Haiku

Haiku Contributors

Jan 18, 2022

GUIDES

Haiku is a library built on top of JAX designed to provide simple, composable abstractions for machine learning research.

```
import haiku as hk
import jax
import jax.numpy as jnp

def forward(x):
  mlp = hk.nets.MLP([300, 100, 10])
  return mlp(x)

forward = hk.transform(forward)

rng = jax.random.PRNGKey(42)
x = jnp.ones([8, 28 * 28])
params = forward.init(rng, x)
logits = forward.apply(params, rng, x)
```

CHAPTER

ONE

INSTALLATION

See https://github.com/google/jax#pip-installation for instructions on installing JAX.

We suggest installing the latest version of Haiku by running:

```
$ pip install git+https://github.com/deepmind/dm-haiku
```

Alternatively, you can install via PyPI:

```
$ pip install -U dm-haiku
```

1.1 Haiku Basics

In this Colab, you will learn the basics of Haiku.

What and Why ?

Haiku is a simple neural network library for JAX that enables users to use familiar object-oriented programming models while allowing full access to JAX's pure function transformations. Haiku is designed to make the common things we do such as managing model parameters and other model state simpler and similar in spirit to the Sonnet library that has been widely used across DeepMind. It preserves Sonnet's module-based programming model for state management while retaining access to JAX's function transformations. Haiku can be expected to compose with other libraries and work well with the rest of JAX.

[1]:
```
import haiku as hk
import jax
import jax.numpy as jnp
import numpy as np
```

1.1.1 A first example with hk.transform

As an initial introduction to Haiku, let us construct a linear module with weights and biases with custom initializations.

Similar to Sonnet modules, Haiku modules are Python objects that hold references to their own parameters, other modules, and methods that apply functions on user inputs. On the other hand, since JAX operates on pure function transformations, Haiku modules cannot be instantiated verbatim. Rather, the modules need to be wrapped into pure function transformations.

Haiku provides a simple function transformation, `hk.transform`, that turns functions that use these object-oriented, functionally "impure" modules into pure functions that can be used with JAX.

[2]:
```
class MyLinear1(hk.Module):

  def __init__(self, output_size, name=None):
    super().__init__(name=name)
    self.output_size = output_size

  def __call__(self, x):
    j, k = x.shape[-1], self.output_size
    w_init = hk.initializers.TruncatedNormal(1. / np.sqrt(j))
    w = hk.get_parameter("w", shape=[j, k], dtype=x.dtype, init=w_init)
    b = hk.get_parameter("b", shape=[k], dtype=x.dtype, init=jnp.ones)
    return jnp.dot(x, w) + b
```

[3]:
```
def _forward_fn_linear1(x):
  module = MyLinear1(output_size=2)
  return module(x)

forward_linear1 = hk.transform(_forward_fn_linear1)
```

We see that the forward wrapper object now contains two methods, `init` and `apply`, that are used to initialize the variables and do forward inference on the module.

[4]:
```
forward_linear1
```

[4]:
```
Transformed(init=<function without_state.<locals>.init_fn at 0x7fa22fe754c0>, apply=
↪<function without_state.<locals>.apply_fn at 0x7fa22fe75550>)
```

Calling the init method will initialize the parameters of the network and return them, as can be seen below. The init method takes a `jax.random.PRNGKey` and a sample input (usually just some dummy values to tell the networks about the expected shapes).

[5]:
```
dummy_x = jnp.array([[1., 2., 3.]])
rng_key = jax.random.PRNGKey(42)

params = forward_linear1.init(rng=rng_key, x=dummy_x)
print(params)
```

```
/tmp/haiku-docs-env/lib/python3.8/site-packages/jax/lib/xla_bridge.py:130:␣
↪UserWarning: No GPU/TPU found, falling back to CPU.
  warnings.warn('No GPU/TPU found, falling back to CPU.')
```

```
FlatMapping({
  'my_linear1': FlatMapping({
                  'w': DeviceArray([[-0.30350363,  0.5123802 ],
                                    [ 0.08009142, -0.3163005 ],
                                    [ 0.6056666 ,  0.5820702 ]], dtype=float32),
                  'b': DeviceArray([1., 1.], dtype=float32),
                }),
})
```

We can now use the params to apply the forward function to some inputs.

[6]:
```
sample_x = jnp.array([[1., 2., 3.]])
sample_x_2 = jnp.array([[4., 5., 6.], [7., 8., 9.]])

output_1 = forward_linear1.apply(params=params, x=sample_x, rng=rng_key)
# Outputs are identical for given inputs since the forward inference is non-
↪stochastic.
```

(continues on next page)

(continued from previous page)

```
output_2 = forward_linear1.apply(params=params, x=sample_x, rng=rng_key)

output_3 = forward_linear1.apply(params=params, x=sample_x_2, rng=rng_key)

print(f'Output 1 : {output_1}')
print(f'Output 2 (same as output 1): {output_2}')
print(f'Output 3 : {output_3}')
```

```
Output 1 : [[2.6736789 2.6259897]]
Output 2 (same as output 1): [[2.6736789 2.6259897]]
Output 3 : [[3.820442 4.960439]
 [4.967205 7.294889]]
```

Inference without random key

The module that we built is inherently non-stochastic. In that case, passing a random key to the apply method seems redundant. Haiku offers another transformation `hk.without_apply_rng` which can be further wrapped around our `hk.transform` method.

[7]:
```
forward_without_rng = hk.without_apply_rng(hk.transform(_forward_fn_linear1))
params = forward_without_rng.init(rng=rng_key, x=sample_x)
output = forward_without_rng.apply(x=sample_x, params=params)
print(f'Output without random key in forward pass \n {output_1}')
```

```
Output without random key in forward pass
 [[2.6736789 2.6259897]]
```

We can also mutate the parameters and then do forward inference to generate a different output for the same inputs. This is what is done to apply gradient descent to our parameters while learning.

[8]:
```
mutated_params = jax.tree_map(lambda x: x+1., params)
print(f'Mutated params \n : {mutated_params}')
mutated_output = forward_without_rng.apply(x=sample_x, params=mutated_params)
print(f'Output with mutated params \n {mutated_output}')
```

```
Mutated params
 : FlatMapping({
  'my_linear1': FlatMapping({
                  'b': DeviceArray([2., 2.], dtype=float32),
                  'w': DeviceArray([[0.69649637, 1.5123801 ],
                                    [1.0800915 , 0.6836995 ],
                                    [1.6056666 , 1.5820701 ]], dtype=float32),
                }),
})
Output with mutated params
 [[9.673679 9.62599 ]]
```

1.1.2 Stateful Inference in Haiku

For some modules you might want to maintain and carry over the internal state across function calls. Here, we demonstrate a simple example, where we declare a state variable `counter` within our Haiku transformation which gets updated on each call to the function. Note that we didn't explicitly instantiate this as a Haiku module (the same could be replicated as a hk module as shown earlier).

[9]:
```
def stateful_f(x):
  counter = hk.get_state("counter", shape=[], dtype=jnp.int32, init=jnp.ones)
```

(continues on next page)

(continued from previous page)

```
  multiplier = hk.get_parameter('multiplier', shape=[1,], dtype=x.dtype, init=jnp.
↪ones)
  hk.set_state("counter", counter + 1)
  output = x + multiplier * counter
  return output

stateful_forward = hk.without_apply_rng(hk.transform_with_state(stateful_f))
sample_x = jnp.array([[5., ]])
params, state = stateful_forward.init(x=sample_x, rng=rng_key)
print(f'Initial params:\n{params}\nInitial state:\n{state}')
print('##########')
for i in range(3):
  output, state = stateful_forward.apply(params, state, x=sample_x)
  print(f'After {i+1} iterations:\nOutput: {output}\nState: {state}')
  print('##########')
```

```
Initial params:
FlatMapping({
  '~': FlatMapping({'multiplier': DeviceArray([1.], dtype=float32)}),
})
Initial state:
FlatMapping({'~': FlatMapping({'counter': DeviceArray(1, dtype=int32)})})
##########
After 1 iterations:
Output: [[6.]]
State: FlatMapping({'~': FlatMapping({'counter': DeviceArray(2, dtype=int32)})})
##########
After 2 iterations:
Output: [[7.]]
State: FlatMapping({'~': FlatMapping({'counter': DeviceArray(3, dtype=int32)})})
##########
After 3 iterations:
Output: [[8.]]
State: FlatMapping({'~': FlatMapping({'counter': DeviceArray(4, dtype=int32)})})
##########
```

1.1.3 Built-in Haiku nets and nested modules

The usual networks we use such as MLP, Convnets etc. are defined already in Haiku and we can compose those modules to construct our custom Haiku Module.

Look at the params dictionary to see how the params are nested in the same way as the modules are nested within our custom Haiku module.

[10]:
```
# See: https://dm-haiku.readthedocs.io/en/latest/api.html#common-modules

class MyModuleCustom(hk.Module):
  def __init__(self, output_size=2, name='custom_linear'):
    super().__init__(name=name)
    self._internal_linear_1 = hk.nets.MLP(output_sizes=[2, 3], name='hk_internal_
↪linear')
    self._internal_linear_2 = MyLinear1(output_size=output_size, name='old_linear')

  def __call__(self, x):
    return self._internal_linear_2(self._internal_linear_1(x))
```

(continues on next page)

(continued from previous page)

```
def _custom_forward_fn(x):
  module = MyModuleCustom()
  return module(x)

custom_forward_without_rng = hk.without_apply_rng(hk.transform(_custom_forward_fn))
params = custom_forward_without_rng.init(rng=rng_key, x=sample_x)
params
```

[10]:
```
FlatMapping({
  'custom_linear/~/hk_internal_linear/~/linear_0': FlatMapping({
                                                     'w': DeviceArray([[ 1.51595   , -
↪0.23353337]], dtype=float32),
                                                     'b': DeviceArray([0., 0.],␣
↪dtype=float32),
                                                   }),
  'custom_linear/~/hk_internal_linear/~/linear_1': FlatMapping({
                                                     'w': DeviceArray([[-0.22075887, -
↪0.27375957,  0.5931483 ],
                                                                       [ 0.7818068 , ␣
↪0.72626334, -0.6860752 ]], dtype=float32),
                                                     'b': DeviceArray([0., 0., 0.],␣
↪dtype=float32),
                                                   }),
  'custom_linear/~/old_linear': FlatMapping({
                                  'w': DeviceArray([[ 0.28584382,  0.31626168],
                                                    [ 0.2335775 , -0.4827032 ],
                                                    [-0.14647584, -0.7185701 ]],␣
↪dtype=float32),
                                  'b': DeviceArray([1., 1.], dtype=float32),
                                }),
})
```

1.1.4 Rng Keys with hk.next_rng_key()

The modules that we saw earlier were all non-stochastic. Below we show how to sample random numbers to do stochastic inference.

Haiku offers a trivial model for working with random numbers. Within a transformed function, `hk.next_rng_key()` returns a unique rng key. These unique keys are deterministically derived from an initial random key passed into the top-level transformed function, and are thus safe to use with JAX program transformations.

Let us define a simple haiku function where we generate two random samples. Note that the next_rng_keys are determined from the initial random key passed to the apply method of the top-level transformed function.

[11]:
```
class HkRandom2(hk.Module):
  def __init__(self, rate=0.5):
    super().__init__()
    self.rate = rate

  def __call__(self, x):
    key1 = hk.next_rng_key()
    return jax.random.bernoulli(key1, 1.0 - self.rate, shape=x.shape)


class HkRandomNest(hk.Module):
  def __init__(self, rate=0.5):
```

(continues on next page)

(continued from previous page)

```
    super().__init__()
    self.rate = rate
    self._another_random_module = HkRandom2()

  def __call__(self, x):
    key2 = hk.next_rng_key()
    p1 = self._another_random_module(x)
    p2 = jax.random.bernoulli(key2, 1.0 - self.rate, shape=x.shape)
    print(f'Bernoullis are  : {p1, p2}')

# Note that the modules that are stochastic cannot be wrapped with hk.without_apply_
↪rng()
forward = hk.transform(lambda x: HkRandomNest()(x))

x = jnp.array(1.)
params = forward.init(rng_key, x=x)
for i in range(5):
  print(f'\n Iteration {i+1}')
  prediction = forward.apply(params, x=x, rng=rng_key)
```

```
Bernoullis are  : (DeviceArray(True, dtype=bool), DeviceArray(False, dtype=bool))

 Iteration 1
Bernoullis are  : (DeviceArray(True, dtype=bool), DeviceArray(False, dtype=bool))

 Iteration 2
Bernoullis are  : (DeviceArray(True, dtype=bool), DeviceArray(False, dtype=bool))

 Iteration 3
Bernoullis are  : (DeviceArray(True, dtype=bool), DeviceArray(False, dtype=bool))

 Iteration 4
Bernoullis are  : (DeviceArray(True, dtype=bool), DeviceArray(False, dtype=bool))

 Iteration 5
Bernoullis are  : (DeviceArray(True, dtype=bool), DeviceArray(False, dtype=bool))
```

[ ]:
```
import haiku as hk
import jax
import jax.numpy as jnp
```

TL;DR: A JAX transform inside of a `hk.transform` is likely to transform a side effecting function, which will result in an `UnexpectedTracerError`. This page describes two ways to get around this.

1.2 Limitations of Nesting JAX Functions and Haiku Modules

Once a Haiku network has been transformed to a pair of pure functions using `hk.transform`, it's possible to freely combine these with any JAX transformations like `jax.jit`, `jax.grad`, `jax.scan` and so on.

If you want to use JAX transformations **inside** of a `hk.transform` however, you need to be more careful. It's possible, but most functions inside of the `hk.transform` boundary are still side effecting, and cannot safely be transformed by JAX. This is a common cause of `UnexpectedTracerErrors` in code using Haiku. These errors are a result of using a JAX transform on a side effecting function (for more information on this JAX error, see https://jax.readthedocs.io/en/latest/errors.html#jax.errors.UnexpectedTracerError).

An example with `jax.eval_shape`:

[ ]:
```
def net(x): # inside of a hk.transform, this is still side-effecting
  w = hk.get_parameter("w", (2, 2), init=jnp.ones)
  return w @ x

def eval_shape_net(x):
  output_shape = jax.eval_shape(net, x) # eval_shape on side-effecting function
  return net(x)                         # UnexpectedTracerError!

init, _ = hk.transform(eval_shape_net)
try:
  init(jax.random.PRNGKey(666), jnp.ones((2, 2)))
except jax.errors.UnexpectedTracerError:
  print("UnexpectedTracerError: applied JAX transform to side effecting function")
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

```
UnexpectedTracerError: applied JAX transform to side effecting function
```

These examples use `jax.eval_shape`, but could have used any higher-order JAX function (eg. `jax.vmap`, `jax.scan`, `jax.while_loop`, ...).

The error points to `hk.get_parameter`. This is the operation which makes `net` a side effecting function. The side effect in this case is the creation of a parameter, which gets stored into the Haiku state. Similarly you would get an error using `hk.next_rng_key`, because it advances the Haiku RNG state and stores a new PRNGKey into the Haiku state. In general, transforming a non-transformed Haiku module will result in an `UnexpectedTracerError`.

You could re-write the code above to create the parameter outside of the `eval_shape` transformation, making `net` a pure function by threading through the parameter explictly as an argument:

[ ]:
```
def net(w, x): # no side effects!
  return w @ x

def eval_shape_net(x):
  w = hk.get_parameter("w", (3, 2), init=jnp.ones)
  output_shape = jax.eval_shape(net, w, x) # net is now side-effect free
  return output_shape, net(w, x)

key = jax.random.PRNGKey(777)
x = jnp.ones((2, 3))
init, apply = hk.transform(eval_shape_net)
params = init(key, x)
apply(params, key, x)
```

```
(ShapeDtypeStruct(shape=(3, 3), dtype=float32),
 DeviceArray([[2., 2., 2.],
              [2., 2., 2.],
              [2., 2., 2.]], dtype=float32))
```

However, that's not always possible. Consider the following code which calls a Haiku module (`hk.nets.MLP`) which we don't own. This module will internally call `get_parameter`.

[ ]:
```
def eval_shape_net(x):
  net = hk.nets.MLP([300, 100])
  output_shape = jax.eval_shape(net, x)
  return output_shape, net(x)
```

(continues on next page)

(continued from previous page)

```
init, _ = hk.transform(eval_shape_net)
try:
  init(jax.random.PRNGKey(666), jnp.ones((2, 2)))
except jax.errors.UnexpectedTracerError:
  print("UnexpectedTracerError: applied JAX transform to side effecting function")
```

```
UnexpectedTracerError: applied JAX transform to side effecting function
```

1.2.1 Using hk.lift

We want a way to get access to our implicit Haiku state, and get a functionally pure version of `hk.nets.MLP`. The way to usually achieve this is by using a `hk.transform`, so all we need is a way to nest an inner `hk.tranform` inside an outer `hk.transform`! We'll create another pair of `init` and `apply` functions through `hk.transform`, and these can then be safely combined with any higher-order JAX function.

However, we need a way to register this nested `hk.tranform` state into the outer scope. We can use `hk.lift` for this. Wrapping our inner `init` function with `hk.lift` will register our inner `params` into the outer parameter scope.

[ ]:
```
def eval_shape_net(x):
  net = hk.nets.MLP([300, 100])     # still side-effecting
  init, apply = hk.transform(net)   # nested transform
  params = hk.lift(init, name="inner")(hk.next_rng_key(), x) # register parameters in␣
↪outer module scope with name "inner"
  output_shape = jax.eval_shape(apply, params, hk.next_rng_key(), x) # apply is a␣
↪functionaly pure function and can be transformed!
  out = net(x)
  return out, output_shape


init, apply = hk.transform(eval_shape_net)
params = init(jax.random.PRNGKey(777), jnp.ones((100, 100)))
apply(params, jax.random.PRNGKey(777), jnp.ones((100, 100)))
jax.tree_map(lambda x: x.shape, params)
```

```
FlatMap({
  'inner/mlp/~/linear_0': FlatMap({'b': (300,), 'w': (100, 300)}),
  'inner/mlp/~/linear_1': FlatMap({'b': (100,), 'w': (300, 100)}),
  'mlp/~/linear_0': FlatMap({'b': (300,), 'w': (100, 300)}),
  'mlp/~/linear_1': FlatMap({'b': (100,), 'w': (300, 100)}),
})
```

1.2.2 Using Haiku versions of JAX transforms

Haiku also provides wrapped versions of some of the JAX functions for convenience. For example: `hk.grad`, `hk.vmap`, ... See https://dm-haiku.readthedocs.io/en/latest/api.html#jax-fundamentals for a full list of available functions.

These wrappers apply the JAX function to a functionally pure version of the Haiku function, by doing the explicit state threading for you. They don't introduce an extra namescoping level like `lift` does.

[ ]:
```
def eval_shape_net(x):
  net = hk.nets.MLP([300, 100])          # still side-effecting
  output_shape = hk.eval_shape(net, x)   # hk.eval_shape threads through the Haiku␣
↪state for you
```

(continues on next page)

(continued from previous page)

```
  out = net(x)
  return out, output_shape


init, apply = hk.transform(eval_shape_net)
params = init(jax.random.PRNGKey(777), jnp.ones((100, 100)))
out = apply(params, jax.random.PRNGKey(777), jnp.ones((100, 100)))
```

1.2.3 Summary

To summarize, some good and bad examples of combining JAX transforms and Haiku modules:

What?	Works?	Example
vmapping outside a hk.transform	✓ yes!	jax.vmap(hk.transform(hk.nets.ResNet50))
vmapping inside a hk.transform	unexpected tracer error	hk.transform(jax.vmap(hk.nets.ResNet50))
vmapping a nested hk.transform (without lift)	inner state is not registered	hk.transform(jax.vmap(hk.transform(hk.nets.ResNet50)))
vmapping a nested hk.transform (with lift)	✓ yes!	hk.transform(jax.vmap(hk.lift(hk.transform(hk.nets.ResNet50))))
using hk.vmap	✓ yes!	hk.transform(hk.vmap(hk.nets.ResNet50))

1.3 Haiku Fundamentals

1.3.1 Haiku Transforms

transform(f, *[, apply_rng])	Transforms a function using Haiku modules into a pair of pure functions.
transform_with_state(f)	Transforms a function using Haiku modules into a pair of pure functions.
multi_transform(f)	Transforms a collection of functions using Haiku into pure functions.
multi_transform_with_state(f)	Transforms a collection of functions using Haiku into pure functions.
without_apply_rng(f)	Removes the rng argument from the apply function.
without_state(f)	Wraps a transformed tuple and ignores state in/out.

transform

haiku.**transform**(*f*, *, *apply_rng=True*)

Transforms a function using Haiku modules into a pair of pure functions.

For a function `out = f(*a, **k)` this function returns a pair of two pure functions that call `f(*a, **k)` explicitly collecting and injecting parameter values:

```
params = init(rng, *a, **k)
out = apply(params, rng, *a, **k)
```

Note that the `rng` argument is typically not required for `apply` and passing `None` is accepted.

The first thing to do is to define a *`Module`*. A module encapsulates some parameters and a computation on those parameters:

```
>>> class MyModule(hk.Module):
...   def __call__(self, x):
...     w = hk.get_parameter("w", [], init=jnp.zeros)
...     return x + w
```

Next, define some function that creates and applies modules. We use *`transform()`* to transform that function into a pair of functions that allow us to lift all the parameters out of the function (`f.init`) and apply the function with a given set of parameters (`f.apply`):

```
>>> def f(x):
...   a = MyModule()
...   b = MyModule()
...   return a(x) + b(x)
>>> f = hk.transform(f)
```

To get the initial state of the module call `init` with an example input:

```
>>> params = f.init(None, 1)
>>> params
{'my_module': {'w': DeviceArray(0., dtype=float32)},
 'my_module_1': {'w': DeviceArray(0., dtype=float32)}}
```

You can then apply the function with the given parameters by calling `apply` (note that since we don't use Haiku's random number APIs to apply our network we pass `None` as an RNG key):

```
>>> f.apply(params, None, 1)
DeviceArray(2., dtype=float32)
```

It is expected that your program will at some point produce updated parameters and you will want to re-apply `apply`. You can do this by calling `apply` with different parameters:

```
>>> new_params = {"my_module": {"w": jnp.array(2.)},
...               "my_module_1": {"w": jnp.array(3.)}}
>>> f.apply(new_params, None, 2)
DeviceArray(9., dtype=float32, weak_type=True)
```

If your transformed function needs to maintain internal state (e.g. moving averages in batch norm) then see *`transform_with_state()`*.

Parameters

- **`f`** – A function closing over *`Module`* instances.
- **`apply_rng`** – In the process of being removed. Can only value *True*.

Return type *Transformed*

Returns A *`Transformed`* tuple with `init` and `apply` pure functions.

transform_with_state

haiku.**transform_with_state**(*f*)

Transforms a function using Haiku modules into a pair of pure functions.

See *transform()* for general details on Haiku transformations.

For a function `out = f(*a, **k)` this function returns a pair of two pure functions that call `f(*a, **k)` explicitly collecting and injecting parameter values and state:

```
params, state = init(rng, *a, **k)
out, state = apply(params, state, rng, *a, **k)
```

Note that the `rng` argument is typically not required for `apply` and passing `None` is accepted.

This function is equivalent to *transform()*, however it allows you to maintain and update internal state (e.g. *ExponentialMovingAverage* in *BatchNorm*) via *get_state()* and *set_state()*:

```
>>> def f():
...   counter = hk.get_state("counter", shape=[], dtype=jnp.int32,
...                          init=jnp.zeros)
...   hk.set_state("counter", counter + 1)
...   return counter
>>> f = hk.transform_with_state(f)
```

```
>>> params, state = f.init(None)
>>> for _ in range(10):
...   counter, state = f.apply(params, state, None)
>>> counter
DeviceArray(9, dtype=int32)
```

Parameters **f** – A function closing over *Module* instances.

Return type *TransformedWithState*

Returns A *TransformedWithState* tuple with `init` and `apply` pure functions.

multi_transform

haiku.**multi_transform**(*f*)

Transforms a collection of functions using Haiku into pure functions.

In many scenarios we have several modules which are used either as primitives for several Haiku modules/functions, or whose pure versions are to be reused in downstream code. This utility enables this by applying *transform()* to an arbitrary tree of Haiku functions which share modules and have a common `init` function.

`f` is expected to return a tuple of two elements. First is a `template` Haiku function which provides an example of how all internal Haiku modules are connected. This function is used to create a common `init` function (with your parameters).

The second object is an arbitrary tree of Haiku functions all of which reuse the modules connected in the `template` function. These functions are transformed to pure `apply` functions.

Example:

```
>>> def f():
...   encoder = hk.Linear(1, name="encoder")
...   decoder = hk.Linear(1, name="decoder")
...
...   def init(x):
...     z = encoder(x)
...     return decoder(z)
...
...   return init, (encoder, decoder)
```

```
>>> f = hk.multi_transform(f)
>>> rng = jax.random.PRNGKey(42)
>>> x = jnp.ones([1, 1])
>>> params = f.init(rng, x)
>>> jax.tree_map(jnp.shape, params)
{'decoder': {'b': (1,), 'w': (1, 1)},
 'encoder': {'b': (1,), 'w': (1, 1)}}
```

```
>>> encode, decode = f.apply
>>> z = encode(params, None, x)
>>> y = decode(params, None, z)
```

Parameters **f** (*Callable[[], Tuple[TemplateFn, TreeOfApplyFns]]*) – A factory function that returns two functions, firstly a common init function that creates all modules, and secondly a pytree of apply functions which make use of those modules.

Return type *MultiTransformed*

Returns

A *MultiTransformed* instance which contains a pure init function that creates all parameters, and a pytree of pure apply functions that given the params apply the given function.

See also:

multi_transform_with_state(): Equivalent for modules using state.

multi_transform_with_state

haiku.**multi_transform_with_state**(*f*)

Transforms a collection of functions using Haiku into pure functions.

See *multi_transform()* for more details.

Example:

```
>>> def f():
...   encoder = hk.Linear(1, name="encoder")
...   decoder = hk.Linear(1, name="decoder")
...
...   def init(x):
...     z = encoder(x)
...     return decoder(z)
...
...   return init, (encoder, decoder)
```

```
>>> f = hk.multi_transform_with_state(f)
>>> rng = jax.random.PRNGKey(42)
>>> x = jnp.ones([1, 1])
>>> params, state = f.init(rng, x)
>>> jax.tree_map(jnp.shape, params)
{'decoder': {'b': (1,), 'w': (1, 1)},
 'encoder': {'b': (1,), 'w': (1, 1)}}
```

```
>>> encode, decode = f.apply
>>> z, state = encode(params, state, None, x)
>>> y, state = decode(params, state, None, z)
```

Parameters **f** (*Callable[[], Tuple[TemplateFn, TreeOfApplyFns]]*) – Function returning a "template" function and an arbitrary tree of functions using modules connected in the template function.

Return type *MultiTransformedWithState*

Returns An `init` function and a tree of pure `apply` functions.

See also:

transform_with_state(): Transform a single apply function. *multi_transform()*: Transform multiple apply functions without state.

without_apply_rng

haiku.**without_apply_rng**(*f*)

Removes the rng argument from the apply function.

This is a convenience wrapper that makes the `rng` argument to `f.apply` default to `None`. This is useful when `f` doesn't actually use random numbers as part of its computation, such that the `rng` argument wouldn't be used. Note that if `f` *does* use random numbers, this will cause an error to be thrown complaining that `f` needs a non-None PRNGKey.

Parameters **f** (*TransformedT*) – A transformed function.

Return type TransformedT

Returns The same transformed function, with a modified `apply`.

without_state

haiku.**without_state**(*f*)

Wraps a transformed tuple and ignores state in/out.

The example below is equivalent to `f = hk.transform(f)`:

```
>>> def f(x):
...   mod = hk.Linear(10)
...   return mod(x)
>>> f = hk.without_state(hk.transform_with_state(f))
>>> rng = jax.random.PRNGKey(42)
>>> x = jnp.zeros([1, 1])
>>> params = f.init(rng, x)
>>> f.apply(params, rng, x)
DeviceArray([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

Parameters **f** (TransformedWithState) – A transformed function.

Return type *Transformed*

Returns A transformed function that does not take or return state.

with_empty_state

haiku.**with_empty_state**(*f*)

Wraps a transformed tuple and passes empty state in/out.

The example below is equivalent to `f = hk.transform_with_state(f)`:

```
>>> def f(x):
...   mod = hk.Linear(10)
...   return mod(x)
>>> f = hk.with_empty_state(hk.transform(f))
>>> rng = jax.random.PRNGKey(42)
>>> x = jnp.zeros([1, 1])
>>> params, state = f.init(rng, x)
>>> state
{}
>>> out, state = f.apply(params, state, rng, x)
>>> out
DeviceArray([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
>>> state
{}
```

Parameters **f** (Transformed) – A transformed function.

Return type *TransformedWithState*

Returns A transformed function that does accepts and returns state.

1.3.2 Modules, Parameters and State

Module([name])	Base class for Haiku modules.
to_module(f)	Converts a function into a callable module class.
get_parameter(name, shape[, dtype, init])	Creates or reuses a parameter for the given transformed function.
get_state(name[, shape, dtype, init])	Gets the current value for state with an optional initializer.
set_state(name, value)	Sets the current value for some state.
transparent(method)	Decorator to wrap a method, preventing automatic variable scope wrapping.
lift(init_fn[, name])	Lifts the given init fn to a function in the current Haiku namespace.

Module

class haiku.**Module**(*name=None*)

Base class for Haiku modules.

A Haiku module is a lightweight container for variables and other modules. Modules typically define one or more "forward" methods (e.g. `__call__`) which apply operations combining user input and module parameters.

Modules must be initialized inside a *`transform()`* call.

For example:

```
>>> class AddModule(hk.Module):
...   def __call__(self, x):
...     w = hk.get_parameter("w", [], init=jnp.ones)
...     return x + w
```

```
>>> def forward_fn(x):
...   mod = AddModule()
...   return mod(x)
```

```
>>> forward = hk.transform(forward_fn)
>>> x = 1.
>>> rng = None
>>> params = forward.init(rng, x)
>>> forward.apply(params, None, x)
DeviceArray(2., dtype=float32)
```

__init__(*name=None*)

Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*`Optional[str]`*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

__post_init__(*name=None*)

Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*`Optional[str]`*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

params_dict()

Returns parameters keyed by name for this module and submodules.

Return type Mapping[str, jnp.array]

state_dict()

Returns state keyed by name for this module and submodules.

Return type Mapping[str, jnp.array]

to_module

haiku.**to_module**(*f*)

Converts a function into a callable module class.

Sample usage:

```
>>> def bias_fn(x):
...   b = hk.get_parameter("b", [], init=hk.initializers.RandomNormal())
...   return x + b
>>> Bias = hk.to_module(bias_fn)
>>> def net(x, y):
...   b = Bias(name="my_bias")
...   # Bias x and y by the same amount.
...   return b(x) * b(y)
```

Parameters **f** (*Callable[.., Any]*) – The function to convert.

Return type Type[CallableModule]

Returns A module class which runs `f` when called.

get_parameter

haiku.**get_parameter**(*name*, *shape*, *dtype=<class 'jax._src.numpy.lax_numpy.float32'>*, *init=None*)

Creates or reuses a parameter for the given transformed function.

```
>>> hk.get_parameter("w", [], init=jnp.ones)
DeviceArray(1., dtype=float32)
```

Parameters within the same *transform()* and/or *Module* with the same name have the same value:

```
>>> w1 = hk.get_parameter("w", [], init=jnp.zeros)
>>> w2 = hk.get_parameter("w", [], init=jnp.zeros)
>>> assert w1 is w2
```

Parameters

- **name** (*str*) – A name for the parameter.
- **shape** (*Sequence[int]*) – The shape of the parameter.
- **dtype** (*Any*) – The dtype of the parameter.
- **init** (*Optional[Initializer]*) – A callable of shape, dtype to generate an initial value for the parameter.

Return type jnp.ndarray

Returns A jnp.ndarray with the parameter of the given shape.

get_state

haiku.**get_state**(*name*, *shape=None*, *dtype=<class 'jax._src.numpy.lax_numpy.float32'>*, *init=None*)

Gets the current value for state with an optional initializer.

"State" can be used to represent mutable state in your network. The most common usage of state is to represent the moving averages used in batch normalization (see *ExponentialMovingAverage*). If your network uses "state" then you are required to use *transform_with_state()* and pass state into and out of the apply function.

```
>>> hk.get_state("counter", [], init=jnp.zeros)
DeviceArray(0., dtype=float32)
```

If the value for the given state is already defined (e.g. using *set_state()*) then you can call with just the name:

```
>>> hk.get_state("counter")
DeviceArray(0., dtype=float32)
```

MOTE: state within the same *transform()* and/or *Module* with the same name have the same value:

```
>>> c1 = hk.get_state("counter")
>>> c2 = hk.get_state("counter")
>>> assert c1 is c2
```

Parameters

- **name** (*str*) – A name for the state.
- **shape** (*Optional[Sequence[int]]*) – The shape of the state.
- **dtype** (*Any*) – The dtype of the state.
- **init** (*Optional[Initializer]*) – A callable `f(shape, dtype)` that returns an initial value for the state.

Return type jnp.ndarray

Returns A jnp.ndarray with the state of the given shape.

set_state

haiku.**set_state**(*name*, *value*)

Sets the current value for some state.

See *get_state()*.

"State" can be used to represent mutable state in your network. The most common usage of state is to represent the moving averages used in batch normalization (see *ExponentialMovingAverage*). If your network uses "state" then you are required to use *transform_with_state()* and pass state into and out of the apply function.

```
>>> hk.set_state("counter", jnp.zeros([]))
>>> hk.get_state("counter")
DeviceArray(0., dtype=float32)
```

NOTE: state within the same *transform()* and/or *Module* with the same name have the same value:

```
>>> w1 = hk.get_state("counter")
>>> w2 = hk.get_state("counter")
>>> assert w1 is w2
```

Parameters

- **name** (*str*) – A name for the state.
- **value** – A value to set.

transparent

haiku.**transparent**(*method*)

Decorator to wrap a method, preventing automatic variable scope wrapping.

By default, all variables and modules created in a method are scoped by the module and method names. This is undesirable in some cases. Any method decorated with *transparent()* will create variables and modules in the scope in which it was called.

Parameters **method** (*T*) – the method to wrap.

Return type T

Returns The method, with a flag indicating no name scope wrapping should occur.

lift

haiku.**lift**(*init_fn*, *name='lifted'*)

Lifts the given init fn to a function in the current Haiku namespace.

During init, the returned callable will run the given `init_fn`, and include the resulting params in the outer transform's dictionaries. During `apply`, the returned callable will instead pull the relevant parameters from the outer transform's dictionaries.

Must be called inside *transform()*, and be passed the `init` member of a *Transformed*.

The user must ensure that the given `init` does not accidentally catch modules from an outer *transform()* via functional closure.

Example

```
>>> def g(x):
...   return hk.Linear(1)(x)
>>> g = hk.transform(g)
>>> init_rng = hk.next_rng_key() if hk.running_init() else None
>>> x = jnp.ones([1, 1])
>>> params = hk.lift(g.init, name='f_lift')(init_rng, x)
>>> out = g.apply(params, None, x)
```

Parameters

- **init_fn** (*Callable[.., hk.Params]*) – The `init` function from an *Transformed*.
- **name** (*str*) – A string name to prefix parameters with.

Return type Callable[.., hk.Params]

Returns A callable that during `init` injects parameter values into the outer context and during `apply` reuses parameters from the outer context. In both cases returns parameter values to be used with an `apply` function.

1.3.3 Getters and Interceptors

custom_creator(creator, *[, params, state])	Registers a custom parameter and/or state creator.
custom_getter(getter, *[, params, state])	Registers a custom parameter or state getter.
GetterContext(full_name, module, ...)	Read only state showing where parameters are being created.
intercept_methods(interceptor)	Register a new method interceptor.
MethodContext(module, method_name, ...)	Read only state showing the calling context for a method.

custom_creator

haiku.**custom_creator**(*creator*, *, *params=True*, *state=False*)

Registers a custom parameter and/or state creator.

When new parameters are created via *get_parameter()* we first run custom creators passing user defined values through. For example:

```
>>> def zeros_creator(next_creator, shape, dtype, init, context):
...   init = jnp.zeros
...   return next_creator(shape, dtype, init)
```

```
>>> with hk.custom_creator(zeros_creator):
...   z = hk.get_parameter("z", [], jnp.float32, jnp.ones)
>>> z
DeviceArray(0., dtype=float32)
```

If `state=True` then your creator will additionally run on calls to *get_state()*:

```
>>> with hk.custom_creator(zeros_creator, state=True):
...   z = hk.get_state("z", [], jnp.float32, jnp.ones)
>>> z
DeviceArray(0., dtype=float32)
```

Parameters

- **creator** (*Creator*) – A parameter creator.
- **params** (*bool*) – Whether to intercept parameter creation, defaults to `True`.
- **state** (*bool*) – Whether to intercept state creation, defaults to `False`.

Return type contextlib.AbstractContextManager

Returns Context manager under which the creator is active.

custom_getter

haiku.**custom_getter**(*getter*, *, *params=True*, *state=False*)

Registers a custom parameter or state getter.

When parameters are retrieved using *get_parameter()* we always run all custom getters before returning a value to the user.

```
>>> def bf16_getter(next_getter, value, context):
...   value = value.astype(jnp.bfloat16)
...   return next_getter(value)
```

```
>>> with hk.custom_getter(bf16_getter):
...   w = hk.get_parameter("w", [], jnp.float32, jnp.ones)
>>> w.dtype
dtype(bfloat16)
```

If `state=True` the getter will additionally run for calls to *get_state()*:

```
>>> with hk.custom_getter(bf16_getter, state=True):
...   c = hk.get_state("c", [], jnp.float32, jnp.ones)
>>> c.dtype
dtype(bfloat16)
```

Parameters

- **getter** (*Getter*) – A parameter getter.
- **params** (*bool*) – Whether the getter should run on *get_parameter()*
- **state** (*bool*) – Whether the getter should run on *get_state()*.

Return type contextlib.AbstractContextManager

Returns Context manager under which the getter is active.

GetterContext

class haiku.**GetterContext**(*full_name: str*, *module: Optional[*Module*]*, *original_dtype: Any*, *original_shape: Sequence[int]*)

Read only state showing where parameters are being created.

full_name
The full name of the given parameter (e.g. `mlp/~/linear_0/w`).

module
The module that owns the current parameter, `None` if this parameter exists outside any module.

original_dtype
The dtype that *get_parameter()* or *get_state()* was originally called with.

original_shape
The shape that *get_parameter()* or *get_state()* was originally called with.

module_name
The full name of enclosing modules.

name
The name of this parameter.

intercept_methods

haiku.**intercept_methods**(*interceptor*)

Register a new method interceptor.

Method interceptors allow you to (at a distance) intercept method calls to modules and modify args/kwargs before calling the underlying method. After the underlying method is called you can modify its result before it is passed back to the user.

For example you could intercept method calls to *BatchNorm* and ensure it is always computed in full precision:

```
>>> def my_interceptor(next_f, args, kwargs, context):
...   if (type(context.module) is not hk.BatchNorm
...       or context.method_name != "__call__"):
...     # We ignore methods other than BatchNorm.__call__.
...     return next_f(*args, **kwargs)
...
...   def cast_if_array(x):
...     if isinstance(x, jnp.ndarray):
...       x = x.astype(jnp.float32)
...     return x
...
...   args, kwargs = jax.tree_map(cast_if_array, (args, kwargs))
...   out = next_f(*args, **kwargs)
...   return out
```

We can create and use our module in the usual way, we just need to wrap any method calls we want to intercept in the context manager:

```
>>> mod = hk.BatchNorm(decay_rate=0.9, create_scale=True, create_offset=True)
>>> x = jnp.ones([], jnp.bfloat16)
>>> with hk.intercept_methods(my_interceptor):
...   out = mod(x, is_training=True)
>>> assert out.dtype == jnp.float32
```

Without the interceptor BatchNorm would compute in bf16, however since we cast x before the underlying method is called we compute in f32.

Parameters **interceptor** (*MethodGetter*) – A method interceptor.

Returns Context manager under which the interceptor is active.

MethodContext

class haiku.**MethodContext**(*module: Modul*, *method_name: str*, *orig_method: Callable[..., Any]*)

Read only state showing the calling context for a method.

For example, let's define two interceptors and print the values in the context. Additionally, we will make the first interceptor conditionally short circuit, since interceptors stack and are run in order, an earlier interceptor can decide to call the next interceptor, or short circuit and call the underlying method directly:

```
>>> module = hk.Linear(1, name="method_context_example")
>>> short_circuit = False
```

```
>>> def my_interceptor_1(next_fun, args, kwargs, context):
...   print('running my_interceptor_1')
...   print('- module.name: ', context.module.name)
```

(continues on next page)

(continued from previous page)

```
...   print('- method_name: ', context.method_name)
...   if short_circuit:
...     return context.orig_method(*args, **kwargs)
...   else:
...     return next_fun(*args, **kwargs)
>>> def my_interceptor_2(next_fun, args, kwargs, context):
...   print('running my_interceptor_2')
...   print('- module.name: ', context.module.name)
...   print('- method_name: ', context.method_name)
...   return next_fun(*args, **kwargs)
```

When `short_circuit=False` the two interceptors will run in order:

```
>>> with hk.intercept_methods(my_interceptor_1), \
...      hk.intercept_methods(my_interceptor_2):
...   _ = module(jnp.ones([1, 1]))
running my_interceptor_1
- module.name:  method_context_example
- method_name:  __call__
running my_interceptor_2
- module.name:  method_context_example
- method_name:  __call__
```

Setting `short_circuit=True` will cause the first interceptor to call the original method (rather than `next_fun` which will trigger the next interceptor):

```
>>> short_circuit = True
>>> with hk.intercept_methods(my_interceptor_1), \
...      hk.intercept_methods(my_interceptor_2):
...   _ = module(jnp.ones([1, 1]))
running my_interceptor_1
- module.name:  method_context_example
- method_name:  __call__
```

module
A *Module* instance whose method is being called.

method_name
The name of the method being called on the module.

orig_method
The underlying method on the module which when called will *not* trigger interceptors. You should only call this if you want to short circuit all the other interceptors, in general you should prefer to call the `next_fun` passed to your interceptor which will run `orig_method` after running all other interceptors.

1.3.4 Random Numbers

PRNGSequence(key_or_seed)	Iterator of JAX random keys.
next_rng_key()	Returns a unique JAX random key split from the current global key.
next_rng_keys(num)	Returns one or more JAX random keys split from the current global key.
maybe_next_rng_key()	*next_rng_key()* if random numbers are available, else `None`.

continues on next page

Table 4 – continued from previous page

reserve_rng_keys(num)	Pre-allocate some number of JAX RNG keys.
with_rng(key)	Provides a new sequence for *next_rng_key()* to draw from.

PRNGSequence

class haiku.**PRNGSequence**(*key_or_seed*)

Iterator of JAX random keys.

```
>>> seq = hk.PRNGSequence(42)  # OR pass a jax.random.PRNGKey
>>> key1 = next(seq)
>>> key2 = next(seq)
>>> assert key1 is not key2
```

If you know how many keys you will want then you can use *reserve()* to more efficiently split the keys you need:

```
>>> seq.reserve(4)
>>> keys = [next(seq) for _ in range(4)]
```

__init__(*key_or_seed*)

Creates a new *PRNGSequence*.

reserve(*num*)

Splits additional `num` keys for later use.

__next__()

Return the next item from the iterator. When exhausted, raise StopIteration

Return type PRNGKey

next()

Return the next item from the iterator. When exhausted, raise StopIteration

Return type PRNGKey

next_rng_key

haiku.**next_rng_key**()

Returns a unique JAX random key split from the current global key.

```
>>> key = hk.next_rng_key()
>>> _ = jax.random.uniform(key, [])
```

Return type PRNGKey

Returns A unique (within a call to `init` or `apply`) JAX rng key that can be used with APIs such as `jax.random.uniform()`.

next_rng_keys

haiku.**next_rng_keys**(*num*)

Returns one or more JAX random keys split from the current global key.

```
>>> k1, k2 = hk.next_rng_keys(2)
>>> assert (k1 != k2).all()
>>> a = jax.random.uniform(k1, [])
>>> b = jax.random.uniform(k2, [])
>>> assert a != b
```

Parameters **num** (*int*) – The number of keys to split.

Return type jnp.ndarray

Returns An array of shape `[num, 2]` unique (within a transformed function) JAX rng keys that can be used with APIs such as `jax.random.uniform()`.

maybe_next_rng_key

haiku.**maybe_next_rng_key**()

next_rng_key() if random numbers are available, else `None`.

Return type Optional[PRNGKey]

reserve_rng_keys

haiku.**reserve_rng_keys**(*num*)

Pre-allocate some number of JAX RNG keys.

See *next_rng_key()*.

This API offers a way to micro-optimize how RNG keys are split when using Haiku. It is unlikely that you need it unless you find compilation time of your `init` function to be a problem, or you sample a lot of random numbers in `apply`.

```
>>> hk.reserve_rng_keys(2)  # Pre-allocate 2 keys for us to consume.
>>> _ = hk.next_rng_key()   # Takes the first pre-allocated key.
>>> _ = hk.next_rng_key()   # Takes the second pre-allocated key.
>>> _ = hk.next_rng_key()   # Splits a new key.
```

Parameters **num** (*int*) – The number of JAX rng keys to allocate.

with_rng

haiku.**with_rng**(*key*)

Provides a new sequence for *next_rng_key()* to draw from.

When *next_rng_key()* is called, it draws a new key from the *PRNGSequence* defined by the input key to the transformed function. This context manager overrides the sequence for the duration of the scope.

```
>>> with hk.with_rng(jax.random.PRNGKey(428)):
...   s = jax.random.uniform(hk.next_rng_key(), ())
>>> print("{:.1f}".format(s))
0.5
```

Parameters **key** (*PRNGKey*) – The key to seed the sequence with.

Returns Context manager under which the given sequence is active.

1.3.5 Type Hints

LSTMState(hidden, cell)	An LSTM core state consists of hidden and cell vectors.
Params	The central part of internal API.
State	The central part of internal API.
Transformed(init, hk.Params], apply, Any])	Holds a pair of pure functions.
TransformedWithState(init, Tuple[hk.Params, ...)	Holds a pair of pure functions.
MultiTransformed(init, hk.Params], apply)	Holds a collection of pure functions.
MultiTransformedWithState(init, ...)	Holds a collection of pure functions.

LSTMState

class haiku.**LSTMState**(*hidden: jnp.ndarray*, *cell: jnp.ndarray*)

An LSTM core state consists of hidden and cell vectors.

hidden

Hidden state.

cell

Cell state.

Params

haiku.**Params**

State

haiku.**State**

Transformed

class haiku.**Transformed**(*init: Callable[..., hk.Params]*, *apply: Callable[..., Any]*)

Holds a pair of pure functions.

init

A pure function: `params = init(rng, *a, **k)`

apply

A pure function: `out = apply(params, rng, *a, **k)`

TransformedWithState

class haiku.**TransformedWithState**(*init: Callable[..., Tuple[hk.Params, hk.State]], apply: Callable[..., Tuple[Any, hk.State]]*)

Holds a pair of pure functions.

init
A pure function: `params, state = init(rng, *a, **k)`

apply
A pure function: `out, state = apply(params, state, rng, *a, **k)`

MultiTransformed

class haiku.**MultiTransformed**(*init: Callable[..., hk.Params], apply: Any*)

Holds a collection of pure functions.

init
A pure function: `params = init(rng, *a, **k)`

apply
A JAX tree of pure functions each with the signature: `out = apply(params, rng, *a, **k)`.

See also:

Transformed: Single apply variant of multi-transform. *MultiTransformedWithState*: Multi apply with state variant.

MultiTransformedWithState

class haiku.**MultiTransformedWithState**(*init: Callable[..., Tuple[hk.Params, hk.State]], apply: Any*)

Holds a collection of pure functions.

init
A pure function: `params, state = init(rng, *a, **k)`

apply
A JAX tree of pure functions each with the signature: `out, state = apply(params, state, rng, *a, **k)`.

See also:

TransformedWithState: Single apply variant of multi-transform. *MultiTransformed*: Multi apply with state variant.

1.4 Common Modules

1.4.1 Linear

Linear(output_size[, with_bias, w_init, ...])	Linear module.
Bias([output_size, bias_dims, b_init, name])	Adds a bias to inputs.

Linear

class haiku.**Linear**(*output_size*, *with_bias=True*, *w_init=None*, *b_init=None*, *name=None*)
Linear module.

__init__(*output_size*, *with_bias=True*, *w_init=None*, *b_init=None*, *name=None*)
Constructs the Linear module.

Parameters

- **output_size** (*int*) – Output dimensionality.
- **with_bias** (*bool*) – Whether to add a bias to the output.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for weights. By default, uses random values from truncated normal, with stddev `1 / sqrt(fan_in)`. See https://arxiv.org/abs/1502.03167v3.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for bias. By default, zero.
- **name** (*Optional[str]*) – Name of the module.

__call__(*inputs*, ***, *precision=None*)
Computes a linear transform of the input.

Return type jnp.ndarray

Bias

class haiku.**Bias**(*output_size=None*, *bias_dims=None*, *b_init=None*, *name=None*)
Adds a bias to inputs.

Example Usage:

```
>>> N, H, W, C = 1, 2, 3, 4
>>> x = jnp.ones([N, H, W, C])
>>> scalar_bias = hk.Bias(bias_dims=[])
>>> scalar_bias_output = scalar_bias(x)
>>> assert scalar_bias.bias_shape == ()
```

Create a bias over all non-minibatch dimensions:

```
>>> all_bias = hk.Bias()
>>> all_bias_output = all_bias(x)
>>> assert all_bias.bias_shape == (H, W, C)
```

Create a bias over the last non-minibatch dimension:

```
>>> last_bias = hk.Bias(bias_dims=[-1])
>>> last_bias_output = last_bias(x)
>>> assert last_bias.bias_shape == (C,)
```

Create a bias over the first non-minibatch dimension:

```
>>> first_bias = hk.Bias(bias_dims=[1])
>>> first_bias_output = first_bias(x)
>>> assert first_bias.bias_shape == (H, 1, 1)
```

Subtract and later add the same learned bias:

```
>>> bias = hk.Bias()
>>> h1 = bias(x, multiplier=-1)
>>> h2 = bias(x)
>>> h3 = bias(x, multiplier=-1)
>>> reconstructed_x = bias(h3)
>>> assert (x == reconstructed_x).all()
```

__init__(*output_size=None*, *bias_dims=None*, *b_init=None*, *name=None*)

Constructs a `Bias` module that supports broadcasting.

Parameters

- **output_size** (*Optional[Sequence[int]]*) – Output size (output shape without batch dimension). If `output_size` is left as *None*, the size will be directly inferred by the input.
- **bias_dims** (*Optional[Sequence[int]]*) – Sequence of which dimensions to retain from the input shape when constructing the bias. The remaining dimensions will be broadcast over (given size of 1), and leading dimensions will be removed completely. See class doc for examples.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for the bias. Default to zeros.
- **name** (*Optional[str]*) – Name of the module.

__call__(*inputs*, *multiplier=None*)

Adds bias to `inputs` and optionally multiplies by `multiplier`.

Parameters

- **inputs** (*jnp.ndarray*) – A Tensor of size `[batch_size, input_size1, ...]`.
- **multiplier** (*Union[float, jnp.ndarray]*) – A scalar or Tensor which the bias term is multiplied by before adding it to `inputs`. Anything which works in the expression `bias * multiplier` is acceptable here. This may be useful if you want to add a bias in one place and subtract the same bias in another place via `multiplier=-1`.

Return type jnp.ndarray

Returns A Tensor of size `[batch_size, input_size1, ...]`.

1.4.2 Pooling

avg_pool(value, window_shape, strides, padding)	Average pool.
AvgPool(window_shape, strides, padding[, ...])	Average pool.
max_pool(value, window_shape, strides, padding)	Max pool.
MaxPool(window_shape, strides, padding[, ...])	Max pool.

Average Pool

haiku.**avg_pool**(*value*, *window_shape*, *strides*, *padding*, *channel_axis=- 1*)
Average pool.

Parameters

- **value** (*jnp.ndarray*) – Value to pool.
- **window_shape** (*Union[int, Sequence[int]]*) – Shape of the pooling window, an int or same rank as value.
- **strides** (*Union[int, Sequence[int]]*) – Strides of the pooling window, an int or same rank as value.
- **padding** (*str*) – Padding algorithm. Either VALID or SAME.
- **channel_axis** (*Optional[int]*) – Axis of the spatial channels for which pooling is skipped, used to infer window_shape or strides if they are an integer.

Return type jnp.ndarray

Returns Pooled result. Same rank as value.

Raises **ValueError** – If the padding is not valid.

class haiku.**AvgPool**(*window_shape*, *strides*, *padding*, *channel_axis=- 1*, *name=None*)
Average pool.

Equivalent to partial application of *avg_pool()*.

__init__(*window_shape*, *strides*, *padding*, *channel_axis=- 1*, *name=None*)
Average pool.

Parameters

- **window_shape** (*Union[int, Sequence[int]]*) – Shape of window to pool over. Same rank as value or int.
- **strides** (*Union[int, Sequence[int]]*) – Strides for the window. Same rank as value or int.
- **padding** (*str*) – Padding algorithm. Either VALID or SAME.
- **channel_axis** (*Optional[int]*) – Axis of the spatial channels for which pooling is skipped.
- **name** (*Optional[str]*) – String name for the module.

__call__(*value*)
Call self as a function.

Return type jnp.ndarray

Max Pool

haiku.**max_pool**(*value*, *window_shape*, *strides*, *padding*, *channel_axis=- 1*)
Max pool.

Parameters

- **value** (*jnp.ndarray*) – Value to pool.
- **window_shape** (*Union[int, Sequence[int]]*) – Shape of the pooling window, an int or same rank as value.
- **strides** (*Union[int, Sequence[int]]*) – Strides of the pooling window, an int or same rank as value.
- **padding** (*str*) – Padding algorithm. Either VALID or SAME.
- **channel_axis** (*Optional[int]*) – Axis of the spatial channels for which pooling is skipped, used to infer window_shape or strides if they are an integer.

Return type jnp.ndarray

Returns Pooled result. Same rank as value.

class haiku.**MaxPool**(*window_shape*, *strides*, *padding*, *channel_axis=- 1*, *name=None*)
Max pool.

Equivalent to partial application of *max_pool()*.

__init__(*window_shape*, *strides*, *padding*, *channel_axis=- 1*, *name=None*)
Max pool.

Parameters

- **window_shape** (*Union[int, Sequence[int]]*) – Shape of window to pool over. Same rank as value or int.
- **strides** (*Union[int, Sequence[int]]*) – Strides for the window. Same rank as value or int.
- **padding** (*str*) – Padding algorithm. Either VALID or SAME.
- **channel_axis** (*Optional[int]*) – Axis of the spatial channels for which pooling is skipped.
- **name** (*Optional[str]*) – String name for the module.

__call__(*value*)
Call self as a function.

Return type jnp.ndarray

1.4.3 Dropout

dropout(rng, rate, x)	Randomly drop units in the input at a given rate.

dropout

haiku.**dropout**(*rng*, *rate*, *x*)

Randomly drop units in the input at a given rate.

See: http://www.cs.toronto.edu/~hinton/absps/dropout.pdf

Parameters

- **rng** (*PRNGKey*) – A JAX random key.
- **rate** (*float*) – Probability that each element of `x` is discarded. Must be a scalar in the range `[0, 1)`.
- **x** (*jnp.ndarray*) – The value to be dropped out.

Return type jnp.ndarray

Returns x, but dropped out and scaled by `1 / (1 - rate)`.

Note: This involves generating *x.size* pseudo-random samples from U([0, 1)) computed with the full precision required to compare them with *rate*. When *rate* is a Python float, this is typically 32 bits, which is often more than what applications require. A work-around is to pass *rate* with a lower precision, e.g. using *np.float16(rate)*.

1.4.4 Combinator

Sequential(layers[, name])	Sequentially calls the given list of layers.

Sequential

class haiku.**Sequential**(*layers*, *name=None*)

Sequentially calls the given list of layers.

Note that *Sequential* is limited in the range of possible architectures it can handle. This is a deliberate design decision; *Sequential* is only meant to be used for the simple case of fusing together modules/ops where the input of a particular module/op is the output of the previous one.

Another restriction is that it is not possible to have extra arguments in the *__call__()* method that are passed to the constituents of the module - for example, if there is a *BatchNorm* module in *Sequential* and the user wishes to switch the `is_training` flag. If this is the desired use case, the recommended solution is to subclass *Module* and implement `__call__`:

```
>>> class CustomModule(hk.Module):
...   def __call__(self, x, is_training):
...     x = hk.Conv2D(32, 4, 2)(x)
...     x = hk.BatchNorm(True, True, 0.9)(x, is_training)
...     x = jax.nn.relu(x)
...     return x
```

__init__(*layers*, *name=None*)

Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*Optional[str]*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

__call__(*inputs*, **args*, ***kwargs*)
Calls all layers sequentially.

1.4.5 Convolutional

ConvND(num_spatial_dims, output_channels, ...)	General N-dimensional convolutional.
Conv1D(output_channels, kernel_shape[, ...])	One dimensional convolution.
Conv2D(output_channels, kernel_shape[, ...])	Two dimensional convolution.
Conv3D(output_channels, kernel_shape[, ...])	Three dimensional convolution.
ConvNDTranspose(num_spatial_dims, ...[, ...])	General n-dimensional transposed convolution (aka.
Conv1DTranspose(output_channels, kernel_shape)	One dimensional transposed convolution (aka.
Conv2DTranspose(output_channels, kernel_shape)	Two dimensional transposed convolution (aka.
Conv3DTranspose(output_channels, kernel_shape)	Three dimensional transposed convolution (aka.
DepthwiseConv2D(channel_multiplier, kernel_shape)	Two dimensional convolution.
DepthwiseConv2D(channel_multiplier, kernel_shape)	Two dimensional convolution.
get_channel_index(data_format)	Returns the channel index when given a valid data format.

ConvND

class haiku.**ConvND**(*num_spatial_dims*, *output_channels*, *kernel_shape*, *stride=1*, *rate=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='channels_last'*, *mask=None*, *feature_group_count=1*, *name=None*)
General N-dimensional convolutional.

__init__(*num_spatial_dims*, *output_channels*, *kernel_shape*, *stride=1*, *rate=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='channels_last'*, *mask=None*, *feature_group_count=1*, *name=None*)
Initializes the module.

Parameters

- **num_spatial_dims** (*int*) – The number of spatial dimensions of the input.
- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length `num_spatial_dims`.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length `num_spatial_dims`. Defaults to 1.
- **rate** (*Union[int, Sequence[int]]*) – Optional kernel dilation rate. Either an integer or a sequence of length `num_spatial_dims`. 1 corresponds to standard ND convolution, `rate > 1` corresponds to dilated convolution. Defaults to 1.
- **padding** (*Union[str, Sequence[Tuple[int, int]], hk.pad.PadFn, Sequence[hk.pad.PadFn]]*) – Optional padding algorithm. Either `VALID` or `SAME` or a sequence of n `(low, high)` integer pairs that give the padding to apply before and after each spatial dimension. or a callable or sequence of callables of size

num_spatial_dims. Any callables must take a single integer argument equal to the effective kernel size and return a sequence of two integers representing the padding before and after. See haiku.pad.* for more details and example functions. Defaults to SAME. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.

- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Can be either channels_first, channels_last, N...C or NC.... By default, channels_last. See *get_channel_index()*.
- **mask** (*Optional[jnp.ndarray]*) – Optional mask of the weights.
- **feature_group_count** (*int*) – Optional number of groups in group convolution. Default value of 1 corresponds to normal dense convolution. If a higher value is used, convolutions are applied separately to that many groups, then stacked together. This reduces the number of parameters and possibly the compute for a given output_channels. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **name** (*Optional[str]*) – The name of the module.

__call__(*inputs*, ***, *precision=None*)

Connects ConvND layer.

Parameters

- **inputs** (*jnp.ndarray*) – An array of shape [spatial_dims, C] and rank-N+1 if unbatched, or an array of shape [N, spatial_dims, C] and rank-N+2 if batched.
- **precision** (*Optional[lax.Precision]*) – Optional jax.lax.Precision to pass to jax.lax.conv_general_dilated().

Return type jnp.ndarray

Returns

An array of shape [spatial_dims, output_channels] and rank-N+1 if unbatched, or an array of shape [N, spatial_dims, output_channels] and rank-N+2 if batched.

Conv1D

class haiku.**Conv1D**(*output_channels*, *kernel_shape*, *stride=1*, *rate=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NWC'*, *mask=None*, *feature_group_count=1*, *name=None*)

One dimensional convolution.

__init__(*output_channels*, *kernel_shape*, *stride=1*, *rate=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NWC'*, *mask=None*, *feature_group_count=1*, *name=None*)

Initializes the module.

Parameters

- **output_channels** (*int*) – Number of output channels.

- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length 1.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length 1. Defaults to 1.
- **rate** (*Union[int, Sequence[int]]*) – Optional kernel dilation rate. Either an integer or a sequence of length 1. 1 corresponds to standard ND convolution, `rate > 1` corresponds to dilated convolution. Defaults to 1.
- **padding** (*Union[str, Sequence[Tuple[int, int]], hk.pad.PadFn, Sequence[hk.pad.PadFn]]*) – Optional padding algorithm. Either `VALID` or `SAME` or a callable or sequence of callables of length 1. Any callables must take a single integer argument equal to the effective kernel size and return a list of two integers representing the padding before and after. See haiku.pad.* for more details and example functions. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Either `NWC` or `NCW`. By default, `NWC`.
- **mask** (*Optional[jnp.ndarray]*) – Optional mask of the weights.
- **feature_group_count** (*int*) – Optional number of groups in group convolution. Default value of 1 corresponds to normal dense convolution. If a higher value is used, convolutions are applied separately to that many groups, then stacked together. This reduces the number of parameters and possibly the compute for a given `output_channels`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **name** (*Optional[str]*) – The name of the module.

Conv2D

class haiku.**Conv2D**(*output_channels*, *kernel_shape*, *stride=1*, *rate=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NHWC'*, *mask=None*, *feature_group_count=1*, *name=None*)

Two dimensional convolution.

__init__(*output_channels*, *kernel_shape*, *stride=1*, *rate=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NHWC'*, *mask=None*, *feature_group_count=1*, *name=None*)

Initializes the module.

Parameters

- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length 2.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length 2. Defaults to 1.

- **rate** (*Union[int, Sequence[int]]*) – Optional kernel dilation rate. Either an integer or a sequence of length 2. 1 corresponds to standard ND convolution, `rate > 1` corresponds to dilated convolution. Defaults to 1.
- **padding** (*Union[str, Sequence[Tuple[int, int]], hk.pad.PadFn, Sequence[hk.pad.PadFn]]*) – Optional padding algorithm. Either `VALID` or `SAME` or a callable or sequence of callables of length 2. Any callables must take a single integer argument equal to the effective kernel size and return a list of two integers representing the padding before and after. See haiku.pad.* for more details and example functions. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Either `NHWC` or `NCHW`. By default, `NHWC`.
- **mask** (*Optional[jnp.ndarray]*) – Optional mask of the weights.
- **feature_group_count** (*int*) – Optional number of groups in group convolution. Default value of 1 corresponds to normal dense convolution. If a higher value is used, convolutions are applied separately to that many groups, then stacked together. This reduces the number of parameters and possibly the compute for a given `output_channels`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **name** (*Optional[str]*) – The name of the module.

Conv3D

class haiku.**Conv3D**(*output_channels*, *kernel_shape*, *stride=1*, *rate=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NDHWC'*, *mask=None*, *feature_group_count=1*, *name=None*)

Three dimensional convolution.

__init__(*output_channels*, *kernel_shape*, *stride=1*, *rate=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NDHWC'*, *mask=None*, *feature_group_count=1*, *name=None*)

Initializes the module.

Parameters

- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length 3.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length 3. Defaults to 1.
- **rate** (*Union[int, Sequence[int]]*) – Optional kernel dilation rate. Either an integer or a sequence of length 3. 1 corresponds to standard ND convolution, *rate > 1* corresponds to dilated convolution. Defaults to 1.
- **padding** (*Union[str, Sequence[Tuple[int, int]], hk.pad.PadFn, Sequence[hk.pad.PadFn]]*) – Optional padding algorithm. Either `VALID` or `SAME`

or a callable or sequence of callables of length 3. Any callables must take a single integer argument equal to the effective kernel size and return a list of two integers representing the padding before and after. See haiku.pad.* for more details and example functions. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.

- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Either `NDHWC` or `NCDHW`. By default, `NDHWC`.
- **mask** (*Optional[jnp.ndarray]*) – Optional mask of the weights.
- **feature_group_count** (*int*) – Optional number of groups in group convolution. Default value of 1 corresponds to normal dense convolution. If a higher value is used, convolutions are applied separately to that many groups, then stacked together. This reduces the number of parameters and possibly the compute for a given `output_channels`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **name** (*Optional[str]*) – The name of the module.

ConvNDTranspose

class haiku.**ConvNDTranspose**(*num_spatial_dims*, *output_channels*, *kernel_shape*, *stride=1*, *output_shape=None*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='channels_last'*, *mask=None*, *name=None*)

General n-dimensional transposed convolution (aka. deconvolution).

__init__(*num_spatial_dims*, *output_channels*, *kernel_shape*, *stride=1*, *output_shape=None*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='channels_last'*, *mask=None*, *name=None*)

Initializes the module.

Parameters

- **num_spatial_dims** (*int*) – The number of spatial dimensions of the input.
- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length `num_spatial_dims`.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length `num_spatial_dims`. Defaults to 1.
- **output_shape** (*Optional[Union[int, Sequence[int]]]*) – Output shape of the spatial dimensions of a transpose convolution. Can be either an integer or an iterable of integers. If a *None* value is given, a default shape is automatically calculated.
- **padding** (*Union[str, Sequence[Tuple[int, int]]]*) – Optional padding algorithm. Either "VALID" or "SAME". Defaults to "SAME". See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.

- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Can be either `channels_first`, `channels_last`, `N...C` or `NC...`. By default, `channels_last`.
- **mask** (*Optional[jnp.ndarray]*) – Optional mask of the weights.
- **name** (*Optional[str]*) – The name of the module.

__call__(*inputs*, *, *precision=None*)

Computes the transposed convolution of the input.

Parameters

- **inputs** (*jnp.ndarray*) – An array of shape `[spatial_dims, C]` and rank-N+1 if unbatched, or an array of shape `[N, spatial_dims, C]` and rank-N+2 if batched.
- **precision** (*Optional[lax.Precision]*) – Optional `jax.lax.Precision` to pass to `jax.lax.conv_transpose()`.

Return type jnp.ndarray

Returns

An array of shape `[spatial_dims, output_channels]` and rank-N+1 if unbatched, or an array of shape `[N, spatial_dims, output_channels]` and rank-N+2 if batched.

Conv1DTranspose

class haiku.**Conv1DTranspose**(*output_channels*, *kernel_shape*, *stride=1*, *output_shape=None*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NWC'*, *mask=None*, *name=None*)

One dimensional transposed convolution (aka. deconvolution).

__init__(*output_channels*, *kernel_shape*, *stride=1*, *output_shape=None*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NWC'*, *mask=None*, *name=None*)

Initializes the module.

Parameters

- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length 1.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length 1. Defaults to 1.
- **output_shape** (*Optional[Union[int, Sequence[int]]]*) – Output shape of the spatial dimensions of a transpose convolution. Can be either an integer or an iterable of integers. If a *None* value is given, a default shape is automatically calculated.
- **padding** (*Union[str, Sequence[Tuple[int, int]]]*) – Optional padding algorithm. Either `VALID` or `SAME`. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.

- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Either `NWC` or `NCW`. By default, `NWC`.
- **mask** (*Optional[jnp.ndarray]*) – Optional mask of the weights.
- **name** (*Optional[str]*) – The name of the module.

Conv2DTranspose

class haiku.**Conv2DTranspose**(*output_channels*, *kernel_shape*, *stride=1*, *output_shape=None*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NHWC'*, *mask=None*, *name=None*)

Two dimensional transposed convolution (aka. deconvolution).

__init__(*output_channels*, *kernel_shape*, *stride=1*, *output_shape=None*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NHWC'*, *mask=None*, *name=None*)

Initializes the module.

Parameters

- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length 2.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length 2. Defaults to 1.
- **output_shape** (*Optional[Union[int, Sequence[int]]]*) – Output shape of the spatial dimensions of a transpose convolution. Can be either an integer or an iterable of integers. If a *None* value is given, a default shape is automatically calculated.
- **padding** (*Union[str, Sequence[Tuple[int, int]]]*) – Optional padding algorithm. Either `VALID` or `SAME`. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Either `NHWC` or `NCHW`. By default, `NHWC`.
- **mask** (*Optional[jnp.ndarray]*) – Optional mask of the weights.
- **name** (*Optional[str]*) – The name of the module.

Conv3DTranspose

class haiku.**Conv3DTranspose**(*output_channels*, *kernel_shape*, *stride=1*, *output_shape=None*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NDHWC'*, *mask=None*, *name=None*)

Three dimensional transposed convolution (aka. deconvolution).

__init__(*output_channels*, *kernel_shape*, *stride=1*, *output_shape=None*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NDHWC'*, *mask=None*, *name=None*)

Initializes the module.

Parameters

- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length 3.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length 3. Defaults to 1.
- **output_shape** (*Optional[Union[int, Sequence[int]]]*) – Output shape of the spatial dimensions of a transpose convolution. Can be either an integer or an iterable of integers. If a *None* value is given, a default shape is automatically calculated.
- **padding** (*Union[str, Sequence[Tuple[int, int]]]*) – Optional padding algorithm. Either `VALID` or `SAME`. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Either `NDHWC` or `NCDHW`. By default, `NDHWC`.
- **mask** (*Optional[jnp.ndarray]*) – Optional mask of the weights.
- **name** (*Optional[str]*) – The name of the module.

DepthwiseConv1D

class haiku.**DepthwiseConv1D**(*channel_multiplier*, *kernel_shape*, *stride=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NWC'*, *name=None*)

One dimensional convolution.

__init__(*channel_multiplier*, *kernel_shape*, *stride=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NWC'*, *name=None*)

Construct a 1D Depthwise Convolution.

Parameters

- **channel_multiplier** (*int*) – Multiplicity of output channels. To keep the number of output channels the same as the number of input channels, set 1.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length 1.

- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length 1. Defaults to 1.
- **padding** (*Union[str, Sequence[Tuple[int, int]]]*) – Optional padding algorithm. Either `VALID`, `SAME` or a sequence of `before, after` pairs. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Can be either `channels_first`, `channels_last`, `N...C` or `NC...`. By default, `channels_last`. See *get_channel_index()*.
- **name** (*Optional[str]*) – The name of the module.

DepthwiseConv2D

class haiku.**DepthwiseConv2D**(*channel_multiplier*, *kernel_shape*, *stride=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NHWC'*, *name=None*)

Two dimensional convolution.

__init__(*channel_multiplier*, *kernel_shape*, *stride=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NHWC'*, *name=None*)

Construct a 2D Depthwise Convolution.

Parameters

- **channel_multiplier** (*int*) – Multiplicity of output channels. To keep the number of output channels the same as the number of input channels, set 1.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length 2.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length 2. Defaults to 1.
- **padding** (*Union[str, Sequence[Tuple[int, int]]]*) – Optional padding algorithm. Either `VALID`, `SAME` or a sequence of `before, after` pairs. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Can be either `channels_first`, `channels_last`, `N...C` or `NC...`. By default, `channels_last`. See *get_channel_index()*.
- **name** (*Optional[str]*) – The name of the module.

DepthwiseConv3D

class haiku.**DepthwiseConv3D**(*channel_multiplier*, *kernel_shape*, *stride=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NDHWC'*, *name=None*)

Three dimensional convolution.

__init__(*channel_multiplier*, *kernel_shape*, *stride=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NDHWC'*, *name=None*)

Construct a 3D Depthwise Convolution.

Parameters

- **channel_multiplier** (*int*) – Multiplicity of output channels. To keep the number of output channels the same as the number of input channels, set 1.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length 3.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length 3. Defaults to 1.
- **padding** (*Union[str, Sequence[Tuple[int, int]]]*) – Optional padding algorithm. Either `VALID`, `SAME` or a sequence of `before, after` pairs. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Can be either `channels_first`, `channels_last`, `N...C` or `NC...`. By default, `channels_last`. See *get_channel_index()*.
- **name** (*Optional[str]*) – The name of the module.

SeparableDepthwiseConv2D

class haiku.**SeparableDepthwiseConv2D**(*channel_multiplier*, *kernel_shape*, *stride=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NHWC'*, *name=None*)

Separable 2-D Depthwise Convolution Module.

__init__(*channel_multiplier*, *kernel_shape*, *stride=1*, *padding='SAME'*, *with_bias=True*, *w_init=None*, *b_init=None*, *data_format='NHWC'*, *name=None*)

Construct a Separable 2D Depthwise Convolution module.

Parameters

- **channel_multiplier** (*int*) – Multiplicity of output channels. To keep the number of output channels the same as the number of input channels, set 1.
- **kernel_shape** (*Union[int, Sequence[int]]*) – The shape of the kernel. Either an integer or a sequence of length `num_spatial_dims`.
- **stride** (*Union[int, Sequence[int]]*) – Optional stride for the kernel. Either an integer or a sequence of length `num_spatial_dims`. Defaults to 1.

- **padding** (*Union[str, Sequence[Tuple[int, int]]]*) – Optional padding algorithm. Either `VALID`, `SAME` or a sequence of `before, after` pairs. Defaults to `SAME`. See: https://www.tensorflow.org/xla/operation_semantics#conv_convolution.
- **with_bias** (*bool*) – Whether to add a bias. By default, true.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Optional weight initialization. By default, truncated normal.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Optional bias initialization. By default, zeros.
- **data_format** (*str*) – The data format of the input. Can be either `channels_first`, `channels_last`, `N...C` or `NC....` By default, `channels_last`.
- **name** (*Optional[str]*) – The name of the module.

__call__(*inputs*)
Call self as a function.

Return type jnp.ndarray

get_channel_index

haiku.**get_channel_index**(*data_format*)
Returns the channel index when given a valid data format.

```
>>> hk.get_channel_index('channels_last')
-1
>>> hk.get_channel_index('channels_first')
1
>>> hk.get_channel_index('N...C')
-1
>>> hk.get_channel_index('NCHW')
1
```

Parameters **data_format** (*str*) – String, the data format to get the channel index from. Valid data formats are spatial (e.g.``NCHW``), sequential (e.g. `BTHWD`), `channels_first` and `channels_last`).

Return type int

Returns The channel index as an int, either `1` or `-1`.

Raises **ValueError** – If the data format is unrecognised.

1.4.6 Normalization

BatchNorm(create_scale, create_offset, ...)	Normalizes inputs to maintain a mean of ~0 and stddev of ~1.
GroupNorm(groups[, axis, create_scale, ...])	Group normalization module.
InstanceNorm(create_scale, create_offset[, ...])	Normalizes inputs along the spatial dimensions.
LayerNorm(axis, create_scale, create_offset)	LayerNorm module.
RMSNorm(axis[, eps, scale_init, name])	RMSNorm module.
SpectralNorm([eps, n_steps, name])	Normalizes an input by its first singular value.
ExponentialMovingAverage(decay[, ...])	Maintains an exponential moving average.

continues on next page

Table 11 – continued from previous page

SNParamsTree([eps, n_steps, ignore_regex, name])	Applies Spectral Normalization to all parameters in a tree.
EMAParamsTree(decay[, zero_debias, ...])	Maintains an exponential moving average for all parameters in a tree.

BatchNorm

class haiku.**BatchNorm**(*create_scale*, *create_offset*, *decay_rate*, *eps=1e-05*, *scale_init=None*, *offset_init=None*, *axis=None*, *cross_replica_axis=None*, *cross_replica_axis_index_groups=None*, *data_format='channels_last'*, *name=None*)

Normalizes inputs to maintain a mean of ~0 and stddev of ~1.

See: https://arxiv.org/abs/1502.03167.

There are many different variations for how users want to manage scale and offset if they require them at all. These are:

- No scale/offset in which case `create_*` should be set to `False` and `scale`/`offset` aren't passed when the module is called.
- Trainable scale/offset in which case `create_*` should be set to `True` and again `scale`/`offset` aren't passed when the module is called. In this case this module creates and owns the `scale`/`offset` variables.
- Externally generated `scale`/`offset`, such as for conditional normalization, in which case `create_*` should be set to `False` and then the values fed in at call time.

NOTE: `jax.vmap(hk.transform(BatchNorm))` will update summary statistics and normalize values on a per-batch basis; we currently do *not* support normalizing across a batch axis introduced by vmap.

__init__(*create_scale*, *create_offset*, *decay_rate*, *eps=1e-05*, *scale_init=None*, *offset_init=None*, *axis=None*, *cross_replica_axis=None*, *cross_replica_axis_index_groups=None*, *data_format='channels_last'*, *name=None*)

Constructs a BatchNorm module.

Parameters

- **create_scale** (*bool*) – Whether to include a trainable scaling factor.
- **create_offset** (*bool*) – Whether to include a trainable offset.
- **decay_rate** (*float*) – Decay rate for EMA.
- **eps** (*float*) – Small epsilon to avoid division by zero variance. Defaults `1e-5`, as in the paper and Sonnet.
- **scale_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for gain (aka scale). Can only be set if `create_scale=True`. By default, `1`.
- **offset_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for bias (aka offset). Can only be set if `create_offset=True`. By default, `0`.
- **axis** (*Optional[Sequence[int]]*) – Which axes to reduce over. The default (`None`) signifies that all but the channel axis should be normalized. Otherwise this is a list of axis indices which will have normalization statistics calculated.
- **cross_replica_axis** (*Optional[str]*) – If not `None`, it should be a string representing the axis name over which this module is being run within a `jax.pmap`. Supplying this argument means that batch statistics are calculated across all replicas on that axis.

- **cross_replica_axis_index_groups** (*Optional[Sequence[Sequence[int]]]*) – Specifies how devices are grouped.
- **data_format** (*str*) – The data format of the input. Can be either `channels_first`, `channels_last`, `N...C` or `NC...`. By default it is `channels_last`. See *get_channel_index()*.
- **name** (*Optional[str]*) – The module name.

__call__ (*inputs*, *is_training*, *test_local_stats=False*, *scale=None*, *offset=None*)

Computes the normalized version of the input.

Parameters

- **inputs** (*jnp.ndarray*) – An array, where the data format is `[..., C]`.
- **is_training** (*bool*) – Whether this is during training.
- **test_local_stats** (*bool*) – Whether local stats are used when is_training=False.
- **scale** (*Optional[jnp.ndarray]*) – An array up to n-D. The shape of this tensor must be broadcastable to the shape of `inputs`. This is the scale applied to the normalized inputs. This cannot be passed in if the module was constructed with `create_scale=True`.
- **offset** (*Optional[jnp.ndarray]*) – An array up to n-D. The shape of this tensor must be broadcastable to the shape of `inputs`. This is the offset applied to the normalized inputs. This cannot be passed in if the module was constructed with `create_offset=True`.

Return type jnp.ndarray

Returns The array, normalized across all but the last dimension.

GroupNorm

class haiku.**GroupNorm**(*groups*, *axis=slice(1, None, None)*, *create_scale=True*, *create_offset=True*, *eps=1e-05*, *scale_init=None*, *offset_init=None*, *data_format='channels_last'*, *name=None*)

Group normalization module.

This applies group normalization to the x. This involves splitting the channels into groups before calculating the mean and variance. The default behaviour is to compute the mean and variance over the spatial dimensions and the grouped channels. The mean and variance will never be computed over the created groups axis.

It transforms the input `x` into:

$$\text{outputs} = \text{scale}\,\frac{x - \mu}{\sigma + \epsilon} + \text{offset}$$

Where μ and σ are respectively the mean and standard deviation of `x`.

There are many different variations for how users want to manage scale and offset if they require them at all. These are:

- No `scale`/`offset` in which case `create_*` should be set to `False` and `scale`/`offset` aren't passed when the module is called.
- Trainable `scale`/`offset` in which case create_* should be set to `True` and again `scale`/`offset` aren't passed when the module is called. In this case this module creates and owns the scale/offset parameters.

- Externally generated `scale`/`offset`, such as for conditional normalization, in which case `create_*` should be set to `False` and then the values fed in at call time.

__init__(*groups*, *axis=slice(1, None, None)*, *create_scale=True*, *create_offset=True*, *eps=1e-05*, *scale_init=None*, *offset_init=None*, *data_format='channels_last'*, *name=None*)

Constructs a `GroupNorm` module.

Parameters

- **groups** (*int*) – number of groups to divide the channels by. The number of channels must be divisible by this.
- **axis** (*Union[int, slice, Sequence[int]]*) – `int`, `slice` or sequence of ints representing the axes which should be normalized across. By default this is all but the first dimension. For time series data use *slice(2, None)* to average over the none Batch and Time data.
- **create_scale** (*bool*) – whether to create a trainable scale per channel applied after the normalization.
- **create_offset** (*bool*) – whether to create a trainable offset per channel applied after normalization and scaling.
- **eps** (*float*) – Small epsilon to add to the variance to avoid division by zero. Defaults to `1e-5`.
- **scale_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for the scale parameter. Can only be set if `create_scale=True`. By default scale is initialized to `1`.
- **offset_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for the offset parameter. Can only be set if `create_offset=True`. By default offset is initialized to `0`.
- **data_format** (*str*) – The data format of the input. Can be either `channels_first`, `channels_last`, `N...C` or `NC...`. By default it is `channels_last`. See *get_channel_index()*.
- **name** (*Optional[str]*) – Name of the module.

__call__(*x*, *scale=None*, *offset=None*)

Returns normalized inputs.

Parameters

- **x** (*jnp.ndarray*) – An n-D tensor of the `data_format` specified in the constructor on which the transformation is performed.
- **scale** (*Optional[jnp.ndarray]*) – A tensor up to n-D. The shape of this tensor must be broadcastable to the shape of `x`. This is the scale applied to the normalized x. This cannot be passed in if the module was constructed with `create_scale=True`.
- **offset** (*Optional[jnp.ndarray]*) – A tensor up to n-D. The shape of this tensor must be broadcastable to the shape of `x`. This is the offset applied to the normalized `x`. This cannot be passed in if the module was constructed with `create_offset=True`.

Return type jnp.ndarray

Returns An n-d tensor of the same shape as x that has been normalized.

InstanceNorm

class haiku.**InstanceNorm**(*create_scale*, *create_offset*, *eps=1e-05*, *scale_init=None*, *offset_init=None*, *data_format='channels_last'*, *name=None*)

Normalizes inputs along the spatial dimensions.

See *LayerNorm* for more details.

__init__(*create_scale*, *create_offset*, *eps=1e-05*, *scale_init=None*, *offset_init=None*, *data_format='channels_last'*, *name=None*)

Constructs an *InstanceNorm* module.

This method creates a module which normalizes over the spatial dimensions.

Parameters

- **create_scale** (*bool*) – `bool` representing whether to create a trainable scale per channel applied after the normalization.
- **create_offset** (*bool*) – `bool` representing whether to create a trainable offset per channel applied after normalization and scaling.
- **eps** (*float*) – Small epsilon to avoid division by zero variance. Defaults to `1e-5`.
- **scale_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for the scale variable. Can only be set if `create_scale=True`. By default scale is initialized to `1`.
- **offset_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for the offset variable. Can only be set if `create_offset=True`. By default offset is initialized to `0`.
- **data_format** (*str*) – The data format of the input. Can be either `channels_first`, `channels_last`, `N...C` or `NC...`. By default it is `channels_last`. See *get_channel_index()*.
- **name** (*Optional[str]*) – Name of the module.

LayerNorm

class haiku.**LayerNorm**(*axis*, *create_scale*, *create_offset*, *eps=1e-05*, *scale_init=None*, *offset_init=None*, *use_fast_variance=False*, *name=None*, ***, *param_axis=None*)

LayerNorm module.

See: https://arxiv.org/abs/1607.06450.

Example usage:

```
>>> ln = hk.LayerNorm(axis=-1, param_axis=-1,
...                   create_scale=True, create_offset=True)
>>> x = ln(jnp.ones([8, 224, 224, 3]))
```

__init__(*axis*, *create_scale*, *create_offset*, *eps=1e-05*, *scale_init=None*, *offset_init=None*, *use_fast_variance=False*, *name=None*, ***, *param_axis=None*)

Constructs a LayerNorm module.

Parameters

- **axis** (*AxisOrAxes*) – Integer, list of integers, or slice indicating which axes to normalize over. Note that the shape of the scale/offset parameters are controlled by the `param_axis` argument.

- **create_scale** (*bool*) – Bool, defines whether to create a trainable scale per channel applied after the normalization.
- **create_offset** (*bool*) – Bool, defines whether to create a trainable offset per channel applied after normalization and scaling.
- **eps** (*float*) – Small epsilon to avoid division by zero variance. Defaults `1e-5`, as in the paper and Sonnet.
- **scale_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for gain (aka scale). By default, one.
- **offset_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for bias (aka offset). By default, zero.
- **use_fast_variance** (*bool*) – If true, use a faster but less numerically stable formulation for computing variance.
- **name** (*Optional[str]*) – The module name.
- **param_axis** (*Optional[AxisOrAxes]*) – Axis used to determine the parameter shape of the learnable scale/offset. Sonnet sets this to the channel/feature axis (e.g. to `-1` for `NHWC`). Other libraries set this to the same as the reduction axis (e.g. `axis=param_axis`).

__call__(*inputs*, *scale=None*, *offset=None*)

Connects the layer norm.

Parameters

- **inputs** (*jnp.ndarray*) – An array, where the data format is `[N, ..., C]`.
- **scale** (*Optional[jnp.ndarray]*) – An array up to n-D. The shape of this tensor must be broadcastable to the shape of `inputs`. This is the scale applied to the normalized inputs. This cannot be passed in if the module was constructed with `create_scale=True`.
- **offset** (*Optional[jnp.ndarray]*) – An array up to n-D. The shape of this tensor must be broadcastable to the shape of `inputs`. This is the offset applied to the normalized inputs. This cannot be passed in if the module was constructed with `create_offset=True`.

Return type jnp.ndarray

Returns The array, normalized.

RMSNorm

class haiku.**RMSNorm**(*axis*, *eps=1e-05*, *scale_init=None*, *name=None*)

RMSNorm module.

RMSNorm provides an alternative that can be both faster and more stable than LayerNorm. The inputs are normalized by the root-mean-squared (RMS) and scaled by a learned parameter, but they are not recentered around their mean.

See https://arxiv.org/pdf/1910.07467.pdf

__init__(*axis*, *eps=1e-05*, *scale_init=None*, *name=None*)

Constructs a RMSNorm modulke.

Parameters

- **axis** (*Union[int, Sequence[int], slice]*) – Integer, list of integers, or slice indicating which axes to normalize over.
- **eps** (*float*) – Small epsilon to avoid division by zero variance. Defaults to 1e-5.
- **scale_init** (*Optional[hk.initializers.Initializer]*) – Optional initializer for gain (aka scale). By default, one.
- **name** (*Optional[str]*) – The module name.

__call__(*inputs*)
Connects the layer norm.

Parameters **inputs** (*jnp.ndarray*) – An array, where the data format is `[N, ..., C]`.

Returns The normalized array, of the same shape as the inputs..

SpectralNorm

class haiku.**SpectralNorm**(*eps=0.0001*, *n_steps=1*, *name=None*)
Normalizes an input by its first singular value.

This module uses power iteration to calculate this value based on the input and an internal hidden state.

__init__(*eps=0.0001*, *n_steps=1*, *name=None*)
Initializes an SpectralNorm module.

Parameters

- **eps** (*float*) – The constant used for numerical stability.
- **n_steps** (*int*) – How many steps of power iteration to perform to approximate the singular value of the input.
- **name** (*Optional[str]*) – The name of the module.

__call__(*value*, *update_stats=True*, *error_on_non_matrix=False*)
Performs Spectral Normalization and returns the new value.

Parameters

- **value** – The array-like object for which you would like to perform an spectral normalization on.
- **update_stats** (*bool*) – A boolean defaulting to True. Regardless of this arg, this function will return the normalized input. When *update_stats* is True, the internal state of this object will also be updated to reflect the input value. When *update_stats* is False the internal stats will remain unchanged.
- **error_on_non_matrix** (*bool*) – Spectral normalization is only defined on matrices. By default, this module will return scalars unchanged and flatten higher-order tensors in their leading dimensions. Setting this flag to True will instead throw errors in those cases.

Return type jnp.ndarray

Returns The input value normalized by it's first singular value.

Raises **ValueError** – If *error_on_non_matrix* is True and *value* has ndims > 2.

ExponentialMovingAverage

class haiku.**ExponentialMovingAverage**(*decay*, *zero_debias=True*, *warmup_length=0*, *name=None*)

Maintains an exponential moving average.

This uses the Adam debiasing procedure. See https://arxiv.org/pdf/1412.6980.pdf for details.

__init__(*decay*, *zero_debias=True*, *warmup_length=0*, *name=None*)

Initializes an ExponentialMovingAverage module.

Parameters

- **decay** – The chosen decay. Must in `[0, 1)`. Values close to 1 result in slow decay; values close to `0` result in fast decay.
- **zero_debias** (*bool*) – Whether to run with zero-debiasing.
- **warmup_length** (*int*) – A positive integer, EMA has no effect until the internal counter has reached *warmup_length* at which point the initial value for the decaying average is initialized to the input value after *warmup_length* iterations.
- **name** (*Optional[str]*) – The name of the module.

initialize(*shape*, *dtype=<class 'jax._src.numpy.lax_numpy.float32'>*)

If uninitialized sets the average to `zeros` of the given shape/dtype.

__call__(*value*, *update_stats=True*)

Updates the EMA and returns the new value.

Parameters

- **value** (*jnp.ndarray*) – The array-like object for which you would like to perform an exponential decay on.
- **update_stats** (*bool*) – A Boolean, whether to update the internal state of this object to reflect the input value. When *update_stats* is False the internal stats will remain unchanged.

Return type jnp.ndarray

Returns The exponentially weighted average of the input value.

SNParamsTree

class haiku.**SNParamsTree**(*eps=0.0001*, *n_steps=1*, *ignore_regex=''*, *name=None*)

Applies Spectral Normalization to all parameters in a tree.

This is isomorphic to EMAParamsTree in moving_averages.py.

__init__(*eps=0.0001*, *n_steps=1*, *ignore_regex=''*, *name=None*)

Initializes an SNParamsTree module.

Parameters

- **eps** (*float*) – The constant used for numerical stability.
- **n_steps** (*int*) – How many steps of power iteration to perform to approximate the singular value of the input.
- **ignore_regex** (*str*) – A string. Any parameter in the tree whose name matches this regex will not have spectral normalization applied to it. The empty string means this module applies to all parameters.

- **name** (*Optional[str]*) – The name of the module.

__call__(*tree*, *update_stats=True*)
Call self as a function.

EMAParamsTree

class haiku.**EMAParamsTree**(*decay*, *zero_debias=True*, *warmup_length=0*, *ignore_regex=''*, *name=None*)
Maintains an exponential moving average for all parameters in a tree.

While ExponentialMovingAverage is meant to be applied to single parameters within a function, this class is meant to be applied to the entire tree of parameters for a function.

Given a set of parameters for some network:

```
>>> network_fn = lambda x: hk.Linear(10)(x)
>>> x = jnp.ones([1, 1])
>>> params = hk.transform(network_fn).init(jax.random.PRNGKey(428), x)
```

You might use the EMAParamsTree like follows:

```
>>> ema_fn = hk.transform_with_state(lambda x: hk.EMAParamsTree(0.2)(x))
>>> _, ema_state = ema_fn.init(None, params)
>>> ema_params, ema_state = ema_fn.apply(None, ema_state, None, params)
```

Here, we are transforming a Haiku function and constructing its parameters via an init_fn as normal, but are creating a second transformed function which expects a tree of parameters as input. This function is then called with the current parameters as input, which then returns an identical tree with every parameter replaced with its exponentially decayed average. This ema_params object can then be passed into the *network_fn* as usual, and will cause it to run with EMA weights.

__init__(*decay*, *zero_debias=True*, *warmup_length=0*, *ignore_regex=''*, *name=None*)
Initializes an EMAParamsTree module.

Parameters

- **decay** – The chosen decay. Must in `[0, 1)`. Values close to `1` result in slow decay; values close to `0` result in fast decay.
- **zero_debias** (*bool*) – Whether to run with zero-debiasing.
- **warmup_length** (*int*) – A positive integer, EMA has no effect until the internal counter has reached *warmup_length* at which point the initial value for the decaying average is initialized to the input value after *warmup_length* iterations.
- **ignore_regex** (*str*) – A string. Any parameter in the tree whose name matches this regex will not have any moving average applied to it. The empty string means this module will EMA all parameters.
- **name** (*Optional[str]*) – The name of the module.

__call__(*tree*, *update_stats=True*)
Call self as a function.

1.4.7 Recurrent

RNNCore([name])	Base class for RNN cores.
dynamic_unroll(core, input_sequence, ... [, ...])	Performs a dynamic unroll of an RNN.
static_unroll(core, input_sequence, ... [, ...])	Performs a static unroll of an RNN.
expand_apply(f[, axis])	Wraps f to temporarily add a size-1 axis to its inputs.
VanillaRNN(hidden_size[, double_bias, name])	Basic fully-connected RNN core.
LSTM(hidden_size[, name])	Long short-term memory (LSTM) RNN core.
GRU(hidden_size[, w_i_init, w_h_init, ...])	Gated Recurrent Unit.
DeepRNN(layers[, name])	Wraps a sequence of cores and callables as a single core.
deep_rnn_with_skip_connections(layers[, name])	Constructs a *DeepRNN* with skip connections.
ResetCore(core[, name])	A wrapper for managing state resets during unrolls.
IdentityCore([name])	A recurrent core that forwards the inputs and an empty state.
Conv1DLSTM(input_shape, output_channels, ...)	1-D convolutional LSTM.
Conv2DLSTM(input_shape, output_channels, ...)	2-D convolutional LSTM.
Conv3DLSTM(input_shape, output_channels, ...)	3-D convolutional LSTM.

RNNCore

class haiku.**RNNCore**(*name=None*)

Base class for RNN cores.

This class defines the basic functionality that every core should implement: *initial_state()*, used to construct an example of the core state; and *__call__()* which applies the core parameterized by a previous state to an input.

Cores may be used with *dynamic_unroll()* and *static_unroll()* to iteratively construct an output sequence from the given input sequence.

abstract __call__(*inputs*, *prev_state*)

Run one step of the RNN.

Parameters

- **inputs** – An arbitrarily nested structure.
- **prev_state** – Previous core state.

Return type Tuple[Any, Any]

Returns A tuple with two elements `output, next_state`. `output` is an arbitrarily nested structure. `next_state` is the next core state, this must be the same shape as `prev_state`.

abstract initial_state(*batch_size*)

Constructs an initial state for this core.

Parameters **batch_size** (*Optional[int]*) – Optional int or an integral scalar tensor representing batch size. If None, the core may either fail or (experimentally) return an initial state without a batch dimension.

Returns Arbitrarily nested initial state for this core.

dynamic_unroll

haiku.**dynamic_unroll**(*core*, *input_sequence*, *initial_state*, *time_major=True*, *reverse=False*)

Performs a dynamic unroll of an RNN.

An *unroll* corresponds to calling the core on each element of the input sequence in a loop, carrying the state through:

```
state = initial_state
for t in range(len(input_sequence)):
   outputs, state = core(input_sequence[t], state)
```

A *dynamic* unroll preserves the loop structure when executed inside `jax.jit()`. See *static_unroll()* for an unroll function which replaces a loop with its body repeated multiple times.

Parameters

- **core** – An *RNNCore* to unroll.
- **input_sequence** – An arbitrarily nested structure of tensors of shape `[T, ...]` if time-major=True, or `[B, T, ...]` if time_major=False, where `T` is the number of time steps.
- **initial_state** – An initial state of the given core.
- **time_major** – If True, inputs are expected time-major, otherwise they are expected batch-major.
- **reverse** – If True, inputs are scanned in the reversed order. Equivalent to reversing the time dimension in both inputs and outputs. See https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.scan.html for more details.

Returns

- **output_sequence** - An arbitrarily nested structure of tensors of shape `[T, ...]` if time-major, otherwise `[B, T, ...]`.
- **final_state** - Core state at time step `T`.

Return type A tuple with two elements

static_unroll

haiku.**static_unroll**(*core*, *input_sequence*, *initial_state*, *time_major=True*)

Performs a static unroll of an RNN.

An *unroll* corresponds to calling the core on each element of the input sequence in a loop, carrying the state through:

```
state = initial_state
for t in range(len(input_sequence)):
   outputs, state = core(input_sequence[t], state)
```

A *static* unroll replaces a loop with its body repeated multiple times when executed inside `jax.jit()`:

```
state = initial_state
outputs0, state = core(input_sequence[0], state)
outputs1, state = core(input_sequence[1], state)
outputs2, state = core(input_sequence[2], state)
...
```

See *dynamic_unroll()* for a loop-preserving unroll function.

Parameters

- **core** – An *RNNCore* to unroll.
- **input_sequence** – An arbitrarily nested structure of tensors of shape `[T, ...]` if time-major=True, or `[B, T, ...]` if time_major=False, where `T` is the number of time steps.
- **initial_state** – An initial state of the given core.
- **time_major** – If True, inputs are expected time-major, otherwise they are expected batch-major.

Returns

- **output_sequence** - An arbitrarily nested structure of tensors of shape `[T, ...]` if time-major, otherwise `[B, T, ...]`.
- **final_state** - Core state at time step `T`.

Return type A tuple with two elements

expand_apply

haiku.**expand_apply**(*f*, *axis=0*)

Wraps f to temporarily add a size-1 axis to its inputs.

Syntactic sugar for:

```
ins = jax.tree_util.tree_map(lambda t: np.expand_dims(t, axis=axis), ins)
out = f(ins)
out = jax.tree_util.tree_map(lambda t: np.squeeze(t, axis=axis), out)
```

This may be useful for applying a function built for `[Time, Batch, ...]` arrays to a single timestep.

Parameters

- **f** – The callable to be applied to the expanded inputs.
- **axis** – Where to add the extra axis.

Returns f, wrapped as described above.

VanillaRNN

class haiku.**VanillaRNN**(*hidden_size*, *double_bias=True*, *name=None*)

Basic fully-connected RNN core.

Given x_t and the previous hidden state h_{t-1} the core computes

$$h_t = \mathrm{ReLU}(w_i x_t + b_i + w_h h_{t-1} + b_h)$$

The output is equal to the new state, h_t.

__init__(*hidden_size*, *double_bias=True*, *name=None*)

Constructs a vanilla RNN core.

Parameters

- **hidden_size** (*int*) – Hidden layer size.

- **double_bias** (*bool*) – Whether to use a bias in the two linear layers. This changes nothing to the learning performance of the cell. However, doubling will create two sets of bias parameters rather than one.
- **name** (*Optional[str]*) – Name of the module.

__call__(*inputs*, *prev_state*)
Run one step of the RNN.

Parameters

- **inputs** – An arbitrarily nested structure.
- **prev_state** – Previous core state.

Returns A tuple with two elements `output, next_state`. `output` is an arbitrarily nested structure. `next_state` is the next core state, this must be the same shape as `prev_state`.

initial_state(*batch_size*)
Constructs an initial state for this core.

Parameters **batch_size** (*Optional[int]*) – Optional int or an integral scalar tensor representing batch size. If None, the core may either fail or (experimentally) return an initial state without a batch dimension.

Returns Arbitrarily nested initial state for this core.

LSTM

class haiku.**LSTM**(*hidden_size*, *name=None*)
Long short-term memory (LSTM) RNN core.

The implementation is based on [1]. Given x_t and the previous state (h_{t-1}, c_{t-1}) the core computes

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i) \\
f_t &= \sigma(W_{if}x_t + W_{hf}h_{t-1} + b_f) \\
g_t &= \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g) \\
o_t &= \sigma(W_{io}x_t + W_{ho}h_{t-1} + b_o) \\
c_t &= f_t c_{t-1} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

where i_t, f_t, o_t are input, forget and output gate activations, and g_t is a vector of cell updates.

The output is equal to the new hidden, h_t.

Notes

Forget gate initialization: Following [2] we add 1.0 to b_f after initialization in order to reduce the scale of forgetting in the beginning of the training.

__init__(*hidden_size*, *name=None*)
Constructs an LSTM.

Parameters

- **hidden_size** (*int*) – Hidden layer size.
- **name** (*Optional[str]*) – Name of the module.

__call__(*inputs*, *prev_state*)
Run one step of the RNN.

Parameters

- **inputs** (*jnp.ndarray*) – An arbitrarily nested structure.
- **prev_state** (LSTMState) – Previous core state.

Return type Tuple[jnp.ndarray, *LSTMState*]

Returns A tuple with two elements `output, next_state`. `output` is an arbitrarily nested structure. `next_state` is the next core state, this must be the same shape as `prev_state`.

initial_state(*batch_size*)
Constructs an initial state for this core.

Parameters **batch_size** (*Optional[int]*) – Optional int or an integral scalar tensor representing batch size. If None, the core may either fail or (experimentally) return an initial state without a batch dimension.

Return type *LSTMState*

Returns Arbitrarily nested initial state for this core.

GRU

class haiku.**GRU**(*hidden_size*, *w_i_init=None*, *w_h_init=None*, *b_init=None*, *name=None*)
Gated Recurrent Unit.

The implementation is based on: https://arxiv.org/pdf/1412.3555v1.pdf with biases.

Given x_t and the previous state h_{t-1} the core computes

$$\begin{aligned} z_t &= \sigma(W_{iz}x_t + W_{hz}h_{t-1} + b_z) \\ r_t &= \sigma(W_{ir}x_t + W_{hr}h_{t-1} + b_r) \\ a_t &= \tanh(W_{ia}x_t + W_{ha}(r_t \bigodot h_{t-1}) + b_a) \\ h_t &= (1 - z_t) \bigodot h_{t-1} + z_t \bigodot a_t \end{aligned}$$

where z_t and r_t are reset and update gates.

The output is equal to the new hidden state, h_t.

__init__(*hidden_size*, *w_i_init=None*, *w_h_init=None*, *b_init=None*, *name=None*)
Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*Optional[str]*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

__call__(*inputs*, *state*)
Run one step of the RNN.

Parameters

- **inputs** – An arbitrarily nested structure.
- **prev_state** – Previous core state.

Returns A tuple with two elements `output, next_state`. `output` is an arbitrarily nested structure. `next_state` is the next core state, this must be the same shape as `prev_state`.

initial_state(*batch_size*)
Constructs an initial state for this core.

Parameters **batch_size** (*Optional[int]*) – Optional int or an integral scalar tensor representing batch size. If None, the core may either fail or (experimentally) return an initial state without a batch dimension.

Returns Arbitrarily nested initial state for this core.

DeepRNN

class haiku.**DeepRNN**(*layers*, *name=None*)
Wraps a sequence of cores and callables as a single core.

```
>>> deep_rnn = hk.DeepRNN([
...     hk.LSTM(hidden_size=4),
...     jax.nn.relu,
...     hk.LSTM(hidden_size=2),
... ])
```

The state of a *DeepRNN* is a tuple with one element per *RNNCore*. If no layers are *RNNCore*s, the state is an empty tuple.

__init__(*layers*, *name=None*)
Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*Optional[str]*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

haiku.**deep_rnn_with_skip_connections**(*layers*, *name=None*)
Constructs a *DeepRNN* with skip connections.

Skip connections alter the dependency structure within a *DeepRNN*. Specifically, input to the i-th layer (i > 0) is given by a concatenation of the core's inputs and the outputs of the (i-1)-th layer.

The output of the *DeepRNN* is the concatenation of the outputs of all cores.

```
outputs0, ... = layers[0](inputs, ...)
outputs1, ... = layers[1](tf.concat([inputs, outputs0], axis=-1], ...)
outputs2, ... = layers[2](tf.concat([inputs, outputs1], axis=-1], ...)
...
```

Parameters

- **layers** (*Sequence[RNNCore]*) – List of *RNNCore*s.
- **name** (*Optional[str]*) – Name of the module.

Return type *RNNCore*

Returns A _DeepRNN with skip connections.

Raises **ValueError** – If any of the layers is not an *RNNCore*.

ResetCore

class haiku.**ResetCore**(*core*, *name=None*)
A wrapper for managing state resets during unrolls.

When unrolling an *RNNCore* on a batch of inputs sequences it may be necessary to reset the core's state at different timesteps for different elements of the batch. The *ResetCore* class enables this by taking a batch of `should_reset` booleans in addition to the batch of inputs, and conditionally resetting the core's state for individual elements of the batch. You may also reset individual entries of the state by passing a `should_reset` nest compatible with the state structure.

__init__(*core*, *name=None*)
Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*Optional[str]*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

__call__(*inputs*, *state*)
Run one step of the wrapped core, handling state reset.

Parameters

- **inputs** – Tuple with two elements, `inputs, should_reset`, where `should_reset` is the signal used to reset the wrapped core's state. `should_reset` can be either tensor or nest. If nest, `should_reset` must match the state structure, and its components' shapes must be prefixes of the corresponding entries tensors' shapes in the state nest. If tensor, supported shapes are all commom shape prefixes of the state component tensors, e.g. `[batch_size]`.
- **state** – Previous wrapped core state.

Returns Tuple of the wrapped core's `output, next_state`.

initial_state(*batch_size*)
Constructs an initial state for this core.

Parameters **batch_size** (*Optional[int]*) – Optional int or an integral scalar tensor representing batch size. If None, the core may either fail or (experimentally) return an initial state without a batch dimension.

Returns Arbitrarily nested initial state for this core.

IdentityCore

class haiku.**IdentityCore**(*name=None*)
A recurrent core that forwards the inputs and an empty state.

This is commonly used when switching between recurrent and feedforward versions of a model while preserving the same interface.

__call__(*inputs*, *state*)
Run one step of the RNN.

Parameters

- **inputs** – An arbitrarily nested structure.
- **prev_state** – Previous core state.

Returns A tuple with two elements `output, next_state`. `output` is an arbitrarily nested structure. `next_state` is the next core state, this must be the same shape as `prev_state`.

initial_state(*batch_size*)
Constructs an initial state for this core.

Parameters **batch_size** (*Optional[int]*) – Optional int or an integral scalar tensor representing batch size. If None, the core may either fail or (experimentally) return an initial state without a batch dimension.

Returns Arbitrarily nested initial state for this core.

Conv1DLSTM

class haiku.**Conv1DLSTM**(*input_shape*, *output_channels*, *kernel_shape*, *name=None*)
1-D convolutional LSTM.

The implementation is based on [3]. Given x_t and the previous state (h_{t-1}, c_{t-1}) the core computes

$$
\begin{aligned}
i_t &= \sigma(W_{ii} * x_t + W_{hi} * h_{t-1} + b_i) \\
f_t &= \sigma(W_{if} * x_t + W_{hf} * h_{t-1} + b_f) \\
g_t &= \tanh(W_{ig} * x_t + W_{hg} * h_{t-1} + b_g) \\
o_t &= \sigma(W_{io} * x_t + W_{ho} * h_{t-1} + b_o) \\
c_t &= f_t c_{t-1} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

where $*$ denotes the convolution operator; i_t, f_t, o_t are input, forget and output gate activations, and g_t is a vector of cell updates.

The output is equal to the new hidden state, h_t.

Notes

Forget gate initialization: Following [2] we add 1.0 to b_f after initialization in order to reduce the scale of forgetting in the beginning of the training.

__init__(*input_shape*, *output_channels*, *kernel_shape*, *name=None*)
Constructs a 1-D convolutional LSTM.

Parameters

- **input_shape** (*Sequence[int]*) – Shape of the inputs excluding batch size.
- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – Sequence of kernel sizes (of length 1), or an int. `kernel_shape` will be expanded to define a kernel size in all dimensions.
- **name** (*Optional[str]*) – Name of the module.

Conv2DLSTM

class haiku.**Conv2DLSTM**(*input_shape*, *output_channels*, *kernel_shape*, *name=None*)
2-D convolutional LSTM.

The implementation is based on [3]. Given x_t and the previous state (h_{t-1}, c_{t-1}) the core computes

$$
\begin{aligned}
i_t &= \sigma(W_{ii} * x_t + W_{hi} * h_{t-1} + b_i) \\
f_t &= \sigma(W_{if} * x_t + W_{hf} * h_{t-1} + b_f) \\
g_t &= \tanh(W_{ig} * x_t + W_{hg} * h_{t-1} + b_g) \\
o_t &= \sigma(W_{io} * x_t + W_{ho} * h_{t-1} + b_o) \\
c_t &= f_t c_{t-1} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

where $*$ denotes the convolution operator; i_t, f_t, o_t are input, forget and output gate activations, and g_t is a vector of cell updates.

The output is equal to the new hidden state, h_t.

Notes

Forget gate initialization: Following [2] we add 1.0 to b_f after initialization in order to reduce the scale of forgetting in the beginning of the training.

__init__(*input_shape*, *output_channels*, *kernel_shape*, *name=None*)
Constructs a 2-D convolutional LSTM.

Parameters

- **input_shape** (*Sequence[int]*) – Shape of the inputs excluding batch size.
- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – Sequence of kernel sizes (of length 2), or an int. `kernel_shape` will be expanded to define a kernel size in all dimensions.
- **name** (*Optional[str]*) – Name of the module.

Conv3DLSTM

class haiku.**Conv3DLSTM**(*input_shape*, *output_channels*, *kernel_shape*, *name=None*)
3-D convolutional LSTM.

The implementation is based on [3]. Given x_t and the previous state (h_{t-1}, c_{t-1}) the core computes

$$
\begin{aligned}
i_t &= \sigma(W_{ii} * x_t + W_{hi} * h_{t-1} + b_i) \\
f_t &= \sigma(W_{if} * x_t + W_{hf} * h_{t-1} + b_f) \\
g_t &= \tanh(W_{ig} * x_t + W_{hg} * h_{t-1} + b_g) \\
o_t &= \sigma(W_{io} * x_t + W_{ho} * h_{t-1} + b_o) \\
c_t &= f_t c_{t-1} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

where $*$ denotes the convolution operator; i_t, f_t, o_t are input, forget and output gate activations, and g_t is a vector of cell updates.

The output is equal to the new hidden state, h_t.

Notes

Forget gate initialization: Following [2] we add 1.0 to b_f after initialization in order to reduce the scale of forgetting in the beginning of the training.

__init__(*input_shape*, *output_channels*, *kernel_shape*, *name=None*)
Constructs a 3-D convolutional LSTM.

Parameters

- **input_shape** (*Sequence[int]*) – Shape of the inputs excluding batch size.
- **output_channels** (*int*) – Number of output channels.
- **kernel_shape** (*Union[int, Sequence[int]]*) – Sequence of kernel sizes (of length 3), or an int. `kernel_shape` will be expanded to define a kernel size in all dimensions.
- **name** (*Optional[str]*) – Name of the module.

1.4.8 Attention

MultiHeadAttention

class haiku.**MultiHeadAttention**(*num_heads*, *key_size*, *w_init_scale*, *value_size=None*, *model_size=None*, *name=None*)
Multi-headed attention mechanism.

As described in the vanilla Transformer paper: "Attention is all you need" https://arxiv.org/abs/1706.03762

__init__(*num_heads*, *key_size*, *w_init_scale*, *value_size=None*, *model_size=None*, *name=None*)
Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*Optional[str]*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

__call__(*query*, *key*, *value*, *mask=None*)
Compute (optionally masked) MHA with queries, keys & values.

Return type jnp.ndarray

1.4.9 Batch

Reshape(output_shape[, preserve_dims, name])	Reshapes input Tensor, preserving the batch dimension.
Flatten([preserve_dims, name])	Flattens the input, preserving the batch dimension(s).
BatchApply(f[, num_dims])	Temporarily merges leading dimensions of input tensors.

Reshape

class haiku.**Reshape**(*output_shape*, *preserve_dims=1*, *name=None*)

Reshapes input Tensor, preserving the batch dimension.

For example, given an input tensor with shape `[B, H, W, C, D]`:

```
>>> B, H, W, C, D = range(1, 6)
>>> x = jnp.ones([B, H, W, C, D])
```

The default behavior when `output_shape` is `(-1, D)` is to flatten all dimensions between `B` and `D`:

```
>>> mod = hk.Reshape(output_shape=(-1, D))
>>> assert mod(x).shape == (B, H*W*C, D)
```

You can change the number of preserved leading dimensions via `preserve_dims`:

```
>>> mod = hk.Reshape(output_shape=(-1, D), preserve_dims=2)
>>> assert mod(x).shape == (B, H, W*C, D)

>>> mod = hk.Reshape(output_shape=(-1, D), preserve_dims=3)
>>> assert mod(x).shape == (B, H, W, C, D)

>>> mod = hk.Reshape(output_shape=(-1, D), preserve_dims=4)
>>> assert mod(x).shape == (B, H, W, C, 1, D)
```

Alternatively, a negative value of `preserve_dims` specifies the number of trailing dimensions to replace with `output_shape`:

```
>>> mod = hk.Reshape(output_shape=(-1, D), preserve_dims=-3)
>>> assert mod(x).shape == (B, H, W*C, D)
```

This is useful in the case of applying the same module to batched and unbatched outputs:

```
>>> mod = hk.Reshape(output_shape=(-1, D), preserve_dims=-3)
>>> assert mod(x[0]).shape == (H, W*C, D)
```

__init__(*output_shape*, *preserve_dims=1*, *name=None*)

Constructs a *Reshape* module.

Parameters

- **output_shape** (*Sequence[int]*) – Shape to reshape the input tensor to while preserving its first `preserve_dims` dimensions. When the special value `-1` appears in `output_shape` the corresponding size is automatically inferred. Note that `-1` can only appear once in `output_shape`. To flatten all non-batch dimensions use *Flatten*.
- **preserve_dims** (*int*) – Number of leading dimensions that will not be reshaped. If negative, this is interpreted instead as the number of trailing dimensions to replace with the new shape.
- **name** (*Optional[str]*) – Name of the module.

Raises **ValueError** – If `preserve_dims` is zero.

__call__(*inputs*)

Call self as a function.

Flatten

class haiku.**Flatten**(*preserve_dims=1*, *name=None*)

Flattens the input, preserving the batch dimension(s).

By default, *Flatten* combines all dimensions except the first. Additional leading dimensions can be preserved by setting `preserve_dims`.

```
>>> x = jnp.ones([3, 2, 4])
>>> flat = hk.Flatten()
>>> flat(x).shape
(3, 8)
```

When the input to flatten has fewer than `preserve_dims` dimensions it is returned unchanged:

```
>>> x = jnp.ones([3])
>>> flat(x).shape
(3,)
```

Alternatively, a negative value of *preserve_dims* specifies the number of trailing dimensions flattened:

```
>>> x = jnp.ones([3, 2, 4])
>>> negative_flat = hk.Flatten(preserve_dims=-2)
>>> negative_flat(x).shape
(3, 8)
```

This allows the same module to be seamlessly applied to a single element or a batch of elements with the same element shape:

>> negative_flat(x[0]).shape (8,)

__init__(*preserve_dims=1*, *name=None*)

Constructs a *Reshape* module.

Parameters

- **output_shape** – Shape to reshape the input tensor to while preserving its first `preserve_dims` dimensions. When the special value `-1` appears in `output_shape` the corresponding size is automatically inferred. Note that `-1` can only appear once in `output_shape`. To flatten all non-batch dimensions use *Flatten*.
- **preserve_dims** (*int*) – Number of leading dimensions that will not be reshaped. If negative, this is interpreted instead as the number of trailing dimensions to replace with the new shape.
- **name** (*Optional[str]*) – Name of the module.

Raises **ValueError** – If `preserve_dims` is zero.

BatchApply

class haiku.**BatchApply**(*f*, *num_dims=2*)
Temporarily merges leading dimensions of input tensors.

Merges the leading dimensions of a tensor into a single dimension, runs the given callable, then splits the leading dimension of the result to match the input.

Input arrays whose rank is smaller than the number of dimensions to collapse are passed unmodified.

This may be useful for applying a module to each timestep of e.g. a `[Time, Batch, ...]` array.

For some `fs` and platforms, this may be more efficient than `jax.vmap()`, especially when combined with other transformations like `jax.grad()`.

__init__(*f*, *num_dims=2*)
Constructs a *BatchApply* module.

Parameters

- **f** – The callable to be applied to the reshaped array.
- **num_dims** – The number of dimensions to merge.

__call__(**args*, ***kwargs*)
Call self as a function.

1.4.10 Embedding

Embed([vocab_size, embed_dim, ...])	Module for embedding tokens in a low-dimensional space.
EmbedLookupStyle(value)	How to return the embedding matrices given IDs.

Embed

class haiku.**Embed**(*vocab_size=None*, *embed_dim=None*, *embedding_matrix=None*, *w_init=None*, *lookup_style='ARRAY_INDEX'*, *name=None*)
Module for embedding tokens in a low-dimensional space.

__init__(*vocab_size=None*, *embed_dim=None*, *embedding_matrix=None*, *w_init=None*, *lookup_style='ARRAY_INDEX'*, *name=None*)
Constructs an Embed module.

Parameters

- **vocab_size** (*Optional[int]*) – The number of unique tokens to embed. If not provided, an existing vocabulary matrix from which `vocab_size` can be inferred must be provided as `embedding_matrix`.
- **embed_dim** (*Optional[int]*) – Number of dimensions to assign to each embedding. If an existing vocabulary matrix initializes the module, this should not be provided as it will be inferred.
- **embedding_matrix** (*Optional[jnp.ndarray]*) – A matrix-like object equivalent in size to `[vocab_size, embed_dim]`. If given, it is used as the initial value for the embedding matrix and neither `vocab_size` or `embed_dim` need be given. If they are given, their values are checked to be consistent with the dimensions of `embedding_matrix`.

- **w_init** (*Optional[hk.initializers.Initializer]*) – An initializer for the embeddings matrix. As a default, embeddings are initialized via a truncated normal distribution.
- **lookup_style** (*Union[str, hk.EmbedLookupStyle]*) – One of the enum values of *EmbedLookupStyle* determining how to access the value of the embeddings given an ID. Regardless the input should be a dense array of integer values representing ids. This setting changes how internally this module maps those ids to embeddings. The result is the same, but the speed and memory tradeoffs are different. It defaults to using NumPy-style array indexing. This value is only the default for the module, and at any given invocation can be overridden in *__call__()*.
- **name** (*Optional[str]*) – Optional name for this module.

Raises **ValueError** – If none of `embed_dim`, `embedding_matrix` and `vocab_size` are supplied, or if `embedding_matrix` is supplied and `embed_dim` or `vocab_size` is not consistent with the supplied matrix.

__call__(*ids*, *lookup_style=None*)

Lookup embeddings.

Looks up an embedding vector for each value in `ids`. All ids must be within `[0, vocab_size)` to prevent `NaN`s from propagating.

Parameters

- **ids** (*jnp.ndarray*) – integer array.
- **lookup_style** (*Optional[Union[str, hk.EmbedLookupStyle]]*) – Overrides the `lookup_style` given in the constructor.

Return type jnp.ndarray

Returns Tensor of `ids.shape + [embedding_dim]`.

Raises

- **AttributeError** – If `lookup_style` is not valid.
- **ValueError** – If `ids` is not an integer array.

EmbedLookupStyle

class haiku.**EmbedLookupStyle**(*value*)

How to return the embedding matrices given IDs.

ARRAY_INDEX = 1

ONE_HOT = 2

1.4.11 Initializers

Initializer	The central part of internal API.
Constant(constant)	Initializes with a constant.
Identity([gain])	Initializer that generates the identity matrix.
Orthogonal([scale, axis])	Uniform scaling initializer.
RandomNormal([stddev, mean])	Initializes by sampling from a normal distribution.
RandomUniform([minval, maxval])	Initializes by sampling from a uniform distribution.

continues on next page

Table 15 – continued from previous page

TruncatedNormal([stddev, mean])	Initializes by sampling from a truncated normal distribution.
VarianceScaling([scale, mode, distribution])	Initializer which adapts its scale to the shape of the initialized array.
UniformScaling([scale])	Uniform scaling initializer.

Initializer

haiku.initializers.**Initializer**

Constant

class haiku.initializers.**Constant**(*constant*)

Initializes with a constant.

__init__(*constant*)

Constructs a Constant initializer.

Parameters **constant** (*Union[float, int, jnp.ndarray]*) – Constant to initialize with.

__call__(*shape*, *dtype*)

Call self as a function.

Return type jnp.ndarray

Identity

class haiku.initializers.**Identity**(*gain=1.0*)

Initializer that generates the identity matrix.

Constructs a 2D identity matrix or batches of these.

__init__(*gain=1.0*)

Constructs an *Identity* initializer.

Parameters **gain** (*Union[float, jnp.ndarray]*) – Multiplicative factor to apply to the identity matrix.

__call__(*shape*, *dtype*)

Call self as a function.

Return type jnp.ndarray

Orthogonal

class haiku.initializers.**Orthogonal**(*scale=1.0*, *axis=- 1*)

Uniform scaling initializer.

__init__(*scale=1.0*, *axis=- 1*)

Construct an initializer for uniformly distributed orthogonal matrices.

These matrices will be row-orthonormal along the access specified by `axis`. If the rank of the weight is greater than 2, the shape will be flattened in all other dimensions and then will be row-orthonormal along

the final dimension. Note that this only works if the `axis` dimension is larger, otherwise the matrix will be transposed (equivalently, it will be column orthonormal instead of row orthonormal).

If the shape is not square, the matrices will have orthonormal rows or columns depending on which side is smaller.

Parameters

- **scale** – Scale factor.
- **axis** – Which axis corresponds to the "output dimension" of the tensor.

Returns An orthogonally initialized parameter.

__call__(*shape*, *dtype*)
Call self as a function.

Return type jnp.ndarray

RandomNormal

class haiku.initializers.**RandomNormal**(*stddev=1.0*, *mean=0.0*)
Initializes by sampling from a normal distribution.

__init__(*stddev=1.0*, *mean=0.0*)
Constructs a *RandomNormal* initializer.

Parameters

- **stddev** – The standard deviation of the normal distribution to sample from.
- **mean** – The mean of the normal distribution to sample from.

__call__(*shape*, *dtype*)
Call self as a function.

Return type jnp.ndarray

RandomUniform

class haiku.initializers.**RandomUniform**(*minval=0.0*, *maxval=1.0*)
Initializes by sampling from a uniform distribution.

__init__(*minval=0.0*, *maxval=1.0*)
Constructs a *RandomUniform* initializer.

Parameters

- **minval** – The lower limit of the uniform distribution.
- **maxval** – The upper limit of the uniform distribution.

__call__(*shape*, *dtype*)
Call self as a function.

Return type jnp.ndarray

TruncatedNormal

class haiku.initializers.**TruncatedNormal**(*stddev=1.0*, *mean=0.0*)
Initializes by sampling from a truncated normal distribution.

__init__(*stddev=1.0*, *mean=0.0*)
Constructs a *TruncatedNormal* initializer.

Parameters

- **stddev** (*Union[float, jnp.ndarray]*) – The standard deviation parameter of the truncated normal distribution.
- **mean** (*Union[float, jnp.ndarray]*) – The mean of the truncated normal distribution.

__call__(*shape*, *dtype*)
Call self as a function.

Return type jnp.ndarray

VarianceScaling

class haiku.initializers.**VarianceScaling**(*scale=1.0*, *mode='fan_in'*, *distribution='truncated_normal'*)
Initializer which adapts its scale to the shape of the initialized array.

The initializer first computes the scaling factor `s = scale / n`, where n is:

- Number of input units in the weight tensor, if `mode = fan_in`.
- Number of output units, if `mode = fan_out`.
- Average of the numbers of input and output units, if `mode = fan_avg`.

Then, with `distribution="truncated_normal"` or `"normal"`, samples are drawn from a distribution with a mean of zero and a standard deviation (after truncation, if used) `stddev = sqrt(s)`.

With `distribution=uniform`, samples are drawn from a uniform distribution within `[-limit, limit]`, with `limit = sqrt(3 * s)`.

The variance scaling initializer can be configured to generate other standard initializers using the scale, mode and distribution arguments. Here are some example configurations:

Name	Parameters
glorot_uniform	VarianceScaling(1.0, “fan_avg”, “uniform”)
glorot_normal	VarianceScaling(1.0, “fan_avg”, “truncated_normal”)
lecun_uniform	VarianceScaling(1.0, “fan_in”, “uniform”)
lecun_normal	VarianceScaling(1.0, “fan_in”, “truncated_normal”)
he_uniform	VarianceScaling(2.0, “fan_in”, “uniform”)
he_normal	VarianceScaling(2.0, “fan_in”, “truncated_normal”)

__init__(*scale=1.0*, *mode='fan_in'*, *distribution='truncated_normal'*)
Constructs the *VarianceScaling* initializer.

Parameters

- **scale** – Scale to multiply the variance by.
- **mode** – One of `fan_in`, `fan_out`, `fan_avg`

- **distribution** – Random distribution to use. One of `truncated_normal`, `normal` or `uniform`.

__call__(*shape*, *dtype*)
Call self as a function.

Return type jnp.ndarray

UniformScaling

class haiku.initializers.**UniformScaling**(*scale=1.0*)
Uniform scaling initializer.

Initializes by sampling from a uniform distribution, but with the variance scaled by the inverse square root of the number of input units, multiplied by the scale.

__init__(*scale=1.0*)
Constructs the *UniformScaling* initializer.

Parameters **scale** – Scale to multiply the upper limit of the uniform distribution by.

__call__(*shape*, *dtype*)
Call self as a function.

Return type jnp.ndarray

1.4.12 Paddings

PadFn	The central part of internal API.
is_padfn(padding)	Tests whether the given argument is a single or sequence of PadFns.
create(padding, kernel, rate, n)	Generates the padding required for a given padding algorithm.
create_from_padfn(padding, kernel, rate, n)	Generates the padding required for a given padding algorithm.
create_from_tuple(padding, n)	Create a padding tuple using partially specified padding tuple.
causal(effective_kernel_size)	Pre-padding such that output has no dependence on the future.
full(effective_kernel_size)	Maximal padding whilst not convolving over just padded elements.
reverse_causal(effective_kernel_size)	Post-padding such that output has no dependence on the past.
same(effective_kernel_size)	Pads such that the output size matches input size for stride=1.
valid(effective_kernel_size)	No padding.

PadFn

haiku.pad.**PadFn**

is_padfn

haiku.pad.**is_padfn**(*padding*)

Tests whether the given argument is a single or sequence of PadFns.

Return type bool

create

haiku.pad.**create**(*padding*, *kernel*, *rate*, *n*)

Generates the padding required for a given padding algorithm.

Parameters

- **padding** (*Union[hk.pad.PadFn, Sequence[hk.pad.PadFn]]*) – callable/tuple or a sequence of callables/tuples. The callables take an integer representing the effective kernel size (kernel size when the rate is 1) and return a sequence of two integers representing the padding before and padding after for that dimension. The tuples are defined with two elements, padding before and after. If *padding* is a sequence it must be of length 1 or *n*.
- **kernel** (*Union[int, Sequence[int]]*) – int or sequence of ints of length n. The size of the kernel for each dimension. If it is an int it will be replicated for the non channel and batch dimensions.
- **rate** (*Union[int, Sequence[int]]*) – int or sequence of ints of length n. The dilation rate for each dimension. If it is an int it will be replicated for the non channel and batch dimensions.
- **n** (*int*) – the number of spatial dimensions.

Return type Sequence[Tuple[int, int]]

Returns A sequence of length n containing the padding for each element. These are of the form `[pad_before, pad_after]`.

create_from_padfn

haiku.pad.**create_from_padfn**(*padding*, *kernel*, *rate*, *n*)

Generates the padding required for a given padding algorithm.

Parameters

- **padding** (*Union[hk.pad.PadFn, Sequence[hk.pad.PadFn]]*) – callable/tuple or a sequence of callables/tuples. The callables take an integer representing the effective kernel size (kernel size when the rate is 1) and return a sequence of two integers representing the padding before and padding after for that dimension. The tuples are defined with two elements, padding before and after. If *padding* is a sequence it must be of length 1 or *n*.
- **kernel** (*Union[int, Sequence[int]]*) – int or sequence of ints of length n. The size of the kernel for each dimension. If it is an int it will be replicated for the non channel and batch dimensions.

- **rate** (*Union[int, Sequence[int]]*) – int or sequence of ints of length `n`. The dilation rate for each dimension. If it is an int it will be replicated for the non channel and batch dimensions.
- **n** (*int*) – the number of spatial dimensions.

Return type Sequence[Tuple[int, int]]

Returns A sequence of length n containing the padding for each element. These are of the form `[pad_before, pad_after]`.

create_from_tuple

`haiku.pad.`**`create_from_tuple`**(*padding*, *n*)

Create a padding tuple using partially specified padding tuple.

Return type Sequence[Tuple[int, int]]

causal

`haiku.pad.`**`causal`**(*effective_kernel_size*)

Pre-padding such that output has no dependence on the future.

Return type Tuple[int, int]

full

`haiku.pad.`**`full`**(*effective_kernel_size*)

Maximal padding whilst not convolving over just padded elements.

Return type Tuple[int, int]

reverse_causal

`haiku.pad.`**`reverse_causal`**(*effective_kernel_size*)

Post-padding such that output has no dependence on the past.

Return type Tuple[int, int]

same

`haiku.pad.`**`same`**(*effective_kernel_size*)

Pads such that the output size matches input size for stride=1.

Return type Tuple[int, int]

valid

haiku.pad.**valid**(*effective_kernel_size*)
No padding.

Return type Tuple[int, int]

1.5 Full Networks

1.5.1 MLP

class haiku.nets.**MLP**(*output_sizes*, *w_init=None*, *b_init=None*, *with_bias=True*, *activation=<jax._src.custom_derivatives.custom_jvp object>*, *activate_final=False*, *name=None*)
A multi-layer perceptron module.

__init__(*output_sizes*, *w_init=None*, *b_init=None*, *with_bias=True*, *activation=<jax._src.custom_derivatives.custom_jvp object>*, *activate_final=False*, *name=None*)
Constructs an MLP.

Parameters

- **output_sizes** (*Iterable[int]*) – Sequence of layer sizes.
- **w_init** (*Optional[hk.initializers.Initializer]*) – Initializer for *Linear* weights.
- **b_init** (*Optional[hk.initializers.Initializer]*) – Initializer for *Linear* bias. Must be `None` if `with_bias=False`.
- **with_bias** (*bool*) – Whether or not to apply a bias in each layer.
- **activation** (*Callable[[jnp.ndarray], jnp.ndarray]*) – Activation function to apply between *Linear* layers. Defaults to ReLU.
- **activate_final** (*bool*) – Whether or not to activate the final layer of the MLP.
- **name** (*Optional[str]*) – Optional name for this module.

Raises **ValueError** – If `with_bias` is `False` and `b_init` is not `None`.

__call__(*inputs*, *dropout_rate=None*, *rng=None*)
Connects the module to some inputs.

Parameters

- **inputs** (*jnp.ndarray*) – A Tensor of shape `[batch_size, input_size]`.
- **dropout_rate** (*Optional[float]*) – Optional dropout rate.
- **rng** – Optional RNG key. Require when using dropout.

Return type jnp.ndarray

Returns The output of the model of size `[batch_size, output_size]`.

reverse(*activate_final=None*, *name=None*)
Returns a new MLP which is the layer-wise reverse of this MLP.

NOTE: Since computing the reverse of an MLP requires knowing the input size of each linear layer this method will fail if the module has not been called at least once.

The contract of reverse is that the reversed module will accept the output of the parent module as input and produce an output which is the input size of the parent.

```
>>> mlp = hk.nets.MLP([1, 2, 3])
>>> y = mlp(jnp.ones([1, 2]))
>>> rev = mlp.reverse()
>>> rev(y)
DeviceArray(...)
```

Parameters

- **activate_final** (*Optional[bool]*) – Whether the final layer of the MLP should be activated.
- **name** (*Optional[str]*) – Optional name for the new module. The default name will be the name of the current module prefixed with `"reversed_"`.

Return type 'MLP'

Returns An MLP instance which is the reverse of the current instance. Note these instances do not share weights and, apart from being symmetric to each other, are not coupled in any way.

1.5.2 MobileNet

MobileNetV1

class haiku.nets.**MobileNetV1**(*strides=(1, 2, 1, 2, 1, 2, 1, 1, 1, 1, 1, 2, 1)*, *channels=(64, 128, 128, 256, 256, 512, 512, 512, 512, 512, 512, 1024, 1024)*, *num_classes=1000*, *use_bn=True*, *name=None*)

MobileNetV1 model.

__init__(*strides=(1, 2, 1, 2, 1, 2, 1, 1, 1, 1, 1, 2, 1)*, *channels=(64, 128, 128, 256, 256, 512, 512, 512, 512, 512, 512, 1024, 1024)*, *num_classes=1000*, *use_bn=True*, *name=None*)

Constructs a MobileNetV1 model.

Parameters

- **strides** (*Sequence[int]*) – The stride to use the in depthwise convolution in each mobilenet block.
- **channels** (*Sequence[int]*) – Number of output channels from the pointwise convolution to use in each block.
- **num_classes** (*int*) – Number of classes.
- **use_bn** (*bool*) – Whether or not to use batch normalization. Defaults to True. When true, biases are not used. When false, biases are used.
- **name** (*Optional[str]*) – Name of the module.

__call__(*inputs*, *is_training*)

Call self as a function.

Return type jnp.ndarray

1.5.3 ResNet

ResNet(blocks_per_group, num_classes[, ...])	ResNet model.
ResNet.BlockGroup(channels, num_blocks, ...)	Higher level block for ResNet implementation.
ResNet.BlockV1(channels, stride, ...[, name])	ResNet V1 block with optional bottleneck.
ResNet.BlockV2(channels, stride, ...[, name])	ResNet V2 block with optional bottleneck.
ResNet18(num_classes[, bn_config, ...])	ResNet18.
ResNet34(num_classes[, bn_config, ...])	ResNet34.
ResNet50(num_classes[, bn_config, ...])	ResNet50.
ResNet101(num_classes[, bn_config, ...])	ResNet101.
ResNet152(num_classes[, bn_config, ...])	ResNet152.
ResNet200(num_classes[, bn_config, ...])	ResNet200.

ResNet

class haiku.nets.**ResNet**(*blocks_per_group*, *num_classes*, *bn_config=None*, *resnet_v2=False*, *bottleneck=True*, *channels_per_group=(256, 512, 1024, 2048)*, *use_projection=(True, True, True, True)*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

ResNet model.

class BlockGroup(*channels*, *num_blocks*, *stride*, *bn_config*, *resnet_v2*, *bottleneck*, *use_projection*, *name=None*)

Higher level block for ResNet implementation.

__call__(*inputs*, *is_training*, *test_local_stats*)

Call self as a function.

__init__(*channels*, *num_blocks*, *stride*, *bn_config*, *resnet_v2*, *bottleneck*, *use_projection*, *name=None*)

Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*Optional[str]*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

class BlockV1(*channels*, *stride*, *use_projection*, *bn_config*, *bottleneck*, *name=None*)

ResNet V1 block with optional bottleneck.

__call__(*inputs*, *is_training*, *test_local_stats*)

Call self as a function.

__init__(*channels*, *stride*, *use_projection*, *bn_config*, *bottleneck*, *name=None*)

Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*Optional[str]*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

class BlockV2(*channels*, *stride*, *use_projection*, *bn_config*, *bottleneck*, *name=None*)

ResNet V2 block with optional bottleneck.

__call__(*inputs*, *is_training*, *test_local_stats*)

Call self as a function.

__init__(*channels*, *stride*, *use_projection*, *bn_config*, *bottleneck*, *name=None*)
Initializes the current module with the given name.

Subclasses should call this constructor before creating other modules or variables such that those modules are named correctly.

Parameters **name** (*Optional[str]*) – An optional string name for the class. Must be a valid Python identifier. If `name` is not provided then the class name for the current instance is converted to `lower_snake_case` and used instead.

__init__(*blocks_per_group*, *num_classes*, *bn_config=None*, *resnet_v2=False*, *bottleneck=True*, *channels_per_group=(256, 512, 1024, 2048)*, *use_projection=(True, True, True, True)*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)
Constructs a ResNet model.

Parameters

- **blocks_per_group** (*Sequence[int]*) – A sequence of length 4 that indicates the number of blocks created in each group.
- **num_classes** (*int*) – The number of classes to classify the inputs into.
- **bn_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – A dictionary of two elements, `decay_rate` and `eps` to be passed on to the *BatchNorm* layers. By default the `decay_rate` is `0.9` and `eps` is `1e-5`.
- **resnet_v2** (*bool*) – Whether to use the v1 or v2 ResNet implementation. Defaults to `False`.
- **bottleneck** (*bool*) – Whether the block should bottleneck or not. Defaults to `True`.
- **channels_per_group** (*Sequence[int]*) – A sequence of length 4 that indicates the number of channels used for each block in each group.
- **use_projection** (*Sequence[bool]*) – A sequence of length 4 that indicates whether each residual block should use projection.
- **logits_config** (*Optional[Mapping[str, Any]]*) – A dictionary of keyword arguments for the logits layer.
- **name** (*Optional[str]*) – Name of the module.
- **initial_conv_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – Keyword arguments passed to the constructor of the initial *Conv2D* module.
- **strides** (*Sequence[int]*) – A sequence of length 4 that indicates the size of stride of convolutions for each block in each group.

__call__(*inputs*, *is_training*, *test_local_stats=False*)
Call self as a function.

ResNet18

class haiku.nets.**ResNet18**(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)
ResNet18.

__init__(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)
Constructs a ResNet model.

Parameters

- **num_classes** (*int*) – The number of classes to classify the inputs into.

- **bn_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – A dictionary of two elements, `decay_rate` and `eps` to be passed on to the *BatchNorm* layers.
- **resnet_v2** (*bool*) – Whether to use the v1 or v2 ResNet implementation. Defaults to `False`.
- **logits_config** (*Optional[Mapping[str, Any]]*) – A dictionary of keyword arguments for the logits layer.
- **name** (*Optional[str]*) – Name of the module.
- **initial_conv_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – Keyword arguments passed to the constructor of the initial *Conv2D* module.
- **strides** (*Sequence[int]*) – A sequence of length 4 that indicates the size of stride of convolutions for each block in each group.

ResNet34

class haiku.nets.**ResNet34**(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

ResNet34.

__init__(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

Constructs a ResNet model.

Parameters

- **num_classes** (*int*) – The number of classes to classify the inputs into.
- **bn_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – A dictionary of two elements, `decay_rate` and `eps` to be passed on to the *BatchNorm* layers.
- **resnet_v2** (*bool*) – Whether to use the v1 or v2 ResNet implementation. Defaults to `False`.
- **logits_config** (*Optional[Mapping[str, Any]]*) – A dictionary of keyword arguments for the logits layer.
- **name** (*Optional[str]*) – Name of the module.
- **initial_conv_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – Keyword arguments passed to the constructor of the initial *Conv2D* module.
- **strides** (*Sequence[int]*) – A sequence of length 4 that indicates the size of stride of convolutions for each block in each group.

ResNet50

class haiku.nets.**ResNet50**(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

ResNet50.

__init__(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

Constructs a ResNet model.

Parameters

- **num_classes** (*int*) – The number of classes to classify the inputs into.

- **bn_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – A dictionary of two elements, `decay_rate` and `eps` to be passed on to the *BatchNorm* layers.
- **resnet_v2** (*bool*) – Whether to use the v1 or v2 ResNet implementation. Defaults to `False`.
- **logits_config** (*Optional[Mapping[str, Any]]*) – A dictionary of keyword arguments for the logits layer.
- **name** (*Optional[str]*) – Name of the module.
- **initial_conv_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – Keyword arguments passed to the constructor of the initial *Conv2D* module.
- **strides** (*Sequence[int]*) – A sequence of length 4 that indicates the size of stride of convolutions for each block in each group.

ResNet101

class haiku.nets.**ResNet101**(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

ResNet101.

__init__(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

Constructs a ResNet model.

Parameters

- **num_classes** (*int*) – The number of classes to classify the inputs into.
- **bn_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – A dictionary of two elements, `decay_rate` and `eps` to be passed on to the *BatchNorm* layers.
- **resnet_v2** (*bool*) – Whether to use the v1 or v2 ResNet implementation. Defaults to `False`.
- **logits_config** (*Optional[Mapping[str, Any]]*) – A dictionary of keyword arguments for the logits layer.
- **name** (*Optional[str]*) – Name of the module.
- **initial_conv_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – Keyword arguments passed to the constructor of the initial *Conv2D* module.
- **strides** (*Sequence[int]*) – A sequence of length 4 that indicates the size of stride of convolutions for each block in each group.

ResNet152

class haiku.nets.**ResNet152**(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

ResNet152.

__init__(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

Constructs a ResNet model.

Parameters

- **num_classes** (*int*) – The number of classes to classify the inputs into.

- **bn_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – A dictionary of two elements, `decay_rate` and `eps` to be passed on to the *BatchNorm* layers.
- **resnet_v2** (*bool*) – Whether to use the v1 or v2 ResNet implementation. Defaults to `False`.
- **logits_config** (*Optional[Mapping[str, Any]]*) – A dictionary of keyword arguments for the logits layer.
- **name** (*Optional[str]*) – Name of the module.
- **initial_conv_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – Keyword arguments passed to the constructor of the initial *Conv2D* module.
- **strides** (*Sequence[int]*) – A sequence of length 4 that indicates the size of stride of convolutions for each block in each group.

ResNet200

class haiku.nets.**ResNet200**(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

ResNet200.

__init__(*num_classes*, *bn_config=None*, *resnet_v2=False*, *logits_config=None*, *name=None*, *initial_conv_config=None*, *strides=(1, 2, 2, 2)*)

Constructs a ResNet model.

Parameters

- **num_classes** (*int*) – The number of classes to classify the inputs into.
- **bn_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – A dictionary of two elements, `decay_rate` and `eps` to be passed on to the *BatchNorm* layers.
- **resnet_v2** (*bool*) – Whether to use the v1 or v2 ResNet implementation. Defaults to `False`.
- **logits_config** (*Optional[Mapping[str, Any]]*) – A dictionary of keyword arguments for the logits layer.
- **name** (*Optional[str]*) – Name of the module.
- **initial_conv_config** (*Optional[Mapping[str, FloatStrOrBool]]*) – Keyword arguments passed to the constructor of the initial *Conv2D* module.
- **strides** (*Sequence[int]*) – A sequence of length 4 that indicates the size of stride of convolutions for each block in each group.

1.5.4 VectorQuantizer

VectorQuantizer(embedding_dim, ...[, dtype, ...])	Haiku module representing the VQ-VAE layer.
VectorQuantizerEMA(embedding_dim, ...[, ...])	Haiku module representing the VQ-VAE layer.

VectorQuantizer

class haiku.nets.**VectorQuantizer**(*embedding_dim*, *num_embeddings*, *commitment_cost*, *dtype=<class 'jax._src.numpy.lax_numpy.float32'>*, *name=None*)

Haiku module representing the VQ-VAE layer.

Implements the algorithm presented in "Neural Discrete Representation Learning" by van den Oord et al. https://arxiv.org/abs/1711.00937

Input any tensor to be quantized. Last dimension will be used as space in which to quantize. All other dimensions will be flattened and will be seen as different examples to quantize.

The output tensor will have the same shape as the input.

For example a tensor with shape `[16, 32, 32, 64]` will be reshaped into `[16384, 64]` and all `16384` vectors (each of `64` dimensions) will be quantized independently.

embedding_dim
integer representing the dimensionality of the tensors in the quantized space. Inputs to the modules must be in this format as well.

num_embeddings
integer, the number of vectors in the quantized space.

commitment_cost
scalar which controls the weighting of the loss terms (see equation 4 in the paper - this variable is Beta).

__init__(*embedding_dim*, *num_embeddings*, *commitment_cost*, *dtype=<class 'jax._src.numpy.lax_numpy.float32'>*, *name=None*)

Initializes a VQ-VAE module.

Parameters

- **embedding_dim** (*int*) – dimensionality of the tensors in the quantized space. Inputs to the modules must be in this format as well.
- **num_embeddings** (*int*) – number of vectors in the quantized space.
- **commitment_cost** (*float*) – scalar which controls the weighting of the loss terms (see equation 4 in the paper - this variable is Beta).
- **dtype** (*Any*) – dtype for the embeddings variable, defaults to `float32`.
- **name** (*Optional[str]*) – name of the module.

__call__(*inputs*, *is_training*)

Connects the module to some inputs.

Parameters

- **inputs** – Tensor, final dimension must be equal to `embedding_dim`. All other leading dimensions will be flattened and treated as a large batch.
- **is_training** – boolean, whether this connection is to training data.

Returns

Dictionary containing the following keys and values:

- `quantize`: Tensor containing the quantized version of the input.
- `loss`: Tensor containing the loss to optimize.
- `perplexity`: Tensor containing the perplexity of the encodings.

- `encodings`: Tensor containing the discrete encodings, ie which element of the quantized space each input element was mapped to.
- `encoding_indices`: Tensor containing the discrete encoding indices, ie which element of the quantized space each input element was mapped to.

Return type dict

quantize(*encoding_indices*)
Returns embedding tensor for a batch of indices.

VectorQuantizerEMA

class haiku.nets.**VectorQuantizerEMA**(*embedding_dim*, *num_embeddings*, *commitment_cost*, *decay*, *epsilon=1e-05*, *dtype=<class 'jax._src.numpy.lax_numpy.float32'>*, *cross_replica_axis=None*, *name=None*)
Haiku module representing the VQ-VAE layer.

Implements a slightly modified version of the algorithm presented in "Neural Discrete Representation Learning" by van den Oord et al. https://arxiv.org/abs/1711.00937

The difference between *VectorQuantizerEMA* and *VectorQuantizer* is that this module uses *ExponentialMovingAverage*s to update the embedding vectors instead of an auxiliary loss. This has the advantage that the embedding updates are independent of the choice of optimizer (SGD, RMSProp, Adam, K-Fac, ...) used for the encoder, decoder and other parts of the architecture. For most experiments the EMA version trains faster than the non-EMA version.

Input any tensor to be quantized. Last dimension will be used as space in which to quantize. All other dimensions will be flattened and will be seen as different examples to quantize.

The output tensor will have the same shape as the input.

For example a tensor with shape `[16, 32, 32, 64]` will be reshaped into `[16384, 64]` and all `16384` vectors (each of 64 dimensions) will be quantized independently.

embedding_dim
integer representing the dimensionality of the tensors in the quantized space. Inputs to the modules must be in this format as well.

num_embeddings
integer, the number of vectors in the quantized space.

commitment_cost
scalar which controls the weighting of the loss terms (see equation 4 in the paper).

decay
float, decay for the moving averages.

epsilon
small float constant to avoid numerical instability.

__init__(*embedding_dim*, *num_embeddings*, *commitment_cost*, *decay*, *epsilon=1e-05*, *dtype=<class 'jax._src.numpy.lax_numpy.float32'>*, *cross_replica_axis=None*, *name=None*)
Initializes a VQ-VAE EMA module.

Parameters

- **embedding_dim** – integer representing the dimensionality of the tensors in the quantized space. Inputs to the modules must be in this format as well.
- **num_embeddings** – integer, the number of vectors in the quantized space.

- **commitment_cost** – scalar which controls the weighting of the loss terms (see equation 4 in the paper - this variable is Beta).
- **decay** – float between 0 and 1, controls the speed of the Exponential Moving Averages.
- **epsilon** (*float*) – small constant to aid numerical stability, default `1e-5`.
- **dtype** (*Any*) – dtype for the embeddings variable, defaults to `float32`.
- **cross_replica_axis** (*Optional[str]*) – If not `None`, it should be a string representing the axis name over which this module is being run within a `jax.pmap()`. Supplying this argument means that cluster statistics and the perplexity are calculated across all replicas on that axis.
- **name** (*Optional[str]*) – name of the module.

__call__(*inputs*, *is_training*)

Connects the module to some inputs.

Parameters

- **inputs** – Tensor, final dimension must be equal to `embedding_dim`. All other leading dimensions will be flattened and treated as a large batch.
- **is_training** – boolean, whether this connection is to training data. When this is set to `False`, the internal moving average statistics will not be updated.

Returns

Dictionary containing the following keys and values:

- `quantize`: Tensor containing the quantized version of the input.
- `loss`: Tensor containing the loss to optimize.
- `perplexity`: Tensor containing the perplexity of the encodings.
- `encodings`: Tensor containing the discrete encodings, ie which element of the quantized space each input element was mapped to.
- `encoding_indices`: Tensor containing the discrete encoding indices, ie which element of the quantized space each input element was mapped to.

Return type dict

quantize(*encoding_indices*)

Returns embedding tensor for a batch of indices.

1.6 JAX Fundamentals

1.6.1 Control Flow

cond(*args, **kwargs)	Equivalent to `jax.lax.cond()` but with Haiku state passed in/out.
fori_loop(lower, upper, body_fun, init_val)	Equivalent to `jax.lax.fori_loop()` with Haiku state passed in/out.
scan(f, init, xs[, length, reverse, unroll])	Equivalent to `jax.lax.scan()` but with Haiku state passed in/out.

continues on next page

Table 19 – continued from previous page

switch(index, branches, operand)	Equivalent to `jax.lax.switch()` but with Haiku state passed in/out.
while_loop(cond_fun, body_fun, init_val)	Equivalent to jax.lax.while_loop with Haiku state threaded in/out.

cond

haiku.**cond**(**args*, ***kwargs*)
Equivalent to `jax.lax.cond()` but with Haiku state passed in/out.

fori_loop

haiku.**fori_loop**(*lower*, *upper*, *body_fun*, *init_val*)
Equivalent to `jax.lax.fori_loop()` with Haiku state passed in/out.

scan

haiku.**scan**(*f*, *init*, *xs*, *length=None*, *reverse=False*, *unroll=1*)
Equivalent to `jax.lax.scan()` but with Haiku state passed in/out.

switch

haiku.**switch**(*index*, *branches*, *operand*)
Equivalent to `jax.lax.switch()` but with Haiku state passed in/out.

while_loop

haiku.**while_loop**(*cond_fun*, *body_fun*, *init_val*)
Equivalent to jax.lax.while_loop with Haiku state threaded in/out.

1.6.2 JAX Transforms

eval_shape(fun, *args, **kwargs)	Equivalent to jax.eval_shape with any changed Haiku state discarded.
grad(fun[, argnums, has_aux, holomorphic])	Creates a function which evaluates the gradient of `fun`.
jit(fun, *[, static_argnums, ...])	Equivalent to jax.jit but passing Haiku state.
remat(fun[, concrete, prevent_cse, policy])	Equivalent to jax.checkpoint but passing Haiku state.
value_and_grad(fun[, argnums, has_aux, ...])	Creates a function which evaluates both `fun` and the grad of `fun`.
vmap(fun[, in_axes, out_axes, axis_name, ...])	Equivalent to `jax.vmap()` with module parameters/state not mapped.

eval_shape

haiku.**eval_shape**(*fun*, **args*, ***kwargs*)
Equivalent to jax.eval_shape with any changed Haiku state discarded.

grad

haiku.**grad**(*fun*, *argnums=0*, *has_aux=False*, *holomorphic=False*)
Creates a function which evaluates the gradient of `fun`.

NOTE: You only need this in a very specific case that you want to take a gradient **inside** a *`transform()`*ed function and the function you are differentiating uses *`set_state()`*. For example:

```
>>> class MyModule(hk.Module):
...   def __call__(self, x):
...     hk.set_state("last", x ** 2)
...     return x ** 2
```

```
>>> def f(x):
...   m = MyModule()
...   g = hk.grad(m)(x)
...   return g
```

```
>>> f = hk.transform_with_state(f)
>>> x = jnp.array(2.)
>>> params, state = jax.jit(f.init)(None, x)
>>> state["my_module"]["last"]
DeviceArray(4., dtype=float32, weak_type=True)
```

Parameters

- **fun** – Function to be differentiated. Its arguments at positions specified by `argnums` should be arrays, scalars, or standard Python containers. It should return a scalar (which includes arrays with shape `()` but not arrays with shape `(1,)` etc.)
- **argnums** – Optional, integer or tuple of integers. Specifies which positional argument(s) to differentiate with respect to (default 0).
- **has_aux** – Optional, bool. Indicates whether `fun` returns a pair where the first element is considered the output of the mathematical function to be differentiated and the second element is auxiliary data. Default False.
- **holomorphic** – Optional, bool. Indicates whether `fun` is promised to be holomorphic. Default False.

Returns A function with the same arguments as `fun`, that evaluates the gradient of `fun`. If *argnums* is an integer then the gradient has the same shape and type as the positional argument indicated by that integer. If argnums is a tuple of integers, the gradient is a tuple of values with the same shapes and types as the corresponding arguments. If `has_aux` is True then a pair of `gradient, auxiliary_data` is returned.

For example:

```
>>> grad_tanh = jax.grad(jax.numpy.tanh)
>>> print(grad_tanh(0.2))
0.96...
```

jit

haiku.**jit**(*fun*, *, *static_argnums=None*, *static_argnames=None*, *device=None*, *backend=None*, *donate_argnums=()*, *inline=False*)

Equivalent to jax.jit but passing Haiku state.

Return type ~F

remat

haiku.**remat**(*fun*, *concrete=False*, *prevent_cse=True*, *policy=None*)

Equivalent to jax.checkpoint but passing Haiku state.

Return type Callable

value_and_grad

haiku.**value_and_grad**(*fun*, *argnums=0*, *has_aux=False*, *holomorphic=False*)

Creates a function which evaluates both `fun` and the grad of `fun`.

NOTE: You only need this in a very specific case that you want to take a gradient **inside** a *transform()*ed function and the function you are differentiating uses *set_state()*. For example:

```
>>> class MyModule(hk.Module):
...   def __call__(self, x):
...     hk.set_state("last", jnp.sum(x))
...     return x ** 2
```

```
>>> def f(x):
...   m = MyModule()
...   y, g = hk.value_and_grad(m)(x)
...   return y, g
```

```
>>> f = hk.transform_with_state(f)
>>> x = jnp.array(2.)
>>> _ = jax.jit(f.init)(None, x)
```

Parameters

- **fun** – Function to be differentiated. Its arguments at positions specified by `argnums` should be arrays, scalars, or standard Python containers. It should return a scalar (which includes arrays with shape `()` but not arrays with shape `(1,)` etc.)
- **argnums** – Optional, integer or tuple of integers. Specifies which positional argument(s) to differentiate with respect to (default 0).
- **has_aux** – Optional, bool. Indicates whether `fun` returns a pair where the first element is considered the output of the mathematical function to be differentiated and the second element is auxiliary data. Default False.
- **holomorphic** – Optional, bool. Indicates whether `fun` is promised to be holomorphic. Default False.

Returns A function with the same arguments as `fun` that evaluates both `fun` and the gradient of `fun` and returns them as a pair (a two-element tuple). If `argnums` is an integer then the gradient has the same shape and type as the positional argument indicated by that integer. If

argnums is a tuple of integers, the gradient is a tuple of values with the same shapes and types as the corresponding arguments.

vmap

haiku.**vmap**(*fun*, *in_axes=0*, *out_axes=0*, *axis_name=None*, *, *split_rng=False*)

Equivalent to `jax.vmap()` with module parameters/state not mapped.

The behaviour of Haiku random key APIs under *vmap()* is controlled by the `split_rng` argument:

```
.. doctest::
```

```
>>> x = jnp.arange(2)
>>> f = hk.vmap(lambda _: hk.next_rng_key(), split_rng=False)
>>> key1, key2 = f(x)
>>> assert (key1 == key2).all()
```

```
>>> f = hk.vmap(lambda _: hk.next_rng_key(), split_rng=True)
>>> key1, key2 = f(x)
>>> assert not (key1 == key2).all()
```

Random numbers in Haiku are typically used for two things, firstly for initialising model parameters, and secondly for creating random samples as part of the forward pass of a neural network (e.g. for dropout). If you are using *vmap()* with a module that uses Haiku random keys for both (e.g. you don't pass keys explicitly into the network), then it is quite likely that you will want to vary the value of `split_rng` depending on whether we are initalizing (e.g. creating model parameters) or applying the model. An easy way to do this is to set `split_rng=(not hk.running_init())`.

Parameters

- **fun** (*Callable[.., Any]*) – See `jax.vmap()`.
- **in_axes** – See `jax.vmap()`.
- **out_axes** – See `jax.vmap()`.
- **axis_name** (*Optional[str]*) – See `jax.vmap()`.
- **split_rng** (*bool*) – Controls whether random key APIs in Haiku (e.g. *next_rng_key()*) return different (aka. the internal key is split before calling your mapped function) or the same (aka. the internal key is broadcast before calling your mapped fucntion) key. See the docstring for examples.

Return type Callable[.., Any]

Returns See `jax.vmap()`.

1.7 Mixed Precision

1.7.1 Automatic Mixed Precision

set_policy(cls, policy)	Uses the given policy for all instances of the module class.

continues on next page

Table 21 – continued from previous page

current_policy()	Retrieves the currently active policy in the current context.
get_policy(cls)	Retrieves the currently active policy for the given class.
clear_policy(cls)	Clears any policy assocated with the given class.

set_policy

haiku.mixed_precision.**set_policy**(*cls*, *policy*)

Uses the given policy for all instances of the module class.

NOTE: Policies are only applied to modules created in the current thread.

A mixed precision policy describes how inputs, module parameters and module outputs should be cast at runtime. By applying a policy to a given type of module, you can control how all instances of that module behave in your program.

For example, you might want to try running a ResNet50 model in a mixture of `float16` and `float32` on GPU to get higher throughput. To do so you can apply a mixed precision policy to the ResNet50 type that will create parameters in `float32`, but cast them to `float16` before use, along with all module inputs:

```
>>> policy = jmp.get_policy('params=float32,compute=float16,output=float32')
>>> hk.mixed_precision.set_policy(hk.nets.ResNet50, policy)
>>> net = hk.nets.ResNet50(4)
>>> x = jnp.ones([4, 224, 224, 3])
>>> net(x, is_training=True)
DeviceArray([[nan, nan, nan, nan],
             [nan, nan, nan, nan],
             [nan, nan, nan, nan],
             [nan, nan, nan, nan]], dtype=float32)
```

Oh no, nan! This is because modules like batch norm are not numerically stable in `float16`. To address this, we apply a second policy to our batch norm modules to keep them in full precision. We are careful to return a `float16` output from the module such that subsequent modules receive `float16` input:

```
>>> policy = jmp.get_policy('params=float32,compute=float32,output=float16')
>>> hk.mixed_precision.set_policy(hk.BatchNorm, policy)
>>> net(x, is_training=True)
DeviceArray([[0., 0., 0., 0.],
             [0., 0., 0., 0.],
             [0., 0., 0., 0.],
             [0., 0., 0., 0.]], dtype=float32)
```

For a fully worked mixed precision example see the imagenet example in Haiku's examples directory. This example shows mixed precision on GPU offering a 2x speedup in training time with only a small impact on final top-1 accuracy.

```
>>> hk.mixed_precision.clear_policy(hk.nets.ResNet50)
>>> hk.mixed_precision.clear_policy(hk.BatchNorm)
```

Parameters

- **cls** (*Type[hk.Module]*) – A Haiku module class.
- **policy** (*jmp.Policy*) – A JMP policy to apply to the module.

See also:

current_policy(): Retrieves the currently active policy (if any). *clear_policy()*: Clears any policies associated with a class. *get_policy()*: Gets the policy for a given class.

current_policy

haiku.mixed_precision.**current_policy**()

Retrieves the currently active policy in the current context.

Return type Optional[jmp.Policy]

Returns The currently active mixed precision policy, or `None`.

See also:

clear_policy(): Clears any policies associated with a class. *get_policy()*: Gets the policy for a given class. *set_policy()*: Sets a policy for a given class.

get_policy

haiku.mixed_precision.**get_policy**(*cls*)

Retrieves the currently active policy for the given class.

Note that policies applied explicitly to a top level class (e.g. `ResNet`) will be applied implicitly to all child modules (e.g. `ConvND`) called from the parent. This function only returns policies that have been applied explicitly (e.g. via *set_policy()*).

Parameters **cls** (*Type[hk.Module]*) – A Haiku module class.

Return type Optional[jmp.Policy]

Returns A JMP policy that is used for the given class, or `None` if one is not active.

See also:

current_policy(): Retrieves the currently active policy (if any). *clear_policy()*: Clears any policies associated with a class. *set_policy()*: Sets a policy for a given class.

clear_policy

haiku.mixed_precision.**clear_policy**(*cls*)

Clears any policy assocated with the given class.

Parameters **cls** (*Type[hk.Module]*) – A Haiku module class.

See also:

current_policy(): Retrieves the currently active policy (if any). *get_policy()*: Gets the policy for a given class. *set_policy()*: Sets a policy for a given class.

1.8 Experimental

1.8.1 TensorFlow Profiler

named_call(fun, *[, name])	Wraps a function in an XLA name_scope and maintains Haiku state.
profiler_name_scopes([enabled])	Enable/disable profiler name_scopes on all haiku module methods.

named_call

haiku.experimental.**named_call**(*fun*, *, *name=None*)

Wraps a function in an XLA name_scope and maintains Haiku state.

Return type Callable[.., Any]

profiler_name_scopes

haiku.experimental.**profiler_name_scopes**(*enabled=True*)

Enable/disable profiler name_scopes on all haiku module methods.

Note: currently only enables for `__call__`. See: *named_call()* if you want to annotate other methods explicitly.

Parameters **enabled** – Whether to enable name scopes or not.

Returns The previous value of the name_scopes setting.

1.8.2 Graphviz Visualisation

to_dot(fun)	Converts a function using Haiku modules to a dot graph.
abstract_to_dot(fun)	Converts a function using Haiku modules to a dot graph.

abstract_to_dot

haiku.experimental.**abstract_to_dot**(*fun*)

Converts a function using Haiku modules to a dot graph.

Same as *to_dot()* but uses JAX's abstract interpretation machinery to evaluate the function without requiring concrete inputs. Valid inputs for the wrapped function include `jax.ShapeDtypeStruct`.

abstract_to_dot() does not support data-dependent control-flow, because no concrete values are provided to the function.

Parameters **fun** (*Callable[.., Any]*) – A function using Haiku modules.

Return type Callable[.., str]

Returns A function that returns the source code string to a graphviz graph describing the operations executed by the given function clustered by Haiku module.

See also:

to_dot(): Generates a graphviz graph using concrete inputs.

to_dot

haiku.experimental.**to_dot**(*fun*)

Converts a function using Haiku modules to a dot graph.

To view the resulting graph in Google Colab or an iPython notebook use the `graphviz` package:

```
dot = hk.experimental.to_dot(f)(x)
import graphviz
graphviz.Source(dot)
```

Parameters **fun** (*Callable[.., Any]*) – A function using Haiku modules.

Return type Callable[.., str]

Returns A function that returns the source code string to a graphviz graph describing the operations executed by the given function clustered by Haiku module.

See also:

abstract_to_dot(): Generates a graphviz graph using abstract inputs.

1.8.3 Summarisation

tabulate(f, *[, columns, filters, ...])	Produces a summarised view of the execution of `f`.
eval_summary(f)	Records module method calls performed by `f`.
ArraySpec(shape, dtype)	Shaped and sized specification of an array.
MethodInvocation(module_details, args_spec, ...)	Record of a method being invoked on a given module.
ModuleDetails(module, method_name, params, state)	Module and method related information.

tabulate

haiku.experimental.**tabulate**(*f*, *, *columns=('module', 'config', 'owned_params', 'input', 'output', 'params_size', 'params_bytes')*, *filters=('has_output')*, *tabulate_kwargs={'tablefmt': 'grid'}*)

Produces a summarised view of the execution of `f`.

```
>>> def f(x):
...   return hk.nets.MLP([300, 100, 10])(x)
>>> x = jnp.ones([8, 28 * 28])
>>> f = hk.transform(f)
>>> print(hk.experimental.tabulate(f)(x))
+------------------------+-----------------------------------------+------------
↪-----+------------+------------+---------------+---------------+
| Module                 | Config                                  | Module␣
↪params   | Input      | Output     |   Param count |   Param bytes |
+========================+=========================================+================
| mlp (MLP)              | MLP(output_sizes=[300, 100, 10])        |           ␣
↪     | f32[8,784] | f32[8,10]  |       266,610 |       1.07 MB |
```

(continues on next page)

(continued from previous page)

```
+------------------------+-----------------------------------------+------------
↪-----+-----------+-----------+---------------+---------------+
| mlp/~/linear_0 (Linear) | Linear(output_size=300, name='linear_0') | w: f32[784,
↪300] | f32[8,784] | f32[8,300] |       235,500 |     942.00 KB |
|   └ mlp (MLP)           |                                         | b:␣
↪f32[300]     |           |           |               |               |
+------------------------+-----------------------------------------+------------
↪-----+-----------+-----------+---------------+---------------+
| mlp/~/linear_1 (Linear) | Linear(output_size=100, name='linear_1') | w: f32[300,
↪100] | f32[8,300] | f32[8,100] |        30,100 |     120.40 KB |
|   └ mlp (MLP)           |                                         | b:␣
↪f32[100]     |           |           |               |               |
+------------------------+-----------------------------------------+------------
↪-----+-----------+-----------+---------------+---------------+
| mlp/~/linear_2 (Linear) | Linear(output_size=10, name='linear_2')  | w: f32[100,
↪10]  | f32[8,100] | f32[8,10]  |         1,010 |       4.04 KB |
|   └ mlp (MLP)           |                                         | b: f32[10]␣
↪     |           |           |               |               |
+------------------------+-----------------------------------------+------------
↪-----+-----------+-----------+---------------+---------------+
```

Possible values for `columns`:

- `module`: Displays module and method name.
- `config`: Displays the constructor arguments used for the module.
- `owned_params`: Displays parameters directly owned by this module.
- `input`: Displays module inputs.
- `output`: Displays module output.
- `params_size`: Displays the number of parameters
- `params_bytes`: Displays parameter size in bytes.

Possible values for `filters`:

- `has_output`: Only include methods returning a value other than `None`.
- `has_params`: Removes methods from modules that do not have parameters.

Parameters

- **f** (*Union[Callable[.., Any], hk.Transformed, hk.TransformedWithState]*) – A function to transform OR one of the init/apply functions from Haiku or the result of `transform()` or `transform_with_state()`.
- **columns** (*Optional[Sequence[str]]*) – A list of column names to enable.
- **filters** (*Optional[Sequence[str]]*) – A list of filters to apply to remove certain module methods.
- **tabulate_kwargs** – Keyword arguments to pass to `tabulate.tabulate(..)`.

Return type Callable[.., str]

Returns A callable that takes the same arguments as `f` but returns a string summarising the modules used during the execution of `f`.

See also:

eval_summary(): Raw data used to generate this table.

eval_summary

haiku.experimental.**eval_summary**(*f*)

Records module method calls performed by `f`.

```
>>> f = lambda x: hk.nets.MLP([300, 100, 10])(x)
>>> x = jnp.ones([8, 28 * 28])
>>> for i in hk.experimental.eval_summary(f)(x):
...   print("mod := {:14} | in := {} out := {}".format(
...       i.module_details.module.module_name, i.args_spec[0], i.output_spec))
mod := mlp            | in := f32[8,784] out := f32[8,10]
mod := mlp/~/linear_0 | in := f32[8,784] out := f32[8,300]
mod := mlp/~/linear_1 | in := f32[8,300] out := f32[8,100]
mod := mlp/~/linear_2 | in := f32[8,100] out := f32[8,10]
```

Parameters **f** (*Union[Callable[.., Any], hk.Transformed, hk.TransformedWithState]*) – A function or transformed function to trace.

Return type Callable[.., Sequence[*MethodInvocation*]]

Returns A callable taking the same arguments as the provided function, but returning a sequence of *MethodInvocation* instances revealing the methods called on each module when applying `f`.

See also:

tabulate(): Pretty prints a summary of the execution of a function.

ArraySpec

class haiku.experimental.**ArraySpec**(*shape*, *dtype*)

Shaped and sized specification of an array.

shape
Shape of the array.

dtype
DType of the array.

MethodInvocation

class haiku.experimental.**MethodInvocation**(*module_details*, *args_spec*, *kwargs_spec*, *output_spec*, *context*, *call_stack*)

Record of a method being invoked on a given module.

module_details
Details about which module and method were invoked.

args_spec
Positional arguments to the method invocation with arrays replaced by *ArraySpec*.

kwargs_spec
Keyword arguments to the method invocation with arrays replaced by *ArraySpec*.

output_spec
Output of the method invocation with arrays replaced by *ArraySpec*.

context
Additional context information for the method call as provided by `intercept_methods()`.

call_stack
Stack of modules currently active while calling this module method. For example if `A` calls `B` which calls `C` then the call stack for `C` will be `[B_DETAILS, A_DETAILS]`.

ModuleDetails

class haiku.experimental.**ModuleDetails**(*module*, *method_name*, *params*, *state*)
Module and method related information.

module
A *Module* instance.

method_name
The method name that was invoked on the module.

params
The modules params dict with arrays converted to *ArraySpec*.

state
The modules state dict with arrays converted to *ArraySpec*.

1.8.4 Managing State

name_scope(name)	Context manager which adds a prefix to all new modules, params or state.
name_like(method_name)	Allows a method to be named like some other method.
lift(init_fn[, name])	Lifts the given init fn to a function in the current Haiku namespace.
lift_with_state(init_fn[, name])	Lifts the given init fn to a function in the current Haiku namespace.
LiftWithStateUpdater(name)	Handles updating the state for a *lift_with_state* computation.

name_scope

haiku.experimental.**name_scope**(*name*)
Context manager which adds a prefix to all new modules, params or state.

```
>>> with hk.experimental.name_scope("my_name_scope"):
...   net = hk.Linear(1, name="my_linear")
>>> net.module_name
'my_name_scope/my_linear'
```

When used inside a module, any submodules, parameters or state created inside the name scope will have a prefix added to their names:

```
>>> class MyModule(hk.Module):
...   def __call__(self, x):
```

(continues on next page)

(continued from previous page)

```
...     with hk.experimental.name_scope("my_name_scope"):
...       submodule = hk.Linear(1, name="submodule")
...       w = hk.get_parameter("w", [], init=jnp.ones)
...     return submodule(x) + w
```

```
>>> f = hk.transform(lambda x: MyModule()(x))
>>> params = f.init(jax.random.PRNGKey(42), jnp.ones([1, 1]))
>>> jax.tree_map(jnp.shape, params)
{'my_module/my_name_scope': {'w': ()},
 'my_module/my_name_scope/submodule': {'b': (1,), 'w': (1, 1)}}
```

Name scopes are very similar to putting all of the code inside the context manager inside a method on a `Module` with the name you provide. Behind the scenes this is precisely how name scopes are implemented.

If you are familiar with TensorFlow then Haiku's *name_scope()* is similar to `tf.variable_scope(..)` in TensorFlow 1 and `tf.name_scope(..)` in TensorFlow 1 and 2 in that it changes the names associated with modules, parameters and state.

Parameters **name** (*str*) – The name scope to use (e.g. `"foo"` or `"foo/bar"`).

Return type ContextManager[None]

Returns A single use context manager that when active prefixes new modules, parameters or state with the given name.

name_like

haiku.experimental.**name_like**(*method_name*)

Allows a method to be named like some other method.

In Haiku submodules are named based on the name of their parent module and the method in which they are created. When refactoring code it may be desirable to maintain previous names in order to keep checkpoint compatibility, this can be achieved using *name_like()*.

As an example, consider the following toy autoencoder:

```
>>> class Autoencoder(hk.Module):
...   def __call__(self, x):
...     z = hk.Linear(10, name="enc")(x)  # name: autoencoder/enc
...     y = hk.Linear(10, name="dec")(z)  # name: autoencoder/dec
...     return y
```

If we want to refactor this such that users can encode or decode, we would create two methods (encode, decode) which would create and apply our modules. In order to retain checkpoint compatibility with the original module we can use *name_like()* to name those submodules as if they were created inside `__call__`:

```
>>> class Autoencoder(hk.Module):
...   @hk.experimental.name_like("__call__")
...   def encode(self, x):
...     return hk.Linear(10, name="enc")(x)  # name: autoencoder/enc
...
...   @hk.experimental.name_like("__call__")
...   def decode(self, z):
...     return hk.Linear(10, name="dec")(z)  # name: autoencoder/dec
...
...   def __call__(self, x):
...     return self.decode(self.encode(x))
```

One sharp edge is if users rely on Haiku's numbering to take care of giving unique names and refactor using *name_like()*. For example when refactoring the following:

```
>>> class Autoencoder(hk.Module):
...   def __call__(self, x):
...     y = hk.Linear(10)(z)  # name: autoencoder/linear_1
...     z = hk.Linear(10)(x)  # name: autoencoder/linear
...     return y
```

To use *name_like()*, the unnamed linear modules in encode/decode will end up with the same name (both: `autoencoder/linear`) because module numbering is only applied within a method:

```
>>> class Autoencoder(hk.Module):
...   @hk.experimental.name_like("__call__")
...   def encode(self, x):
...     return hk.Linear(10)(x)  # name: autoencoder/linear
...
...   @hk.experimental.name_like("__call__")
...   def decode(self, z):
...     return hk.Linear(10)(z)  # name: autoencoder/linear  <-- NOT INTENDED
```

To fix this case you need to explicitly name the modules within the method with their former name:

```
>>> class Autoencoder(hk.Module):
...   @hk.experimental.name_like("__call__")
...   def encode(self, x):
...     return hk.Linear(10, name="linear")(x)    # name: autoencoder/linear
...
...   @hk.experimental.name_like("__call__")
...   def decode(self, z):
...     return hk.Linear(10, name="linear_1")(z)  # name: autoencoder/linear_1
```

Parameters **method_name** (*str*) – The name of a method whose name we should adopt. This method does not actually have to be defined on the class.

Return type Callable[[T], T]

Returns A decorator that when applied to a method marks it as having a different name.

lift_with_state

haiku.experimental.**lift_with_state**(*init_fn*, *name='lifted'*)

Lifts the given init fn to a function in the current Haiku namespace.

This function returns two objects. The first is a callable that runs your init function with slightly behaviour based on init vs. apply time. The second is an updater that can be used to pass updated state values that result from running your apply function. See later in the docs for a worked example.

During init, the returned callable will run the given `init_fn`, and include the resulting params/state in the outer transform's dictionaries. During `apply`, the returned callable will instead pull the relevant params/state from the outer transform's dictionaries.

Must be called inside `transform_with_state()`, and be passed the `init` member of a `TransformedWithState`.

The user must ensure that the given `init` does not accidentally catch modules from an outer `transform_with_state()` via functional closure.

Example

```
>>> def g(x):
...   return hk.nets.ResNet50(1)(x, True)
>>> g = hk.transform_with_state(g)
>>> params_and_state_fn, updater = (
...   hk.experimental.lift_with_state(g.init, name='f_lift'))
>>> init_rng = hk.next_rng_key() if hk.running_init() else None
>>> x = jnp.ones([1, 224, 224, 3])
>>> params, state = params_and_state_fn(init_rng, x)
>>> out, state = g.apply(params, state, None, x)
>>> updater.update(state)
```

Parameters

- **init_fn** (*Callable[.., Tuple[hk.Params, hk.State]]*) – The `init` function from an `TransformedWithState`.
- **name** (*str*) – A string name to prefix parameters with.

Return type Tuple[Callable[.., Tuple[hk.Params, hk.State]], *LiftWithStateUpdater*]

Returns A callable that during `init` injects parameter values into the outer context and during `apply` reuses parameters from the outer context. In both cases returns parameter values to be used with an `apply` function. The `init` function additionally returns an object used to update the outer context with new state after `apply` is called.

LiftWithStateUpdater

class haiku.experimental.**LiftWithStateUpdater**(*name*)

Handles updating the state for a *lift_with_state* computation.

1.8.5 Optimizations

optimize_rng_use(fun)	Optimizes a RNG key splitting in `fun`.
layer_stack(num_layers[, ...])	Utility to wrap a Haiku function and recursively apply it to an input.

optimize_rng_use

haiku.experimental.**optimize_rng_use**(*fun*)

Optimizes a RNG key splitting in `fun`.

Our strategy here is to use abstract interpretation to run your function twice, the first time we use `jax.eval_shape()` to avoid spending any flops and simply observe how many times you call *next_rng_key()*. We then run your function again, but this time we reserve enough RNG keys ahead of time such that we only need to call `jax.random.split()` once.

In the following example, we need three random samples for our weight matrices in our 3-layer MLP. To draw these samples we use *next_rng_key()* which will split a new key for each sample. By using *optimize_rng_use()* Haiku will pre-allocate exactly enough RNGs for `f` to be evaluated by splitting the input key once and only once. For large models (unlike this example) this can lead to a significant reduction in compilation time for `init`:

```
>>> def f(x):
...   net = hk.nets.MLP([300, 100, 10])
...   return net(x)
>>> f = hk.experimental.optimize_rng_use(f)
>>> f = hk.transform(f)
>>> params = f.init(jax.random.PRNGKey(42), jnp.ones([1, 1]))
```

Parameters **fun** – A function to wrap.

Returns A function that applies `fun` but only requires one call to `jax.random.split()` by Haiku.

layer_stack

haiku.experimental.**layer_stack**(*num_layers*, *with_per_layer_inputs=False*, *unroll=1*, *name=None*)

Utility to wrap a Haiku function and recursively apply it to an input.

This can be used to improve model compile times.

A function is valid if it uses only explicit position parameters, and its return type matches its input type. The position parameters can be arbitrarily nested structures with `jnp.ndarray` at the leaf nodes. Note that kwargs are not supported, neither are functions with variable number of parameters (specified by `*args`).

If `with_per_layer_inputs=False` then the new, wrapped function can be understood as performing the following:

```
>>> f = lambda x: x+1
>>> num_layers = 4
>>> x = 0
>>> for i in range(num_layers):
...   x = f(x)
>>> x
4
```

And if `with_per_layer_inputs=True`, assuming `f` takes two arguments on top of `x`:

```
>>> f = lambda x, y0, y1: (x+1, y0+y1)
>>> num_layers = 4
>>> x = 0
>>> ys_0 = [1, 2, 3, 4]
>>> ys_1 = [5, 6, 7, 8]
>>> zs = []
>>> for i in range(num_layers):
...   x, z = f(x, ys_0[i], ys_1[i])
...   zs.append(z)
>>> x, zs
(4, [6, 8, 10, 12])
```

The code using `layer_stack` for the above function would be:

```
>>> f = lambda x, y0, y1: (x+1, y0+y1)
>>> num_layers = 4
>>> x = 0
>>> ys_0 = jnp.array([1, 2, 3, 4])
>>> ys_1 = jnp.array([5, 6, 7, 8])
>>> stack = hk.experimental.layer_stack(num_layers,
```

(continues on next page)

(continued from previous page)

```
...                                          with_per_layer_inputs=True)
>>> x, zs = stack(f)(x, ys_0, ys_1)
>>> x, zs
(DeviceArray(4, dtype=int32, weak_type=True),
    DeviceArray([ 6,  8, 10, 12], dtype=int32))
```

Check the tests in `layer_stack_test.py` for further examples.

Crucially, any parameters created inside `f` will not be shared across iterations.

Parameters

- **num_layers** (*int*) – The number of times to iterate the wrapped function.
- **with_per_layer_inputs** – Whether or not to pass per-layer inputs to the wrapped function.
- **unroll** (*int*) – the unroll used by `scan`.
- **name** (*Optional[str]*) – name of the Haiku context.

Returns Callable that will produce a layer stack when called with a valid function.

1.9 Utilities

1.9.1 Data Structures

filter(predicate, structure)	Filters an input structure according to a user specified predicate.
is_subset(*, subset, superset)	Checks whether the leaves of subset appear in superset.
map(fn, structure)	Maps a function to an input structure accordingly.
merge(*structures)	Merges multiple input structures.
partition(predicate, structure)	Partitions the input structure in two according to a given predicate.
partition_n(fn, structure, n)	Partitions a structure into *n* structures.
to_haiku_dict(structure)	Returns a copy of the given two level structure.
to_immutable_dict(mapping)	Returns an immutable copy of the given mapping.
to_mutable_dict(mapping)	Turns an immutable FlatMapping into a mutable dict.
traverse(structure)	Iterates over a structure yielding module names, names and values.
tree_bytes(tree)	Sums the size in bytes of all arrays in a pytree.
tree_size(tree)	Sums the sizes of all arrays in a pytree.

filter

haiku.data_structures.**filter**(*predicate*, *structure*)

Filters an input structure according to a user specified predicate.

```
>>> params = {'linear': {'w': None, 'b': None}}
>>> predicate = lambda module_name, name, value: name == 'w'
>>> hk.data_structures.filter(predicate, params)
{'linear': {'w': None}}
```

Note: returns a new structure not a view.

Parameters

- **predicate** (*Callable[[str, str, T], bool]*) – criterion to be used to partition the input data. The `predicate` argument is expected to be a boolean function taking as inputs the name of the module, the name of a given entry in the module data bundle (e.g. parameter name) and the corresponding data.
- **structure** (*Mapping[str, Mapping[str, T]]*) – Haiku params or state data structure to be filtered.

Return type Mapping[str, Mapping[str, T]]

Returns All the input parameters or state as selected by the input predicate.

is_subset

haiku.data_structures.**is_subset**(*, *subset*, *superset*)

Checks whether the leaves of subset appear in superset.

Note that this is vacuously true in the case that both structures have no leaves:

```
>>> hk.data_structures.is_subset(subset={'a': {}}, superset={})
True
```

Parameters

- **subset** (*Mapping[str, Mapping[str, Any]]*) – The subset to check.
- **superset** (*Mapping[str, Mapping[str, Any]]*) – The superset to check.

Return type bool

Returns A boolean indicating whether all elements in subset are contained in superset.

map

haiku.data_structures.**map**(*fn*, *structure*)

Maps a function to an input structure accordingly.

```
>>> params = {'linear': {'w': 1.0, 'b': 2.0}}
>>> fn = lambda module_name, name, value: 2 * value if name == 'w' else value
>>> hk.data_structures.map(fn, params)
{'linear': {'b': 2.0, 'w': 2.0}}
```

Note: returns a new structure not a view.

Parameters

- **fn** (*Callable[[str, str, InT], OutT]*) – criterion to be used to map the input data. The `fn` argument is expected to be a boolean function taking as inputs the name of the module, the name of a given entry in the module data bundle (e.g. parameter name) and the corresponding data.
- **structure** (*Mapping[str, Mapping[str, InT]]*) – Haiku params or state data structure to be mapped.

Return type Mapping[str, Mapping[str, OutT]]

Returns All the input parameters or state as mapped by the input fn.

merge

haiku.data_structures.**merge**(**structures*)

Merges multiple input structures.

```
>>> weights = {'linear': {'w': None}}
>>> biases = {'linear': {'b': None}}
>>> hk.data_structures.merge(weights, biases)
{'linear': {'w': None, 'b': None}}
```

When structures are not disjoint the output will contain the value from the last structure for each path:

```
>>> weights1 = {'linear': {'w': 1}}
>>> weights2 = {'linear': {'w': 2}}
>>> hk.data_structures.merge(weights1, weights2)
{'linear': {'w': 2}}
```

Note: returns a new structure not a view.

Parameters ***structures** – One or more structures to merge.

Return type Mapping[str, Mapping[str, Any]]

Returns A single structure with an entry for each path in the input structures.

partition

haiku.data_structures.**partition**(*predicate*, *structure*)

Partitions the input structure in two according to a given predicate.

For a given set of parameters, you can use *partition()* to split them:

```
>>> params = {'linear': {'w': None, 'b': None}}
>>> predicate = lambda module_name, name, value: name == 'w'
>>> weights, biases = hk.data_structures.partition(predicate, params)
>>> weights
{'linear': {'w': None}}
>>> biases
{'linear': {'b': None}}
```

Note: returns new structures not a view.

Parameters

- **predicate** (*Callable[[str, str, jnp.ndarray], bool]*) – criterion to be used to partition the input data. The `predicate` argument is expected to be a boolean

function taking as inputs the name of the module, the name of a given entry in the module data bundle (e.g. parameter name) and the corresponding data.

- **structure** (*Mapping[str, Mapping[str, T]]*) – Haiku params or state data structure to be partitioned.

Return type Tuple[Mapping[str, Mapping[str, T]], Mapping[str, Mapping[str, T]]]

Returns

A tuple containing all the params or state as partitioned by the input predicate. Entries matching the predicate will be in the first structure, and the rest will be in the second.

partition_n

haiku.data_structures.**partition_n**(*fn*, *structure*, *n*)

Partitions a structure into *n* structures.

For a given set of parameters, you can use *partition_n()* to split them into `n` groups. For example, to split your parameters/gradients by module name:

```
>>> def partition_by_module(structure):
...   cnt = itertools.count()
...   d = collections.defaultdict(lambda: next(cnt))
...   fn = lambda m, n, v: d[m]
...   return hk.data_structures.partition_n(fn, structure, len(structure))
```

```
>>> structure = {f'layer_{i}': {'w': None, 'b': None} for i in range(3)}
>>> for substructure in partition_by_module(structure):
...   print(substructure)
{'layer_0': {'b': None, 'w': None}}
{'layer_1': {'b': None, 'w': None}}
{'layer_2': {'b': None, 'w': None}}
```

Parameters

- **fn** (*Callable[[str, str, T], int]*) – Callable returning which bucket in `[0, n)` the given element should be output.
- **structure** (*Mapping[str, Mapping[str, T]]*) – Haiku params or state data structure to be partitioned.
- **n** (*int*) – The total number of buckets.

Return type Tuple[Mapping[str, Mapping[str, T]], ..]

Returns A tuple of size `n`, where each element will contain the values for which the function returned the current index.

to_haiku_dict

haiku.data_structures.**to_haiku_dict**(*structure*)

Returns a copy of the given two level structure.

Uses the same mapping type as Haiku will return from `init` or `apply` functions.

Parameters **structure** (*Mapping[K, V]*) – A two level mapping to copy.

Return type Mapping[K, V]

Returns A new two level mapping with the same contents as the input.

to_immutable_dict

haiku.data_structures.**to_immutable_dict**(*mapping*)

Returns an immutable copy of the given mapping.

Return type Mapping[K, V]

to_mutable_dict

haiku.data_structures.**to_mutable_dict**(*mapping*)

Turns an immutable FlatMapping into a mutable dict.

traverse

haiku.data_structures.**traverse**(*structure*)

Iterates over a structure yielding module names, names and values.

NOTE: Items are iterated in key sorted order.

Parameters **structure** (*Mapping[str, Mapping[str, T]]*) – The structure to traverse.

Yields Tuples of the module name, name and value from the given structure.

Return type Generator[Tuple[str, str, T], None, None]

tree_bytes

haiku.data_structures.**tree_bytes**(*tree*)

Sums the size in bytes of all arrays in a pytree.

Note that this is the minimum size of the array (e.g. for a float32 we need at least 4 bytes) however on some accelerators buffers may occupy more memory due to padding/alignment constraints.

For example given a ResNet50 model:

```
>>> f = hk.transform_with_state(lambda x: hk.nets.ResNet50(1000)(x, True))
>>> rng = jax.random.PRNGKey(42)
>>> x = jnp.ones([128, 224, 224, 3])
>>> params, state = f.init(rng, x)
```

We can count the number of parameters and their size at f32:

```
>>> num_params = hk.data_structures.tree_size(params)
>>> byte_size = hk.data_structures.tree_bytes(params)
>>> print(f'{num_params} params, size: {byte_size / 1e6:.2f}MB')
25557032 params, size: 102.23MB
```

And compare that with casting our parameters to bf16:

```
>>> params = jax.tree_map(lambda x: x.astype(jnp.bfloat16), params)
>>> num_params = hk.data_structures.tree_size(params)
>>> byte_size = hk.data_structures.tree_bytes(params)
>>> print(f'{num_params} params, size: {byte_size / 1e6:.2f}MB')
25557032 params, size: 51.11MB
```

Parameters **tree** – A tree of jnp.ndarrays.

Return type int

Returns The total size in bytes of the array(s) in the input.

tree_size

haiku.data_structures.**tree_size**(*tree*)

Sums the sizes of all arrays in a pytree.

For example given a ResNet50 model:

```
>>> f = hk.transform_with_state(lambda x: hk.nets.ResNet50(1000)(x, True))
>>> rng = jax.random.PRNGKey(42)
>>> x = jnp.ones([128, 224, 224, 3])
>>> params, state = f.init(rng, x)
```

We can count the number of parameters and their size at f32:

```
>>> num_params = hk.data_structures.tree_size(params)
>>> byte_size = hk.data_structures.tree_bytes(params)
>>> print(f'{num_params} params, size: {byte_size / 1e6:.2f}MB')
25557032 params, size: 102.23MB
```

And compare that with casting our parameters to bf16:

```
>>> params = jax.tree_map(lambda x: x.astype(jnp.bfloat16), params)
>>> num_params = hk.data_structures.tree_size(params)
>>> byte_size = hk.data_structures.tree_bytes(params)
>>> print(f'{num_params} params, size: {byte_size / 1e6:.2f}MB')
25557032 params, size: 51.11MB
```

Parameters **tree** – A tree of jnp.ndarrays.

Return type int

Returns The total size (number of elements) of the array(s) in the input.

1.9.2 Testing

transform_and_run([f, seed, run_apply, ...])	Transforms the given function and runs init then (optionally) apply.

transform_and_run

haiku.testing.**transform_and_run**(*f=None*, *seed=42*, *run_apply=True*, *jax_transform=None*)

Transforms the given function and runs init then (optionally) apply.

Equivalent to:

```
>>> def f(x):
...   return x
>>> x = jnp.ones([])
>>> rng = jax.random.PRNGKey(42)
>>> f = hk.transform_with_state(f)
>>> params, state = f.init(rng, x)
>>> out = f.apply(params, state, rng, x)
```

This function makes it very convenient to unit test Haiku:

```
>>> class MyTest(unittest.TestCase):
...   @hk.testing.transform_and_run
...   def test_linear_output(self):
...     mod = hk.Linear(1)
...     out = mod(jnp.ones([1, 1]))
...     self.assertEqual(out.ndim, 2)
```

It can also be combined with `chex` to test all pure/jit/pmap versions of a function:

```
>>> class MyTest(unittest.TestCase):
...   @chex.all_variants
...   def test_linear_output(self):
...     @hk.testing.transform_and_run(jax_transform=self.variant)
...     def f(inputs):
...       mod = hk.Linear(1)
...       return mod(inputs)
...     out = f(jnp.ones([1, 1]))
...     self.assertEqual(out.ndim, 2)
```

And can also be useful in an interactive environment like ipython, Jupyter or Google Colaboratory:

```
>>> f = lambda x: hk.Bias()(x)
>>> hk.testing.transform_and_run(f)(jnp.ones([1, 1]))
DeviceArray([[1.]], dtype=float32)
```

See `transform()` for more details.

Parameters

- **f** (*Optional[Fn]*) – A function method to transform.
- **seed** (*Optional[int]*) – A seed to pass to init and apply.
- **run_apply** (*bool*) – Whether to run apply as well as init. Defaults to true.

- **jax_transform** (*Optional[Callable[[Fn], Fn]]*) – An optional jax transform to apply on the init and apply functions.

Return type T

Returns A function that *transform()*s `f` and runs `init` and optionally `apply`.

1.9.3 Conditional Computation

running_init()	Return True if running the `init` function of a Haiku transform.

running_init

haiku.**running_init**()

Return True if running the `init` function of a Haiku transform.

In general you should not need to gate behaviour of your module based on whether you are running `init` or `apply`, but sometimes (e.g. when making use of JAX control flow) this is required.

For example, if you want to use *switch()* to pick between experts, when we run your init function we need to ensure that params/state for all experts are created (unconditionally) but during apply we want to conditionally apply (and perhaps update the internal state) of only one of our experts:

```
>>> experts = [hk.nets.ResNet50(10) for _ in range(5)]
>>> x = jnp.ones([1, 224, 224, 3])
>>> if hk.running_init():
...   # During init unconditionally create params/state for all experts.
...   for expert in experts:
...     out = expert(x, is_training=True)
... else:
...   # During apply conditionally apply (and update) only one expert.
...   index = jax.random.randint(hk.next_rng_key(), [], 0, len(experts) - 1)
...   out = hk.switch(index, experts, x)
```

Return type bool

Returns True if running `init` otherwise False.

1.9.4 Functions

multinomial(rng, logits, num_samples)	Draws samples from a multinomial distribution.
one_hot(x, num_classes[, dtype])	Returns a one-hot version of indices.

multinomial

haiku.**multinomial**(*rng*, *logits*, *num_samples*)

Draws samples from a multinomial distribution.

DEPRECATED: Use `jax.random.categorical` instead.

Parameters

- **rng** – A JAX PRNGKey.
- **logits** – Unnormalized log-probabilities, where last dimension is categories.
- **num_samples** – Number of samples to draw.

Returns Chosen categories, of shape `logits.shape[:-1] + (num_samples,)`.

one_hot

haiku.**one_hot**(*x*, *num_classes*, *dtype=<class 'jax._src.numpy.lax_numpy.float32'>*)

Returns a one-hot version of indices.

DEPRECATED: Use `jax.nn.one_hot(x, num_classes).astype(dtype)` instead.

Parameters

- **x** – A tensor of indices.
- **num_classes** – Number of classes in the one-hot dimension.
- **dtype** – The dtype.

Returns

The one-hot tensor. If indices' shape is [A, B, . . .], shape is [A, B, . . . num_classes].

1.10 References

1.11 Haiku and `jax2tf`

`jax2tf` is an advanced JAX feature supporting staging JAX programs out as TensorFlow graphs.

This is a useful feature if you want to integrate with an existing TensorFlow codebase or tool. In this tutorial we will demonstrate defining a simple model in Haiku, converting it to TensorFlow as a `tf.Module` and then training it.

We'll then save the model as a TensorFlow SavedModel so it can be used later in other TensorFlow programs.

[1]:
```
!pip install dm-tree dm-sonnet tensorflow tensorflow_datasets ipywidgets matplotlib >/
↪dev/null
```

[2]:
```
import haiku as hk
import jax
import jax.numpy as jnp
from jax.experimental import jax2tf
import sonnet as snt
import tensorflow as tf
import tree
```

1.11.1 Define your model in JAX

First things first, we need to define our model using Haiku and JAX. For MNIST we can use a trivial model like an MLP.

We initialize the model using JAX and get initial parameter values. If you wanted you could additionally go on to train your model using JAX, but in this example we will do that in TensorFlow.

[3]:
```
def f(x):
  net = hk.nets.MLP([300, 100, 10])
  return net(x)

f = hk.transform(f)

rng = jax.random.PRNGKey(42)
x = jnp.ones([1, 28 * 28 * 1])
params = f.init(rng, x)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

1.11.2 Convert to TensorFlow

TensorFlow ships with a module abstraction that supports common tasks like collecting model parameters.

Sonnet is a library of `tf.Module` subclasses including common NN layers, optimizers and some metrics. Sonnet is a sister library to Haiku developed by the same team.

We will use Sonnet's module class for some nice `name_scope`-ing and later we will use the Adam optimizer implemented in Sonnet as well as some utility functions.

[4]:
```
def create_variable(path, value):
  name = '/'.join(map(str, path)).replace('~', '_')
  return tf.Variable(value, name=name)

class JaxModule(snt.Module):
  def __init__(self, params, apply_fn, name=None):
    super().__init__(name=name)
    self._params = tree.map_structure_with_path(create_variable, params)
    self._apply = jax2tf.convert(lambda p, x: apply_fn(p, None, x))
    self._apply = tf.autograph.experimental.do_not_convert(self._apply)

  def __call__(self, inputs):
    return self._apply(self._params, inputs)

net = JaxModule(params, f.apply)
[v.name for v in net.trainable_variables]
```

[4]:
```
['jax_module/mlp/~/linear_0/b:0',
 'jax_module/mlp/~/linear_0/w:0',
 'jax_module/mlp/~/linear_1/b:0',
 'jax_module/mlp/~/linear_1/w:0',
 'jax_module/mlp/~/linear_2/b:0',
 'jax_module/mlp/~/linear_2/w:0']
```

1.11.3 Train using TensorFlow

TensorFlow datasets is a great library with lots of common datasets that you might want to do research with. Here we will use it to load the MNIST handwritten digit dataset and define a simple pipeline that will randomly shuffle training images and normalize them into `[0, 1)`.

[5]:
```
import tensorflow_datasets as tfds

ds_train, ds_test = tfds.load('mnist', split=('train', 'test'),
                              shuffle_files=True, as_supervised=True)

def normalize_img(image, label):
  """Normalizes images: `uint8` -> `float32`."""
  image = tf.cast(image, tf.float32) / 255.
  return image, label

ds_train = ds_train.map(normalize_img, num_parallel_calls=tf.data.experimental.
↪AUTOTUNE)
ds_train = ds_train.cache()
ds_train = ds_train.shuffle(60000)
ds_train = ds_train.batch(100)
ds_train = ds_train.repeat()
ds_train = ds_train.prefetch(tf.data.experimental.AUTOTUNE)

ds_test = ds_test.map(normalize_img, num_parallel_calls=tf.data.experimental.AUTOTUNE)
ds_test = ds_test.batch(100)
ds_test = ds_test.cache()
ds_test = ds_test.prefetch(tf.data.experimental.AUTOTUNE)
```

In order to train our model we need a training loop that updates model parameters based on gradients for some loss. For this example we will use the Adam optimizer from Sonnet and perform a gradient update to our parameters for each mini-batch.

[6]:
```
net = JaxModule(params, f.apply)
opt = snt.optimizers.Adam(1e-3)

@tf.function(experimental_compile=True, autograph=False)
def train_step(images, labels):
  """Performs one optimizer step on a single mini-batch."""
  with tf.GradientTape() as tape:
    images = snt.flatten(images)
    logits = net(images)
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits,
                                                          labels=labels)
    loss = tf.reduce_mean(loss)
    params = tape.watched_variables()
    loss += 1e-4 * sum(map(tf.nn.l2_loss, params))

  grads = tape.gradient(loss, params)
  opt.apply(grads, params)
  return loss

for step, (images, labels) in enumerate(ds_train.take(6001)):
  loss = train_step(images, labels)
  if step % 1000 == 0:
    print(f"Step {step}: {loss.numpy()}")
```

```
Step 0: 2.309901475906372
Step 1000: 0.23313118517398834
Step 2000: 0.058662284165620804
Step 3000: 0.060427404940128326
Step 4000: 0.07748399674892426
Step 5000: 0.07069656997919083
Step 6000: 0.03870276361703873
```

To evaluate how our newly trained model performs we can use top-1 accuracy on our test set.

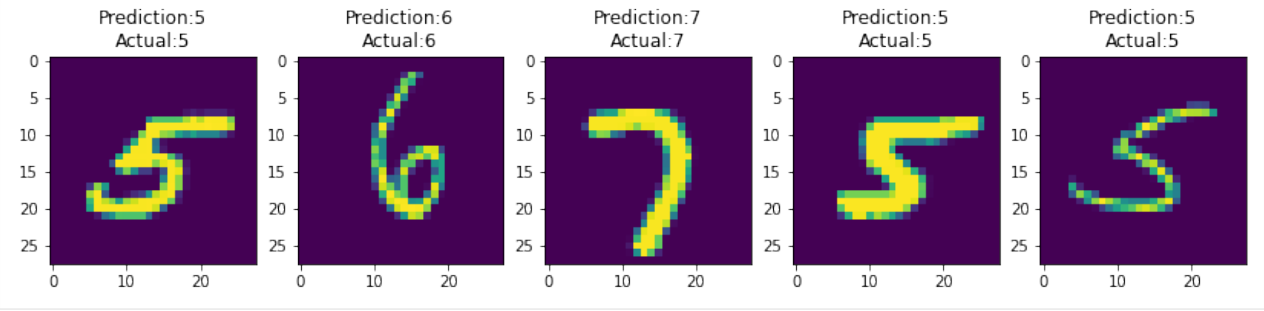
[7]:
```
def accuracy(model):
  total = 0
  correct = 0
  for images, labels in ds_test:
    predictions = tf.argmax(model(snt.flatten(images)), axis=1)
    correct += tf.math.count_nonzero(tf.equal(predictions, labels))
    total += images.shape[0]

  print("Got %d/%d (%.02f%%) correct" % (correct, total, correct / total * 100.))

accuracy(net)
```

```
Got 9805/10000 (98.05%) correct
```

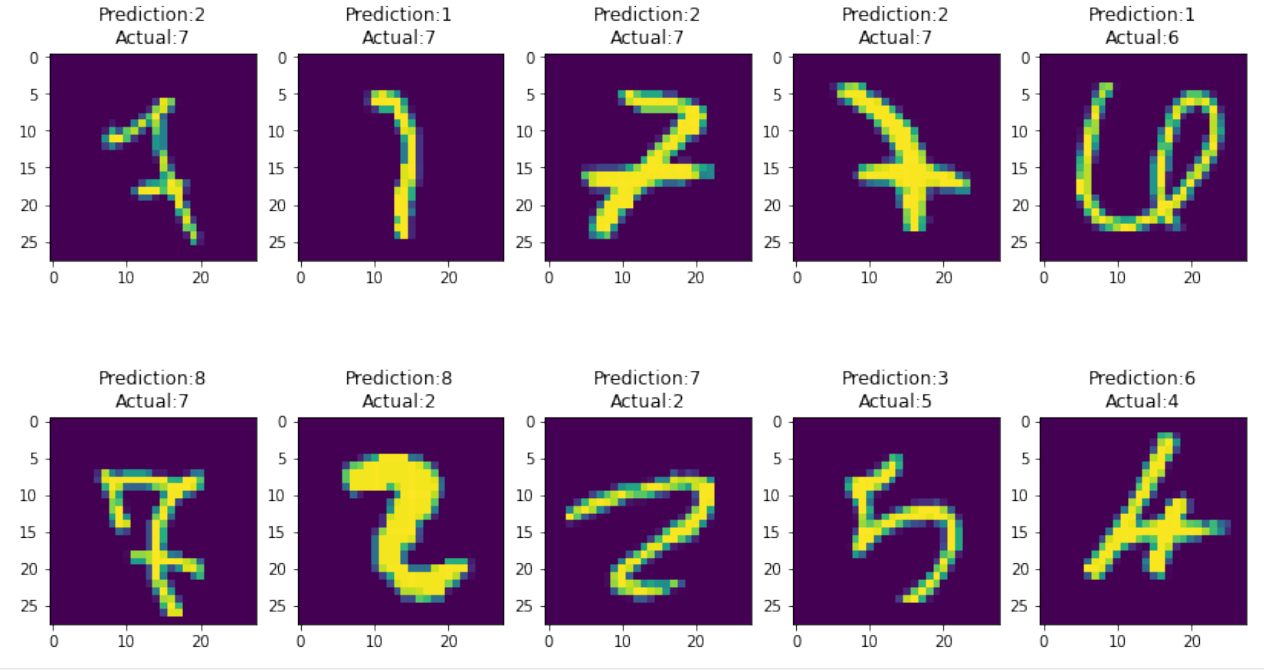
It is useful to visualize predictions the model is making against the input we are providing. This can be particularly useful where the model mispredicts the label, you can see that in some cases the handwriting is a bit dubious!

[8]:
```
import matplotlib.pyplot as plt

def sample(correct, rows, cols):
  """Utility function to show a sample of images."""
  n = 0

  f, ax = plt.subplots(rows, cols)
  if rows > 1:
    ax = tf.nest.flatten([tuple(ax[i]) for i in range(rows)])
  f.set_figwidth(14)
  f.set_figheight(4 * rows)

  for images, labels in ds_test:
    predictions = tf.argmax(net(snt.flatten(images)), axis=1)
    eq = tf.equal(predictions, labels)
    for i, x in enumerate(eq):
      if x.numpy() == correct:
        label = labels[i]
        prediction = predictions[i]
        image = tf.squeeze(images[i])

        ax[n].imshow(image)
        ax[n].set_title("Prediction:{}\nActual:{}".format(prediction, label))

        n += 1
        if n == (rows * cols):
          break

    if n == (rows * cols):
      break
```

[9]:
```
sample(correct=True, rows=1, cols=5)
```



[10]:
```
sample(correct=False, rows=2, cols=5)
```



1.11.4 Save to disk as a TensorFlow SavedModel

It is very common to take a model trained using TensorFlow and save it to disk as a "saved model". This is a language independent format that allows you to load your model code using Python, C++ or other languages supported by TensorFlow.

Saving

In order to save our model to disk, we need to define what the functions are that we want to save, and provide references to any state we want to save:

[11]:
```
@tf.function(autograph=False, input_signature=[tf.TensorSpec([100, 28 * 28])])
def forward(x):
  return net(x)

to_save = tf.Module()
```

(continues on next page)

(continued from previous page)

```
to_save.forward = forward
to_save.params = list(net.variables)
tf.saved_model.save(to_save, "/tmp/example_saved_model")
```

```
INFO:tensorflow:Assets written to: /tmp/example_saved_model/assets
```

```
INFO:tensorflow:Assets written to: /tmp/example_saved_model/assets
```

Loading

Loading a saved model is trivial, and you can see that this looks a lot like the model we saved:

[12]:
```
loaded = tf.saved_model.load("/tmp/example_saved_model")
preds = loaded.forward(tf.ones([100, 28 * 28]))
assert preds.shape == [100, 10]
assert len(loaded.params) == 6

[v.name for v in loaded.params]
```

```
WARNING:tensorflow:Importing a function (__inference_forward_26770) with ops with␣
↪custom gradients. Will likely fail if a gradient is requested.
```

```
WARNING:tensorflow:Importing a function (__inference_forward_26770) with ops with␣
↪custom gradients. Will likely fail if a gradient is requested.
```

[12]:
```
['jax_module/mlp/_/linear_0/b:0',
 'jax_module/mlp/_/linear_0/w:0',
 'jax_module/mlp/_/linear_1/b:0',
 'jax_module/mlp/_/linear_1/w:0',
 'jax_module/mlp/_/linear_2/b:0',
 'jax_module/mlp/_/linear_2/w:0']
```

Thankfully the restored model performs just as well as the model that we saved:

[13]:
```
accuracy(loaded.forward)
```

```
Got 9805/10000 (98.05%) correct
```

1.12 Build your own Haiku

In this Colab, we will build a highly-simplified version of Haiku from scratch, to give you some insight into how Haiku works.

This is an "advanced" tutorial for folks seeking a deeper understanding of Haiku's internals. It's not required to understand how to use Haiku in practice. ("advanced" is in quotes because it's not actually all that complicated, so don't be afraid!)

The implementation here is based on the design of the real Haiku library, but with most details simplified. Therefore, while this should give you a reasonably accurate sense of what's going on under-the-hood conceptually, don't rely on the details to match.

1.12.1 The Problem

We want to be able to write object-oriented classes with parameter attributes, like this,

[1]:
```
class MyModule:

  def apply(self, x):
    return self.w * x
```

and automatically transform them into pure functions, like this:

[2]:
```
def my_stateless_apply(params, x):
  return params['w'] * x
```

(However, instead of using attribute access via `self.*`, we define our own accessor function called `get_param()`. It makes it much easier to intercept its usage, which we need to collect and inject parameter values later.)

Additionally, it would be nice if this transformation also defined parameter initialisation, and automatically handled assigning parameters unique names, as managing that manually in large networks can get unwieldy. E.g., if some other module also called its parameter `w`, we'd like to automatically resolve such conflicts.

We will tackle this problem in steps.

- At the first step, we will implement a basic `transform` that converts object-oriented-style functions into pure ones.
- The next step will be to add the initialisation.
- Finally, we will handle the plumbing involved when several copies of the same module are used, or different modules use the same name for their parameters.

At that stage, we will already be able to define and train a simple neural network just like with the real Haiku.

1.12.2 The Basic Strategy

We will define a function that implements the transformation from the stateful style that uses `get_param` to a stateless function. This function will be aptly called `transform`. It will wrap a `MyModule().apply` into a function that works just like `my_stateless_apply`.

Here's how it will work. `transform(f)` will return a wrapped version of `f` that accepts an extra `params` argument. When called, it will run `f`, and every time `f` will call `get_param`, it will extract the corresponding value from `params` and return it.

[3]:
```
# Global state which holds the parameters for the transformed function.
# get_param uses this to know where to get params from.
current_params = []

def transform(f):

  def apply_f(params, *args, **kwargs):
    current_params.append(params)
    outs = f(*args, **kwargs)
    current_params.pop()
    return outs

  return apply_f
```

(continues on next page)

(continued from previous page)

```
def get_param(identifier):
  return current_params[-1][identifier]
```

let's test it:

[4]:
```
params = dict(w=5)
my_stateless_apply(params, 5)
```

[4]:
```
25
```

[5]:
```
class MyModule:
  def apply(self, x):
    return get_param('w') * x

transform(MyModule().apply)(params, 5)
```

[5]:
```
25
```

"Hold on!" you say. Isn't JAX all about not having global state? This won't possibly work in JAX! Well, let's try it with JAX:

[6]:
```
import jax
import jax.numpy as jnp

def linear(x):
  return x @ get_param('w') + get_param('b')

params = dict(w=jnp.ones((3, 5)), b=jnp.ones((5,)))
apply = transform(linear)

jax.jit(apply)(params, jnp.ones((10, 3)))
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

[6]:
```
DeviceArray([[4., 4., 4., 4., 4.],
             [4., 4., 4., 4., 4.],
             [4., 4., 4., 4., 4.],
             [4., 4., 4., 4., 4.],
             [4., 4., 4., 4., 4.],
             [4., 4., 4., 4., 4.],
             [4., 4., 4., 4., 4.],
             [4., 4., 4., 4., 4.],
             [4., 4., 4., 4., 4.],
             [4., 4., 4., 4., 4.]], dtype=float32)
```

The reason this works is that while we use global state, we're careful about how we use it. We leave the global state after a function call the same as before it, and we ensure that the outputs of a wrapped function depend only on its inputs. Thus, JAX is none the wiser – for all it cares, the transformed function is pure.

1.12.3 Adding initialization

So far, so good, but we aren't able to reuse modules, because our transform will share parameters between all copies of the same module, because they will all be named the same. Also, defining the initial state is a pain – can we automate that?

Let's tackle initialisation first. For simplicity, our parameters will always initialise using the normal distribution, but it's not hard to add the option of different initialisers.

In this new version, we `transform` one object-oriented stateful function into two pure functions: one which initialises params, and one which applies them. These correspond to running the original function in two modes: initialisation and application.

To support this, we add extra machinery (the `Frame`) to track which mode we're in, and change the behaviour of `get_param()`:

- We add a `shape` argument, which tells us what shape the param should be if initialising.
- If initialising, `get_param()` will create the param of the correct shape and add it to the current params in the `Frame` before returning.

Thus, `get_param()` generates the initial values just in time for them to get used in a call of the stateful function.

[7]:
```
from typing import NamedTuple, Dict, Callable
import numpy as np
```

[8]:
```
# Since we're tracking more than just the current params,
# we introduce the concept of a frame as the object that holds
# state during a transformed execution.
frame_stack = []

class Frame(NamedTuple):
  """Tracks what's going on during a call of a transformed function."""
  params: Dict[str, jnp.ndarray]
  is_initialising: bool = False

def current_frame():
  return frame_stack[-1]


class Transformed(NamedTuple):
  init: Callable
  apply: Callable


def transform(f) -> Transformed:

  def init_f(*args, **kwargs):
    frame_stack.append(Frame({}, is_initialising=True))
    f(*args, **kwargs)
    frame = frame_stack.pop()
    return frame.params

  def apply_f(params, *args, **kwargs):
    frame_stack.append(Frame(params))
    outs = f(*args, **kwargs)
    frame_stack.pop()
    return outs
```

(continues on next page)

(continued from previous page)

```
  return Transformed(init_f, apply_f)

def get_param(identifier, shape):
  if current_frame().is_initialising:
    current_frame().params[identifier] = np.random.normal(size=shape)

  return current_frame().params[identifier]
```

Let's test it by implementing a Linear module:

[9]:
```
# Make printing parameters a little more readable
def parameter_shapes(params):
  return jax.tree_map(lambda p: p.shape, params)


class Linear:

  def __init__(self, width):
    self._width = width

  def __call__(self, x):
    w = get_param('w', shape=(x.shape[-1], self._width))
    b = get_param('b', shape=(self._width,))
    return x @ w + b

init, apply = transform(Linear(4))

data = jnp.ones((2, 3))

params = init(data)
parameter_shapes(params)
```

[9]:
```
{'b': (4,), 'w': (3, 4)}
```

[10]:
```
apply(params, data)
```

[10]:
```
DeviceArray([[-1.0345883,  0.3280404, -2.4382973,  0.5717376],
             [-1.0345883,  0.3280404, -2.4382973,  0.5717376]],
 ↪dtype=float32)
```

1.12.4 Adding unique parameter names: our finished mini-Haiku

Alright! Time to tackle nesting modules, and our prototype will be done.

For this, we need to give each parameter an unambiguous name. Here, we will use a scheme that's somewhat different and incompatible with the real Haiku, but which is simpler. The idea is to record the names of the functions being called, and to assign each parameter a unique identifier based on its location in the call stack.

For this, we will define a Module class. Each module will have a unique identifier based on the class name and the number of instances of the module created so far. (Real Haiku allows to customise these names, but we ignore that for simplicity)

We will also define a decorator for Module methods, called `module_method`, which will tell us when the wrapped function is called, allowing us to track the current parameter scope. Real haiku uses metaclasses to automatically wrap all methods on a `Module`, but for simplicity we do this manually.

[11]:
```
import dataclasses
import collections

@dataclasses.dataclass
class Frame:
  """Tracks what's going on during a call of a transformed function."""
  params: Dict[str, jnp.ndarray]
  is_initialising: bool = False

  # Keeps track of how many modules of each class have been created so far.
  # Used to assign new modules unique names.
  module_counts: Dict[str, int] = dataclasses.field(
      default_factory=lambda: collections.defaultdict(lambda: 0))

  # Keeps track of the entire path to the current module method call.
  # Module methods, when called, will add themselves to this stack.
  # Used to give each parameter a unique name corresponding to the
  # method scope it is in.
  call_stack: list = dataclasses.field(default_factory=list)

  def create_param_path(self, identifier) -> str:
    """Creates a unique path for this param."""
    return '/'.join(['~'] + self.call_stack + [identifier])

  def create_unique_module_name(self, module_name: str) -> str:
    """Assigns a unique name to the module by appending its number to its name."""
    number = self.module_counts[module_name]
    self.module_counts[module_name] += 1
    return f"{module_name}_{number}"

frame_stack = []

def current_frame():
  return frame_stack[-1]


class Module:
  def __init__(self):
    # Assign a unique (for the current `transform` call)
    # name to this instance of the module.
    self._unique_name = current_frame().create_unique_module_name(
        self.__class__.__name__)


def module_method(f):
  """A decorator for Module methods."""
  # In the real Haiku, this doesn't face the user but is applied by a metaclass.

  def wrapped(self, *args, **kwargs):
    """A version of f that lets the frame know it's being called."""
    # Self is the instance to which this method is attached.
    module_name = self._unique_name
    call_stack = current_frame().call_stack
    call_stack.append(module_name)
    call_stack.append(f.__name__)
    outs = f(self, *args, **kwargs)
    assert call_stack.pop() == f.__name__
```

(continues on next page)

(continued from previous page)

```
    assert call_stack.pop() == module_name
    return outs

  return wrapped


def get_param(identifier, shape):
  frame = current_frame()
  param_path = frame.create_param_path(identifier)

  if frame.is_initialising:
    frame.params[param_path] = np.random.normal(size=shape)

  return frame.params[param_path]


class Linear(Module):

  def __init__(self, width):
    super().__init__()
    self._width = width

  @module_method  # Again, this decorator is behind-the-scenes in real Haiku.
  def __call__(self, x):
    w = get_param('w', shape=(x.shape[-1], self._width))
    b = get_param('b', shape=(self._width,))
    return x @ w + b
```

At this stage, we have replicated some core Haiku functionality, but we still don't have: * control over initialisation * rng handling * state handling (though conceptually that's analogous to parameter handling) * any kind of validation and error handling * freezing parameters once they're created * thread-safety * JAX transformations inside a transform (e.g. hk.remat) * JAX control flow inside transforms (e.g. hk.scan) * last but not least, documentation :)

and lots more. However, the basics work, so we can take our mini-Haiku for a ride:

[12]:
```
init, apply = transform(lambda x: Linear(4)(x))

params = init(data)
parameter_shapes(params)
```

[12]:
```
{'~/Linear_0/__call__/b': (4,), '~/Linear_0/__call__/w': (3, 4)}
```

[13]:
```
apply(params, data)
```

[13]:
```
DeviceArray([[-1.1969297,  1.3215988,  5.175427 , -1.9018829],
             [-1.1969297,  1.3215988,  5.175427 , -1.9018829]],
 ↪dtype=float32)
```

Different Modules in a function call all have separate parameters:

[14]:
```
class MLP(Module):

  def __init__(self, widths):
    super().__init__()
    self._widths = widths

  @module_method
```

(continues on next page)

(continued from previous page)

```
  def __call__(self, x):
    for w in self._widths:
      out = Linear(w)(x)
      x = jax.nn.sigmoid(out)
    return out
```

[15]:
```
init, apply = transform(lambda x: MLP([3, 5])(x))
parameter_shapes(init(data))
```

[15]:
```
{'~/MLP_0/__call__/Linear_0/__call__/b': (3,),
 '~/MLP_0/__call__/Linear_0/__call__/w': (3, 3),
 '~/MLP_0/__call__/Linear_1/__call__/b': (5,),
 '~/MLP_0/__call__/Linear_1/__call__/w': (3, 5)}
```

While the same module called in different places reuses parameters:

[16]:
```
class ParameterReuseTest(Module):

  @module_method
  def __call__(self, x):
    f = Linear(x.shape[-1])

    x = f(x)
    x = jax.nn.relu(x)
    return f(x)

init, forward = transform(lambda x: ParameterReuseTest()(x))
parameter_shapes(init(data))
```

[16]:
```
{'~/ParameterReuseTest_0/__call__/Linear_0/__call__/b': (3,),
 '~/ParameterReuseTest_0/__call__/Linear_0/__call__/w': (3, 3)}
```

1.12.5 Example training loop

[17]:
```
import matplotlib.pyplot as plt
```

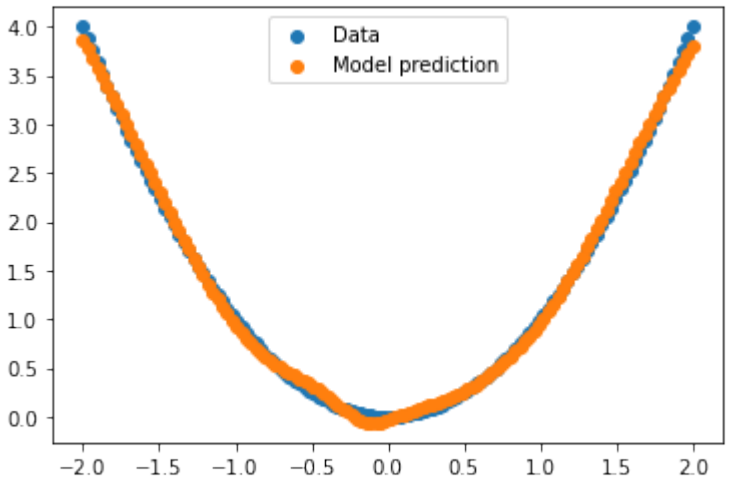
[18]:
```
# Data: a quadratic curve.
xs = np.linspace(-2., 2., num=128)[:, None]  # Generate array of shape (128, 1).
ys = xs ** 2

# Model
def mlp(x):
  return MLP([128, 128, 1])(x)

init, forward = transform(mlp)
params = init(xs)
parameter_shapes(params)
```

[18]:
```
{'~/MLP_0/__call__/Linear_0/__call__/b': (128,),
 '~/MLP_0/__call__/Linear_0/__call__/w': (1, 128),
 '~/MLP_0/__call__/Linear_1/__call__/b': (128,),
 '~/MLP_0/__call__/Linear_1/__call__/w': (128, 128),
 '~/MLP_0/__call__/Linear_2/__call__/b': (1,),
 '~/MLP_0/__call__/Linear_2/__call__/w': (128, 1)}
```

[19]:
```
# Loss function and update function
def loss_fn(params, x, y):
  return jnp.mean((forward(params, x) - y) ** 2)

LEARNING_RATE = 0.003

@jax.jit
def update(params, x, y):
  grads = jax.grad(loss_fn)(params, x, y)
  return jax.tree_multimap(
      lambda p, g: p - LEARNING_RATE * g, params, grads
  )
```

[20]:
```
for _ in range(5000):
  params = update(params, xs, ys)
```

[21]:
```
plt.scatter(xs, ys, label='Data')
plt.scatter(xs, forward(params, xs), label='Model prediction')
plt.legend()
plt.show()
```



[ ]:
```
```

1.13 Visualization

Haiku supports two ways to visualize your program. To use these you need to install two additional dependencies:

[1]:
```
!pip install dm-tree graphviz
```

```
Requirement already satisfied: dm-tree in /tmp/haiku-docs-env/lib/python3.8/site-
↪packages (0.1.5)
Requirement already satisfied: graphviz in /tmp/haiku-docs-env/lib/python3.8/site-
↪packages (0.16)
Requirement already satisfied: six>=1.12.0 in /tmp/haiku-docs-env/lib/python3.8/site-
↪packages (from dm-tree) (1.15.0)
```

[2]:
```
import jax
import jax.numpy as jnp
import haiku as hk
```

1.13.1 Tabulate

Like many neural network libraries, Haiku supports showing a summary of the execution of your program as a table of modules. Haiku's approach is to trace the execution of your program and to produce a table of (interesting) module method calls.

For example, the interesting methods for a 3 layer MLP would be `MLP.__call__` which in turns calls `Linear.__call__` on three inner modules. For each module method we show columns relating to the input/output size of arrays, as well as details of the modules parameters and where it fits in the module hierarchy.

[3]:
```
def f(x):
  return hk.nets.MLP([300, 100, 10])(x)

f = hk.transform(f)
x = jnp.ones([8, 28 * 28])

print(hk.experimental.tabulate(f)(x))
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

```
+-------------------------+-----------------------------------------+----------------
↪-+------------+------------+---------------+---------------+
| Module                  | Config                                  | Module params␣
↪ | Input      | Output     |   Param count |   Param bytes |
+=========================+=========================================+================+============
| mlp (MLP)               | MLP(output_sizes=[300, 100, 10])        |                ␣
↪ | f32[8,784] | f32[8,10]  |       266,610 |       1.07 MB |
+-------------------------+-----------------------------------------+----------------
↪-+------------+------------+---------------+---------------+
| mlp/~/linear_0 (Linear) | Linear(output_size=300, name='linear_0') | w: f32[784,
↪300] | f32[8,784] | f32[8,300] |       235,500 |     942.00 KB |
|   └ mlp (MLP)           |                                         | b: f32[300]    ␣
↪ |            |            |               |               |
+-------------------------+-----------------------------------------+----------------
↪-+------------+------------+---------------+---------------+
| mlp/~/linear_1 (Linear) | Linear(output_size=100, name='linear_1') | w: f32[300,
↪100] | f32[8,300] | f32[8,100] |        30,100 |     120.40 KB |
|   └ mlp (MLP)           |                                         | b: f32[100]    ␣
↪ |            |            |               |               |
+-------------------------+-----------------------------------------+----------------
↪-+------------+------------+---------------+---------------+
| mlp/~/linear_2 (Linear) | Linear(output_size=10, name='linear_2')  | w: f32[100,10]␣
↪ | f32[8,100] | f32[8,10]  |         1,010 |       4.04 KB |
|   └ mlp (MLP)           |                                         | b: f32[10]     ␣
↪ |            |            |               |               |
+-------------------------+-----------------------------------------+----------------
↪-+------------+------------+---------------+---------------+
```

We also offer access to the raw data used to build this table if you want to create your own summary:

[4]:
```
for method_invocation in hk.experimental.eval_summary(f)(x):
  print(method_invocation)
```

```
MethodInvocation(module_details=ModuleDetails(module=MLP(output_sizes=[300, 100, 10]),
↪ method_name='__call__', params={'mlp/~/linear_0/b': f32[300], 'mlp/~/linear_0/w':␣
↪f32[784,300], 'mlp/~/linear_1/b': f32[100], 'mlp/~/linear_1/w': f32[300,100], 'mlp/~
↪/linear_2/b': f32[10], 'mlp/~/linear_2/w': f32[100,10]}), args_spec=(f32[8,784],),␣
↪kwargs_spec={}, output_spec=f32[8,10], context=MethodContext(module=MLP(output_
↪sizes=[300, 100, 10]), method_name='__call__', orig_method=functools.partial(
↪<function MLP.__call__ at 0x7f6d1b193c10>, MLP(output_sizes=[300, 100, 10]))), call_
↪stack=(ModuleDetails(module=MLP(output_sizes=[300, 100, 10]), method_name='__call__
↪', params={'mlp/~/linear_0/b': f32[300], 'mlp/~/linear_0/w': f32[784,300], 'mlp/~/
↪linear_1/b': f32[100], 'mlp/~/linear_1/w': f32[300,100], 'mlp/~/linear_2/b':␣
↪f32[10], 'mlp/~/linear_2/w': f32[100,10]}),))
MethodInvocation(module_details=ModuleDetails(module=Linear(output_size=300, name=
↪'linear_0'), method_name='__call__', params={'mlp/~/linear_0/b': f32[300], 'mlp/~/
↪linear_0/w': f32[784,300]}), args_spec=(f32[8,784],), kwargs_spec={}, output_
↪spec=f32[8,300], context=MethodContext(module=Linear(output_size=300, name='linear_0
↪'), method_name='__call__', orig_method=functools.partial(<function Linear.__call__␣
↪at 0x7f6d1b1933a0>, Linear(output_size=300, name='linear_0'))), call_
↪stack=(ModuleDetails(module=Linear(output_size=300, name='linear_0'), method_name='_
↪_call__', params={'mlp/~/linear_0/b': f32[300], 'mlp/~/linear_0/w': f32[784,300]}),␣
↪ModuleDetails(module=MLP(output_sizes=[300, 100, 10]), method_name='__call__',␣
↪params={'mlp/~/linear_0/b': f32[300], 'mlp/~/linear_0/w': f32[784,300], 'mlp/~/
↪linear_1/b': f32[100], 'mlp/~/linear_1/w': f32[300,100], 'mlp/~/linear_2/b':␣
↪f32[10], 'mlp/~/linear_2/w': f32[100,10]})))
MethodInvocation(module_details=ModuleDetails(module=Linear(output_size=100, name=
↪'linear_1'), method_name='__call__', params={'mlp/~/linear_1/b': f32[100], 'mlp/~/
↪linear_1/w': f32[300,100]}), args_spec=(f32[8,300],), kwargs_spec={}, output_
↪spec=f32[8,100], context=MethodContext(module=Linear(output_size=100, name='linear_1
↪'), method_name='__call__', orig_method=functools.partial(<function Linear.__call__␣
↪at 0x7f6d1b1933a0>, Linear(output_size=100, name='linear_1'))), call_
↪stack=(ModuleDetails(module=Linear(output_size=100, name='linear_1'), method_name='_
↪_call__', params={'mlp/~/linear_1/b': f32[100], 'mlp/~/linear_1/w': f32[300,100]}),␣
↪ModuleDetails(module=MLP(output_sizes=[300, 100, 10]), method_name='__call__',␣
↪params={'mlp/~/linear_0/b': f32[300], 'mlp/~/linear_0/w': f32[784,300], 'mlp/~/
↪linear_1/b': f32[100], 'mlp/~/linear_1/w': f32[300,100], 'mlp/~/linear_2/b':␣
↪f32[10], 'mlp/~/linear_2/w': f32[100,10]})))
MethodInvocation(module_details=ModuleDetails(module=Linear(output_size=10, name=
↪'linear_2'), method_name='__call__', params={'mlp/~/linear_2/b': f32[10], 'mlp/~/
↪linear_2/w': f32[100,10]}), args_spec=(f32[8,100],), kwargs_spec={}, output_
↪spec=f32[8,10], context=MethodContext(module=Linear(output_size=10, name='linear_2
↪'), method_name='__call__', orig_method=functools.partial(<function Linear.__call__␣
↪at 0x7f6d1b1933a0>, Linear(output_size=10, name='linear_2'))), call_
↪stack=(ModuleDetails(module=Linear(output_size=10, name='linear_2'), method_name='__
↪call__', params={'mlp/~/linear_2/b': f32[10], 'mlp/~/linear_2/w': f32[100,10]}),␣
↪ModuleDetails(module=MLP(output_sizes=[300, 100, 10]), method_name='__call__',␣
↪params={'mlp/~/linear_0/b': f32[300], 'mlp/~/linear_0/w': f32[784,300], 'mlp/~/
↪linear_1/b': f32[100], 'mlp/~/linear_1/w': f32[300,100], 'mlp/~/linear_2/b':␣
↪f32[10], 'mlp/~/linear_2/w': f32[100,10]})))
```

1.13.2 Graphviz (aka. `to_dot`)

Haiku supports rendering your program as a graphviz graph. We show all of the JAX primitives involved in a given computation clustered by Haiku module.

Lets start by visualizing a simple program not using Haiku modules:

[5]:
```
def f(a):
  b = jnp.sin(a)
  c = jnp.cos(b)
  d = b + c
  e = a + d
  return e

x = jnp.ones([1])
dot = hk.experimental.to_dot(f)(x)

import graphviz
graphviz.Source(dot)
```

[5]:

The visualization above shows our program as a simple dataflow graph of our single input highligted in orange (`args[0]`) being passed through some operations and producing a result (highlighted in blue). Primitive operations (e.g. `sin`, `cos` and `add`) are highlighted in yellow.

Actual Haiku programs are often far more complex, involving many modules and many more primitive operations. For these programs it is often useful to visualize the program on a module by module basis.

`to_dot` offers this by clustering operations by their module. Again it is probably simplest to see an example:

[6]:
```
def f(x):
  return hk.nets.MLP([300, 100, 10])(x)

f = hk.transform(f)

rng = jax.random.PRNGKey(42)
x = jnp.ones([8, 28 * 28])
params = f.init(rng, x)

dot = hk.experimental.to_dot(f.apply)(params, None, x)
graphviz.Source(dot)
```

[6]:

1.14 Training a subset of parameters

Sometimes when training a neural network it is useful to hold some parameters of your network fixed while updating others. This is commonly referred to as "non-trainable variables" or "layer freezing".

In typical neural network training, parameters are updated by computing gradients and computing an update via an optimizer such as SGD or ADAM. Updates are then applied to parameters and the process repeats until you have converged.

As such to implement "layer freezing" or "non-trainable variables" in JAX, we simply need to not compute and apply updates for certain parameters of our network.

In JAX computing gradients and applying updates to parameters are fully in your control as a user. JAX's autodiff mechanics allow you to compute gradients wrt any positional argument to a function.

In Haiku (and other NN libraries) it is typical to pass your parameters as a single positional argument to your function (e.g. `grads = jax.grad(loss_fn)(params, ...)`).

To support taking gradients wrt a subset of parameters, we need to allow users to split their parameters into two positional arguments, such that they can compute gradients wrt a subset of their parameters (e.g. `trainable_params_grads = jax.grad(loss_fn)(trainable_params, non_trainable_params, ...)`).

Haiku ships with some utilities that make it easier to manipulate the parameters dictionary in order to split into these trainable/non-trainable sets as well as to recombine your parameters into a single dictionary.

We will walk through how to do this with a simple MLP and teach it the identity function.

[1]:
```
import haiku as hk
import jax
import jax.numpy as jnp
import numpy as np
```

The forward pass of our network is a standard MLP. We want to adjust the parameters of this MLP such that it computes the identity. That is `forward([[1.0], [2.0], [3.0]) == [1, 2, 3]`. We will do this for a maximum of 10 numbers.

Our network starts randomly initialised so the results initially do not make much sense:

[2]:
```
num_classes = 10

def f(x):
  return hk.nets.MLP([300, 100, num_classes])(x)

f = hk.transform(f)

def test(params, num_classes=num_classes):
  x = np.arange(num_classes).reshape([num_classes, 1]).astype(np.float32)
  y = jnp.argmax(f.apply(params, None, x), axis=-1)
  for x, y in zip(x, y):
    print(x, "->", y)

rng = jax.random.PRNGKey(42)
x = np.zeros([num_classes, 1])
params = f.init(rng, x)

print("before training")
test(params)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
/tmp/haiku-docs-env/lib/python3.8/site-packages/jax/_src/lax/lax.py:6271: UserWarning:
↪ Explicitly requested dtype float64 requested in zeros is not available, and will␣
↪be truncated to dtype float32. To enable more dtypes, set the jax_enable_x64␣
↪configuration option or the JAX_ENABLE_X64 shell environment variable. See https://
↪github.com/google/jax#current-gotchas for more.
  warnings.warn(msg.format(dtype, fun_name , truncated_dtype))
```

```
before training
[0.] -> 0
[1.] -> 3
[2.] -> 3
[3.] -> 3
[4.] -> 3
[5.] -> 3
```

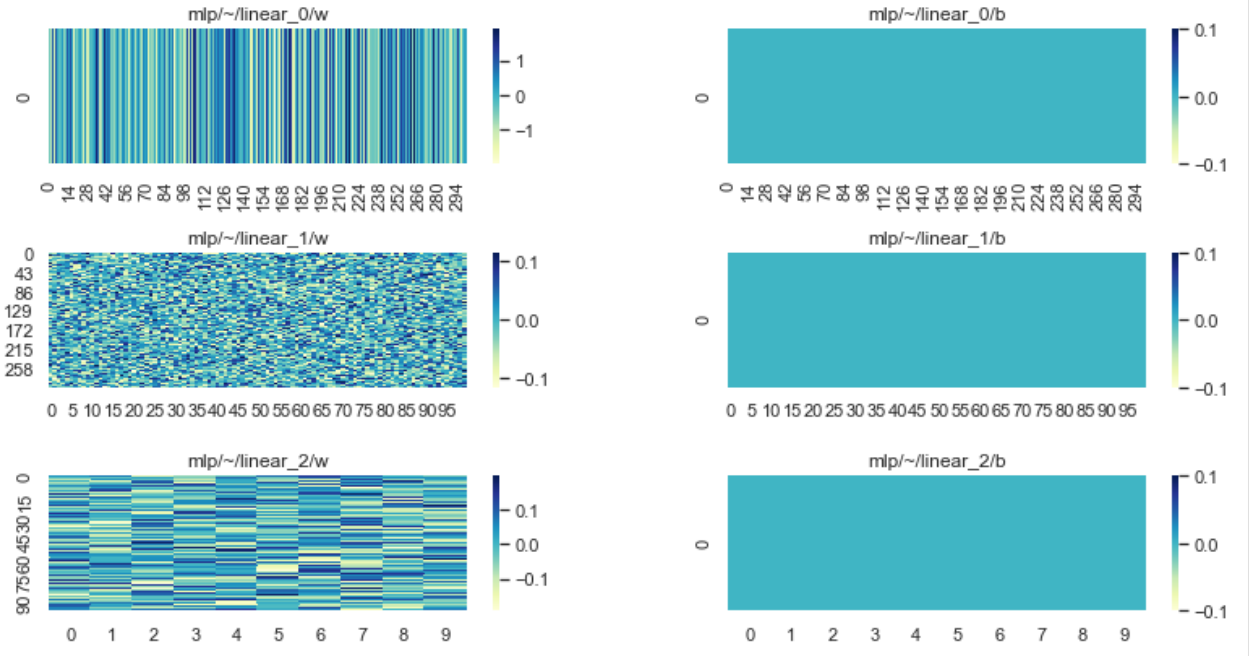
(continues on next page)

(continued from previous page)

```
[6.] -> 3
[7.] -> 3
[8.] -> 3
[9.] -> 3
```

It is useful to visualise our paramerters so we can compare to their final state:

[3]:
```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_theme()

def plot_params(params):
  fig, axs = plt.subplots(ncols=2, nrows=3)
  fig.tight_layout()
  fig.set_figwidth(12)
  fig.set_figheight(6)
  for row, module in enumerate(sorted(params)):
    ax = axs[row][0]
    sns.heatmap(params[module]["w"], cmap="YlGnBu", ax=ax)
    ax.title.set_text(f"{module}/w")

    ax = axs[row][1]
    b = np.expand_dims(params[module]["b"], axis=0)
    sns.heatmap(b, cmap="YlGnBu", ax=ax)
    ax.title.set_text(f"{module}/b")

plot_params(params)
```



To train our network we'll create some simple synthetic batches of data:

[4]:
```
def dataset(*, batch_size, num_records):
  for _ in range(num_records):
    y = np.arange(num_classes)
```

(continues on next page)

(continued from previous page)

```
    y = np.random.permutation(y)[:batch_size]
    x = y.reshape([batch_size, 1]).astype(np.float32)
    yield x, y

for x, y in dataset(batch_size=4, num_records=5):
  print("x :=", x.tolist(), "y :=", y)
```

```
x := [[0.0], [8.0], [7.0], [1.0]] y := [0 8 7 1]
x := [[6.0], [7.0], [0.0], [9.0]] y := [6 7 0 9]
x := [[4.0], [0.0], [9.0], [6.0]] y := [4 0 9 6]
x := [[0.0], [4.0], [6.0], [5.0]] y := [0 4 6 5]
x := [[4.0], [3.0], [0.0], [5.0]] y := [4 3 0 5]
```

Now for the interesting part. Lets pretend that we only want to update the parameters of the first and last layer of our MLP.

The simplest and most efficient way to do this is to partition our parameters into two groups, "trainable" and "non trainable". Haiku provides a convenience function for doing this in `hk.data_structures.partition(..)`:

[5]:
```
# Partition our params into trainable and non trainable explicitly.
trainable_params, non_trainable_params = hk.data_structures.partition(
    lambda m, n, p: m != "mlp/~/linear_1", params)

print("trainable:", list(trainable_params))
print("non_trainable:", list(non_trainable_params))
```

```
trainable: ['mlp/~/linear_0', 'mlp/~/linear_2']
non_trainable: ['mlp/~/linear_1']
```

The reason we split our parameters is that this allows us to pass them to our loss function as separate positional arguments.

In JAX gradients are taken with respect to positional arguments. By splitting our parameters into two groups we can take gradients with respect to just one of the positional arguments. We can then use those gradients to update a subset of our parameters.

The last piece of the puzzle is that we need to combine our "trainable" and "non trainable" parameters together before calling our apply function. Again Haiku provides `hk.data_structures.merge(..)` to make this easy:

[6]:
```
def loss_fn(trainable_params, non_trainable_params, images, labels):
  # NOTE: We need to combine trainable and non trainable before calling apply.
  params = hk.data_structures.merge(trainable_params, non_trainable_params)

  # NOTE: From here on this is a standard softmax cross entropy loss.
  logits = f.apply(params, None, images)
  labels = jax.nn.one_hot(labels, logits.shape[-1])
  return -jnp.sum(labels * jax.nn.log_softmax(logits)) / labels.shape[0]

def sgd_step(params, grads, *, lr):
  return jax.tree_multimap(lambda p, g: p - g * lr, params, grads)

def train_step(trainable_params, non_trainable_params, x, y):
  # NOTE: We will only compute gradients wrt `trainable_params`.
  trainable_params_grads = jax.grad(loss_fn)(trainable_params,
                                             non_trainable_params, x, y)

  # NOTE: We are only updating `trainable_params`.
  trainable_params = sgd_step(trainable_params, trainable_params_grads, lr=0.1)
```

(continues on next page)

(continued from previous page)

```
  return trainable_params

train_step = jax.jit(train_step)

for x, y in dataset(batch_size=num_classes, num_records=10000):
  # NOTE: In our training loop only our trainable parameters are updated.
  trainable_params = train_step(trainable_params, non_trainable_params, x, y)
```

We can see that even though we only trained a subset of our parameters, our NN is able to learn this simple function:

[7]:
```
# Merge params again for inference.
params = hk.data_structures.merge(trainable_params, non_trainable_params)

print("after training")
test(params)
```

```
after training
[0.] -> 0
[1.] -> 1
[2.] -> 2
[3.] -> 3
[4.] -> 4
[5.] -> 5
[6.] -> 6
[7.] -> 7
[8.] -> 8
[9.] -> 9
```

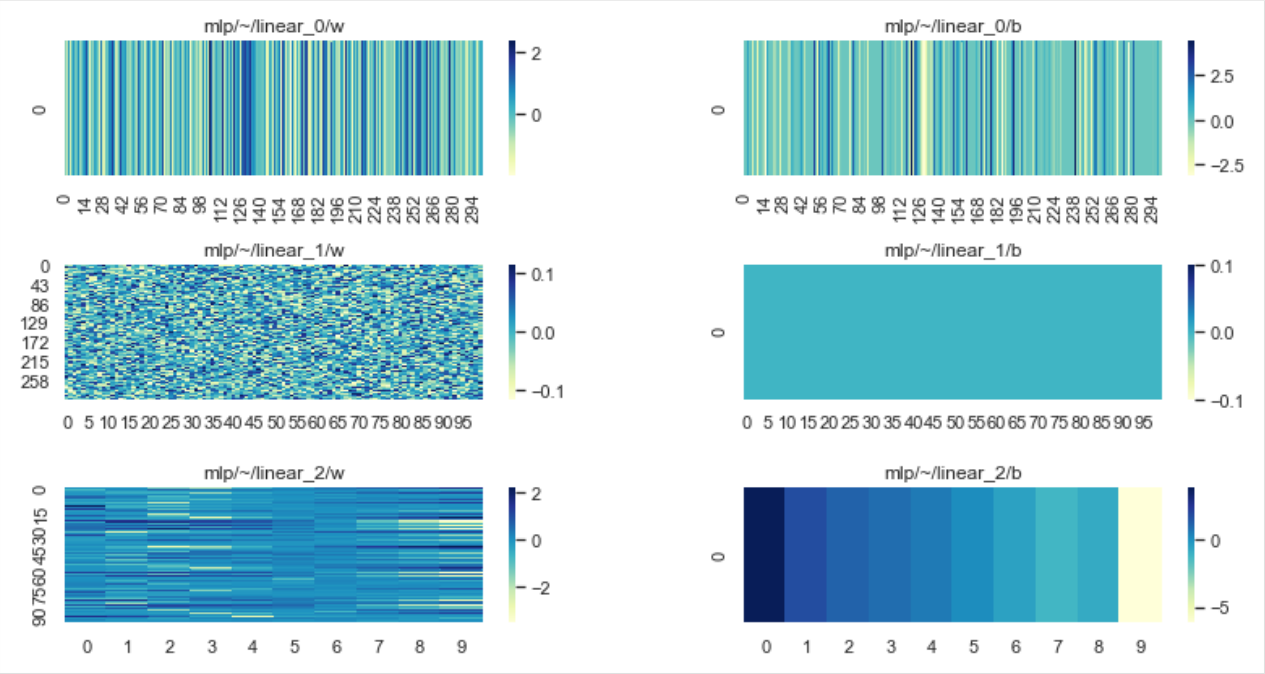
Of course it is not smart enough to generalize to out of distribution inputs:

[8]:
```
test(params, num_classes=num_classes+10)
```

```
[0.] -> 0
[1.] -> 1
[2.] -> 2
[3.] -> 3
[4.] -> 4
[5.] -> 5
[6.] -> 6
[7.] -> 7
[8.] -> 8
[9.] -> 9
[10.] -> 9
[11.] -> 9
[12.] -> 9
[13.] -> 9
[14.] -> 9
[15.] -> 9
[16.] -> 9
[17.] -> 9
[18.] -> 9
[19.] -> 9
```

Looking at our parameters we can see that `linear_1` is still in its initial state (randomly initialised weight matrix and zero initialized bias):

[9]:
```
plot_params(params)
```


mlp/~/linear_0/w
mlp/~/linear_0/b
mlp/~/linear_1/w
mlp/~/linear_1/b
mlp/~/linear_2/w
mlp/~/linear_2/b

CHAPTER

TWO

KNOWN ISSUES

Warning: Using JAX transformations like `jax.jit()` and `jax.remat()` inside of Haiku networks can lead to hard to interpret tracing errors and potentially silently wrong results. Read *Limitations of Nesting JAX Functions and Haiku Modules* to find out how to work around these issues.

CHAPTER

THREE

CONTRIBUTE

- Issue tracker: https://github.com/deepmind/dm-haiku/issues
- Source code: https://github.com/deepmind/dm-haiku/tree/main

CHAPTER

FOUR

SUPPORT

If you are having issues, please let us know by filing an issue on our issue tracker.

CHAPTER

FIVE

LICENSE

Haiku is licensed under the Apache 2.0 License.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

[1] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014. URL: https://arxiv.org/abs/1409.2329.

[2] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, 2342–2350. 2015.

[3] SHI Xingjian, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: a machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems*, 802–810. 2015.

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

N

O

P

Q

R

S

T

U

V

W