class == target_class)

init_random_params, predict = stax.serial(
    Dense(1024), Relu,
    Dense(1024), Relu,
    Dense(10), LogSoftmax)

if __name__ == "__main__":
  step_size = 0.001
```

```
num_epochs = 10
batch_size = 32
momentum_mass = 0.9

train_images, train_labels, test_images, test_labels = datasets.mnist()
num_train = train_images.shape[0]
num_complete_batches, leftover = divmod(num_train, batch_size)
num_batches = num_complete_batches + bool(leftover)

def data_stream():
  rng = npr.RandomState(0)
  while True:
    perm = rng.permutation(num_train)
    for i in range(num_batches):
      batch_idx = perm[i * batch_size:(i + 1) * batch_size]
      yield train_images[batch_idx], train_labels[batch_idx]
batches = data_stream()

opt_init, opt_update = minmax.momentum(step_size, mass=momentum_mass)

@jit
def update(i, opt_state, batch):
  params = minmax.get_params(opt_state)
  return opt_update(i, grad(loss)(params, batch), opt_state)

_, init_params = init_random_params((-1, 28 * 28))
opt_state = opt_init(init_params)
itercount = itertools.count()

for epoch in range(num_epochs):
  start_time = time.time()
  for _ in range(num_batches):
    opt_state = update(next(itercount), opt_state, next(batches))
  epoch_time = time.time() - start_time

  params = minmax.get_params(opt_state)
  train_acc = accuracy(params, (train_images, train_labels))
  test_acc = accuracy(params, (test_images, test_labels))
  print("Epoch {} in {:0.2f} sec".format(epoch, epoch_time))
  print("Training set accuracy {}".format(train_acc))
  print("Test set accuracy {}".format(test_acc))
```

```
"""A basic MNIST example using Numpy and JAX.

The primary aim here is simplicity and minimal dependencies.
"""


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

import time

import numpy.random as npr

from jax.api import jit, grad
from jax.scipy.misc import logsumexp
import jax.numpy as np
import datasets



def init_random_params(scale, layer_sizes, rng=npr.RandomState(0)):
  return [(scale * rng.randn(m, n), scale * rng.randn(n))
          for m, n, in zip(layer_sizes[:-1], layer_sizes[1:])]

def predict(params, inputs):
  for w, b in params:
    outputs = np.dot(inputs, w) + b
    inputs = np.tanh(outputs)
  return outputs - logsumexp(outputs, axis=1, keepdims=True)

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return -np.mean(preds * targets)

def accuracy(params, batch):
  inputs, targets = batch
  target_class = np.argmax(targets, axis=1)
  predicted_class = np.argmax(predict(params, inputs), axis=1)
  return np.mean(predicted_class == target_class)
```

```
if __name__ == "__main__":
  layer_sizes = [784, 1024, 1024, 10]    # TODO(mattjj): revise to standard arch
  param_scale = 0.1
  step_size = 0.001
  num_epochs = 10
  batch_size = 32

  train_images, train_labels, test_images, test_labels = datasets.mnist()
  num_train = train_images.shape[0]
  num_complete_batches, leftover = divmod(num_train, batch_size)
  num_batches = num_complete_batches + bool(leftover)

  def data_stream():
    rng = npr.RandomState(0)
    while True:
      perm = rng.permutation(num_train)
      for i in range(num_batches):
        batch_idx = perm[i * batch_size:(i + 1) * batch_size]
        yield train_images[batch_idx], train_labels[batch_idx]
  batches = data_stream()

  @jit
  def update(params, batch):
    grads = grad(loss)(params, batch)
    return [(w - step_size * dw, b - step_size * db)
            for (w, b), (dw, db) in zip(params, grads)]

  params = init_random_params(param_scale, layer_sizes)
  for epoch in range(num_epochs):
    start_time = time.time()
    for _ in range(num_batches):
      params = update(params, next(batches))
    epoch_time = time.time() - start_time

    train_acc = accuracy(params, (train_images, train_labels))
    test_acc = accuracy(params, (test_images, test_labels))
    print("Epoch {} in {:0.2f} sec".format(epoch, epoch_time))
    print("Training set accuracy {}".format(train_acc))
    print("Test set accuracy {}".format(test_acc))
```

```python
"""A basic variational autoencoder (VAE) on binarized MNIST using Numpy and JAX.

This file uses the stax network definition library and the minmax optimization
library.
"""


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

import os
import time

import matplotlib.pyplot as plt

import jax.numpy as np
from jax import jit, grad, lax, random
from jax.experimental import minmax
from jax.experimental import stax
from jax.experimental.stax import Dense, FanOut, Relu, Softplus
import datasets



def gaussian_kl(mu, sigmasq):
  """KL divergence from a diagonal Gaussian to the standard Gaussian."""
  return -0.5 * np.sum(1. + np.log(sigmasq) - mu**2. - sigmasq)

def gaussian_sample(rng, mu, sigmasq):
  """Sample a diagonal Gaussian."""
  return mu + np.sqrt(sigmasq) * random.normal(rng, mu.shape)

def bernoulli_logpdf(logits, x):
  """Bernoulli log pdf of data x given logits."""
  return -np.sum(np.logaddexp(0., np.where(x, -1., 1.) * logits))

def elbo(rng, params, images):
  """Monte Carlo estimate of the negative evidence lower bound."""
  enc_params, dec_params = params
  mu_z, sigmasq_z = encode(enc_params, images)
  logits_x = decode(dec_params, gaussian_sample(rng, mu_z, sigmasq_z))
```

```
    return bernoulli_logpdf(logits_x, images) - gaussian_kl(mu_z, sigmasq_z)

def image_sample(rng, params, nrow, ncol):
    """Sample images from the generative model."""
    _, dec_params = params
    code_rng, img_rng = random.split(rng)
    logits = decode(dec_params, random.normal(code_rng, (nrow * ncol, 10)))
    sampled_images = random.bernoulli(img_rng, np.logaddexp(0., logits))
    return image_grid(nrow, ncol, sampled_images, (28, 28))

def image_grid(nrow, ncol, imagevecs, imshape):
    """Reshape a stack of image vectors into an image grid for plotting."""
    images = iter(imagevecs.reshape((-1,) + imshape))
    return np.vstack([np.hstack([next(images).T for _ in range(ncol)][::-1])
                      for _ in range(nrow)]).T


encoder_init, encode = stax.serial(
    Dense(512), Relu,
    Dense(512), Relu,
    FanOut(2),
    stax.parallel(Dense(10), stax.serial(Dense(10), Softplus)),
)

decoder_init, decode = stax.serial(
    Dense(512), Relu,
    Dense(512), Relu,
    Dense(28 * 28),
)


if __name__ == "__main__":
    step_size = 0.001
    num_epochs = 100
    batch_size = 32
    nrow, ncol = 10, 10  # sampled image grid size
    rng = random.PRNGKey(0)

    test_rng = random.PRNGKey(1)  # fixed prng key for evaluation
    imfile = os.path.join(os.getenv("TMPDIR", "/tmp/"), "mnist_vae_{:03d}.png")
```

```
train_images, _, test_images, _ = datasets.mnist(permute_train=True)
num_complete_batches, leftover = divmod(train_images.shape[0], batch_size)
num_batches = num_complete_batches + bool(leftover)

_, init_encoder_params = encoder_init((batch_size, 28 * 28))
_, init_decoder_params = decoder_init((batch_size, 10))
init_params = init_encoder_params, init_decoder_params

opt_init, opt_update = minmax.momentum(step_size, mass=0.9)

def binarize_batch(rng, i, images):
  i = i % num_batches
  batch = lax.dynamic_slice_in_dim(images, i * batch_size, batch_size)
  return random.bernoulli(rng, batch)

@jit
def run_epoch(rng, opt_state):
  def body_fun(i, (rng, opt_state, images)):
    rng, elbo_rng, data_rng = random.split(rng, 3)
    batch = binarize_batch(data_rng, i, images)
    loss = lambda params: -elbo(elbo_rng, params, batch) / batch_size
    g = grad(loss)(minmax.get_params(opt_state))
    return rng, opt_update(i, g, opt_state), images
  init_val = rng, opt_state, train_images
  _, opt_state, _ =   lax.fori_loop(0, num_batches, body_fun, init_val)
  return opt_state

@jit
def evaluate(opt_state, images):
  params = minmax.get_params(opt_state)
  elbo_rng, data_rng, image_rng = random.split(test_rng, 3)
  binarized_test = random.bernoulli(data_rng, images)
  test_elbo = elbo(elbo_rng, params, binarized_test) / images.shape[0]
  sampled_images = image_sample(image_rng, params, nrow, ncol)
  return test_elbo, sampled_images

opt_state = opt_init(init_params)
for epoch in range(num_epochs):
  tic = time.time()
  rng, epoch_rng = random.split(rng)
  opt_state = run_epoch(epoch_rng, opt_state)
```

```
    test_elbo, sampled_images = evaluate(opt_state, test_images)
    print("{: 3d} {} ({:.3f} sec)".format(epoch, test_elbo, time.time() - tic))
    plt.imsave(imfile.format(epoch), sampled_images, cmap=plt.cm.gray)

"""A mock-up showing a ResNet50 network with training on synthetic data.

This file uses the stax neural network definition library and the minmax
optimization library.
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy.random as npr

import jax.numpy as np
from jax import jit, grad
from jax.experimental import minmax
from jax.experimental import stax
from jax.experimental.stax import (AvgPool, BatchNorm, Conv, Dense, FanInSum,
                                   FanOut, Flatten, GeneralConv, Identity,
                                   MaxPool, Relu, LogSoftmax)


# ResNet blocks compose other layers

def ConvBlock(kernel_size, filters, strides=(2, 2)):
  ks = kernel_size
  filters1, filters2, filters3 = filters
  Main = stax.serial(
      Conv(filters1, (1, 1), strides), BatchNorm(), Relu,
      Conv(filters2, (ks, ks), padding='SAME'), BatchNorm(), Relu,
      Conv(filters3, (1, 1)), BatchNorm())
  Shortcut = stax.serial(Conv(filters3, (1, 1), strides), BatchNorm())
  return stax.serial(FanOut(2), stax.parallel(Main, Shortcut), FanInSum, Relu)


def IdentityBlock(kernel_size, filters):
  ks = kernel_size
  filters1, filters2 = filters
  def make_main(input_shape):
```

```
        # the number of output channels depends on the number of input channels
        return stax.serial(
            Conv(filters1, (1, 1)), BatchNorm(), Relu,
            Conv(filters2, (ks, ks), padding='SAME'), BatchNorm(), Relu,
            Conv(input_shape[3], (1, 1)), BatchNorm())
    Main = stax.shape_dependent(make_main)
    return stax.serial(FanOut(2), stax.parallel(Main, Identity), FanInSum, Relu)


# ResNet architectures compose layers and ResNet blocks

def ResNet50(num_classes):
    return stax.serial(
        GeneralConv(('HWCN', 'OIHW', 'NHWC'), 64, (7, 7), (2, 2), 'SAME'),
        BatchNorm(), Relu, MaxPool((3, 3), strides=(2, 2)),
        ConvBlock(3, [64, 64, 256], strides=(1, 1)),
        IdentityBlock(3, [64, 64]),
        IdentityBlock(3, [64, 64]),
        ConvBlock(3, [128, 128, 512]),
        IdentityBlock(3, [128, 128]),
        IdentityBlock(3, [128, 128]),
        IdentityBlock(3, [128, 128]),
        ConvBlock(3, [256, 256, 1024]),
        IdentityBlock(3, [256, 256]),
        IdentityBlock(3, [256, 256]),
        IdentityBlock(3, [256, 256]),
        IdentityBlock(3, [256, 256]),
        IdentityBlock(3, [256, 256]),
        ConvBlock(3, [512, 512, 2048]),
        IdentityBlock(3, [512, 512]),
        IdentityBlock(3, [512, 512]),
        AvgPool((7, 7)), Flatten, Dense(num_classes), LogSoftmax)


if __name__ == "__main__":
    batch_size = 8
    num_classes = 1001
    input_shape = (224, 224, 3, batch_size)
    step_size = 0.1
    num_steps = 10
```

```
init_fun, predict_fun = ResNet50(num_classes)
_, init_params = init_fun(input_shape)

def loss(params, batch):
    inputs, targets = batch
    logits = predict_fun(params, inputs)
    return np.sum(logits * targets)

def accuracy(params, batch):
    inputs, targets = batch
    target_class = np.argmax(targets, axis=-1)
    predicted_class = np.argmax(predict_fun(params, inputs), axis=-1)
    return np.mean(predicted_class == target_class)

def synth_batches():
    rng = npr.RandomState(0)
    while True:
        images = rng.rand(*input_shape).astype('float32')
        labels = rng.randint(num_classes, size=(batch_size, 1))
        onehot_labels = labels == np.arange(num_classes)
        yield images, onehot_labels

opt_init, opt_update = minmax.momentum(step_size, mass=0.9)
batches = synth_batches()

@jit
def update(i, opt_state, batch):
    params = minmax.get_params(opt_state)
    return opt_update(i, grad(loss)(params, batch), opt_state)

opt_state = opt_init(init_params)
for i in xrange(num_steps):
    opt_state = update(i, opt_state, next(batches))
trained_params = minmax.get_params(opt_state)
```