```python
"""Stax is a small but flexible neural net specification library from scratch.

For an example of its use, see examples/resnet50.py.
"""


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

import functools
import itertools
import operator as op

import numpy as onp
import numpy.random as npr
from six.moves import reduce

from jax import lax
from jax import random
from jax.scipy.misc import logsumexp
import jax.numpy as np



# Following the convention used in Keras and tf.layers, we use CamelCase for the
# names of layer constructors, like Conv and Relu, while using snake_case for
# other functions, like lax.conv and relu.



def relu(x): return np.maximum(x, 0.)
def softplus(x): return np.logaddexp(x, 0.)

def logsoftmax(x, axis=-1):
  """Apply log softmax to an array of logits, log-normalizing along an axis."""
  return x - logsumexp(x, axis, keepdims=True)

def fastvar(x, axis, keepdims):
  """A fast but less numerically-stable variance calculation than np.var."""
  return np.mean(x**2, axis, keepdims) - np.mean(x, axis, keepdims)**2


# Initializers
```

```python
def randn(stddev=1e-2, rng=npr):
    """An initializer function for random normal coefficients."""
    def init(shape):
        return rng.normal(size=shape, scale=stddev).astype('float32')
    return init

def glorot(out_dim=0, in_dim=1, scale=onp.sqrt(2), rng=npr):
    """An initializer function for random Glorot-scaled coefficients."""
    def init(shape):
        fan_in, fan_out = shape[in_dim], shape[out_dim]
        size = onp.prod(onp.delete(shape, [in_dim, out_dim]))
        std = scale / np.sqrt((fan_in + fan_out) / 2. * size)
        return rng.normal(size=shape, scale=std).astype('float32')
    return init

zeros = functools.partial(np.zeros, dtype='float32')
ones = functools.partial(np.ones, dtype='float32')


# Layers

# Each layer constructor function returns an (init_fun, apply_fun) pair, where
#     init_fun: takes an input shape and returns an (output_shape, params) pair,
#     apply_fun: takes params, inputs, and an rng key and applies the layer.


def Dense(out_dim, W_init=glorot(), b_init=randn()):
    """Layer constructor function for a dense (fully-connected) layer."""
    def init_fun(input_shape):
        output_shape = input_shape[:-1] + (out_dim,)
        W, b = W_init((input_shape[-1], out_dim)), b_init((out_dim,))
        return output_shape, (W, b)
    def apply_fun(params, inputs, rng=None):
        W, b = params
        return np.dot(inputs, W) + b
    return init_fun, apply_fun


def GeneralConv(dimension_numbers, out_chan, filter_shape,
                strides=None, padding='VALID', W_init=None, b_init=randn(1e-6)):
```

```
    """Layer construction function for a general convolution layer."""
    lhs_spec, rhs_spec, out_spec = dimension_numbers
    one = (1,) * len(filter_shape)
    strides = strides or one
    W_init = W_init or glorot(rhs_spec.index('O'), rhs_spec.index('I'))
    def init_fun(input_shape):
        filter_shape_iter = iter(filter_shape)
        kernel_shape = [out_chan if c == 'O' else
                        input_shape[lhs_spec.index('C')] if c == 'I' else
                        next(filter_shape_iter) for c in rhs_spec]
        output_shape = lax.conv_general_shape_tuple(
            input_shape, kernel_shape, strides, padding, dimension_numbers)
        bias_shape = [out_chan if c == 'C' else 1 for c in out_spec]
        bias_shape = tuple(itertools.dropwhile(lambda x: x == 1, bias_shape))
        W, b = W_init(kernel_shape), b_init(bias_shape)
        return output_shape, (W, b)
    def apply_fun(params, inputs, rng=None):
        W, b = params
        return lax.conv_general_dilated(inputs, W, strides, padding, one, one,
                                        dimension_numbers) + b
    return init_fun, apply_fun
Conv = functools.partial(GeneralConv, ('NHWC', 'HWIO', 'NHWC'))


def BatchNorm(axis=(0, 1, 2), epsilon=1e-5, center=True, scale=True,
              beta_init=zeros, gamma_init=ones):
    """Layer construction function for a batch normalization layer."""
    _beta_init = lambda shape: beta_init(shape) if center else ()
    _gamma_init = lambda shape: gamma_init(shape) if scale else ()
    axis = (axis,) if np.isscalar(axis) else axis
    def init_fun(input_shape):
        shape = (1 if i in axis else d for i, d in enumerate(input_shape))
        shape = tuple(itertools.dropwhile(lambda x: x == 1, shape))
        beta, gamma = _beta_init(shape), _gamma_init(shape)
        return input_shape, (beta, gamma)
    def apply_fun(params, x, rng=None):
        beta, gamma = params
        mean, var = np.mean(x, axis, keepdims=True), fastvar(x, axis, keepdims=True)
        z = (x - mean) / (var + epsilon)**2
        if center and scale: return gamma * z + beta
        if center: return z + beta
```

```
        if scale: return gamma * z
        return z
    return init_fun, apply_fun


def _elemwise_no_params(fun, **kwargs):
    init_fun = lambda input_shape: (input_shape, ())
    apply_fun = lambda params, inputs, rng=None: fun(inputs, **kwargs)
    return init_fun, apply_fun
Tanh = _elemwise_no_params(np.tanh)
Relu = _elemwise_no_params(relu)
LogSoftmax = _elemwise_no_params(logsoftmax, axis=-1)
Softplus = _elemwise_no_params(softplus)


def _pooling_layer(reducer, init_val, rescaler=None):
    def PoolingLayer(window_shape, strides=None, padding='VALID'):
        """Layer construction function for a pooling layer."""
        strides = strides or (1,) * len(window_shape)
        rescale = rescaler(window_shape, strides, padding) if rescaler else None
        dims = (1,) + window_shape + (1,)    # NHWC
        strides = (1,) + strides + (1,)
        def init_fun(input_shape):
            out_shape = lax.reduce_window_shape_tuple(input_shape, dims, strides, padding)
            return out_shape, ()
        def apply_fun(params, inputs, rng=None):
            out = lax.reduce_window(inputs, init_val, reducer, dims, strides, padding)
            return rescale(out, inputs) if rescale else out
        return init_fun, apply_fun
    return PoolingLayer
MaxPool = _pooling_layer(lax.max, -np.inf)
SumPool = _pooling_layer(lax.add, 0.)


def _normalize_by_window_size(dims, strides, padding):
    def rescale(outputs, inputs):
        one = np.ones(inputs.shape[1:3], dtype=inputs.dtype)
        window_sizes = lax.reduce_window(one, 0., lax.add, dims, strides, padding)
        return outputs / window_sizes
    return rescale
AvgPool = _pooling_layer(lax.add, 0., _normalize_by_window_size)
```

```
def Flatten():
  """Layer construction function for flattening all but the leading dim."""
  def init_fun(input_shape):
    output_shape = input_shape[0], reduce(op.mul, input_shape[1:], 1)
    return output_shape, ()
  def apply_fun(params, inputs, rng=None):
    return np.reshape(inputs, (inputs.shape[0], -1))
  return init_fun, apply_fun
Flatten = Flatten()


def Identity():
  """Layer construction function for an identity layer."""
  init_fun = lambda input_shape: (input_shape, ())
  apply_fun = lambda params, inputs, rng=None: inputs
  return init_fun, apply_fun
Identity = Identity()


def FanOut(num):
  """Layer construction function for a fan-out layer."""
  init_fun = lambda input_shape: ([input_shape] * num, ())
  apply_fun = lambda params, inputs, rng=None: [inputs] * num
  return init_fun, apply_fun


def FanInSum():
  """Layer construction function for a fan-in sum layer."""
  init_fun = lambda input_shape: (input_shape[0], ())
  apply_fun = lambda params, inputs, rng=None: sum(inputs)
  return init_fun, apply_fun
FanInSum = FanInSum()


def Dropout(rate, mode='train'):
  """Layer construction function for a dropout layer with given rate."""
  def init_fun(input_shape):
    return input_shape, ()
  def apply_fun(params, inputs, rng):
```

```
    if mode == 'train':
      keep = random.bernoulli(rng, rate, inputs.shape)
      return np.where(keep, inputs / rate, 0)
    else:
      return inputs
  return init_fun, apply_fun


# Composing layers via combinators


def serial(*layers):
  """Combinator for composing layers in serial.

  Args:
    *layers: a sequence of layers, each an (init_fun, apply_fun) pair.

  Returns:
    A new layer, meaning an (init_fun, apply_fun) pair, representing the serial
    composition of the given sequence of layers.
  """
  nlayers = len(layers)
  init_funs, apply_funs = zip(*layers)
  def init_fun(input_shape):
    params = []
    for init_fun in init_funs:
      input_shape, param = init_fun(input_shape)
      params.append(param)
    return input_shape, params
  def apply_fun(params, inputs, rng=None):
    rngs = random.split(rng, nlayers) if rng is not None else (None,) * nlayers
    for fun, param, rng in zip(apply_funs, params, rngs):
      inputs = fun(param, inputs, rng)
    return inputs
  return init_fun, apply_fun


def parallel(*layers):
  """Combinator for composing layers in parallel.

  The layer resulting from this combinator is often used with the FanOut and
```

```
  FanInSum layers.

  Args:
    *layers: a sequence of layers, each an (init_fun, apply_fun) pair.

  Returns:
    A new layer, meaning an (init_fun, apply_fun) pair, representing the
    parallel composition of the given sequence of layers. In particular, the
    returned layer takes a sequence of inputs and returns a sequence of outputs
    with the same length as the argument `layers`.
  """
  nlayers = len(layers)
  init_funs, apply_funs = zip(*layers)
  def init_fun(input_shape):
    return zip(*[init(shape) for init, shape in zip(init_funs, input_shape)])
  def apply_fun(params, inputs, rng=None):
    rngs = random.split(rng, nlayers) if rng is not None else (None,) * nlayers
    return [f(p, x, r) for f, p, x, r in zip(apply_funs, params, inputs, rngs)]
  return init_fun, apply_fun


def shape_dependent(make_layer):
  """Combinator to delay layer constructor pair until input shapes are known.

  Args:
    make_layer: a one-argument function that takes an input shape as an argument
      (a tuple of positive integers) and returns an (init_fun, apply_fun) pair.

  Returns:
    A new layer, meaning an (init_fun, apply_fun) pair, representing the same
    layer as returned by `make_layer` but with its construction delayed until
    input shapes are known.
  """
  def init_fun(input_shape):
    return make_layer(input_shape)[0](input_shape)
  def apply_fun(params, inputs, rng=None):
    return make_layer(inputs.shape)[1](params, inputs, rng)
  return init_fun, apply_fun
```