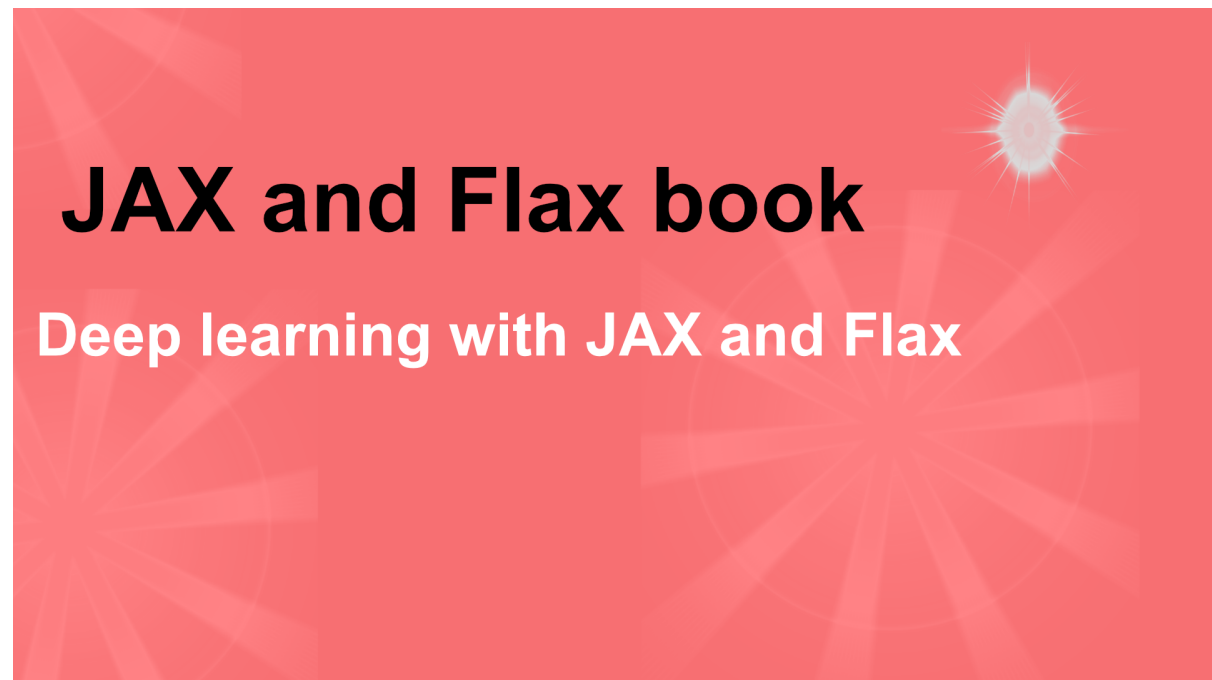


# JAX and Flax ebook sample



## JAX (What it is and how to use it in Python)

JAX is a Python library offering high performance in machine learning with XLA and Just In Time (JIT) compilation. Its API is similar to NumPy's with a few differences. JAX ships with functionalities that aim to improve and increase speed in machine learning research. These functionalities include:

- Automatic differentiation
- Vectorization
- JIT compilation

This article will cover these functionalities and other JAX concepts. Let's get started.

## What is XLA?

XLA (Accelerated Linear Algebra) is a linear algebra compiler for accelerating machine learning models. It leads to an increase in the speed of model execution and reduced memory usage. XLA programs can be generated by JAX, PyTorch, Julia, and NX.

## Installing JAX

JAX can be installed from the Python Package Index using:

```
pip install jax
```

JAX is pre-installed on Google Colab. See the link below for other installation options.

# Setting up TPUs on Google Colab

You need to set up JAX to use TPUs on Colab. That is done by executing the following code. Ensure that you have changed the runtime to TPU by going to Runtime-> Change Runtime Type. If no accelerator is available, JAX will use the CPU.

```
import jax.tools.colab_tpu
jax.tools.colab_tpu.setup_tpu()
jax.devices()
```

```
✓ 47s ▶ import jax
import jax.tools.colab_tpu
jax.tools.colab_tpu.setup_tpu()
jax.devices()

[TpuDevice(id=0, process_index=0, coords=(0,0,0), core_on_chip=0),
 TpuDevice(id=1, process_index=0, coords=(0,0,0), core_on_chip=1),
 TpuDevice(id=2, process_index=0, coords=(1,0,0), core_on_chip=0),
 TpuDevice(id=3, process_index=0, coords=(1,0,0), core_on_chip=1),
 TpuDevice(id=4, process_index=0, coords=(0,1,0), core_on_chip=0),
 TpuDevice(id=5, process_index=0, coords=(0,1,0), core_on_chip=1),
 TpuDevice(id=6, process_index=0, coords=(1,1,0), core_on_chip=0),
 TpuDevice(id=7, process_index=0, coords=(1,1,0), core_on_chip=1)]
```

# Data types in JAX

The data types in NumPy are similar to those in JAX arrays. For instance, here is how you can create `float` and `int` data in JAX.

```
import jax.numpy as jnp
x = jnp.float32(1.25844)
x = jnp.int32(45.25844)
```

When you check the type of the data, you will see that it's a `DeviceArray`.

```
[3] x = jnp.float32(1.25844)
0s

x
0s
DeviceArray(1.25844, dtype=float32)

[5] type(x)
0s
jax._src.device_array._DeviceArray

[6] x = jnp.int32(45.25844)
0s

[7] x
0s
DeviceArray(45, dtype=int32)
```

`DeviceArray` in JAX is the equivalent of `numpy.ndarray` in NumPy.

`jax.numpy` provides an interface similar to NumPy's. However, JAX also provides `jax.lax` a low-level API that is more powerful and stricter. For example, with `jax.numpy` you can add numbers that have mixed types but `jax.lax` will not allow this.

# Ways to create JAX arrays

You can create JAX arrays like you would in NumPy. For example, can use:

- `arange`
- `linspace`
- Python lists.
- `ones` .
- `zeros` .
- `identity` .

```
jnp.arange(10)
jnp.arange(0,10)
scores = [50,60,70,30,25,70]
scores_array = jnp.array(scores)
jnp.zeros(5)
jnp.ones(5)
```

```
jnp.eye(5)
jnp.identity(5)
```

```
✓ [8] ▶ jnp.arange(10)
      DeviceArray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int32)

✓ [9] jnp.arange(0,10)
      DeviceArray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int32)

✓ [10] jnp.arange(0,10,2)
      DeviceArray([0, 2, 4, 6, 8], dtype=int32)

▼ Covert Python List to a NumPy Array

✓ [11] scores = [50,60,70,30,25,70]

✓ [16] scores_array = jnp.array(scores)

✓ [17] scores_array
      DeviceArray([50, 60, 70, 30, 25, 70], dtype=int32)
```

# Generating random numbers with JAX

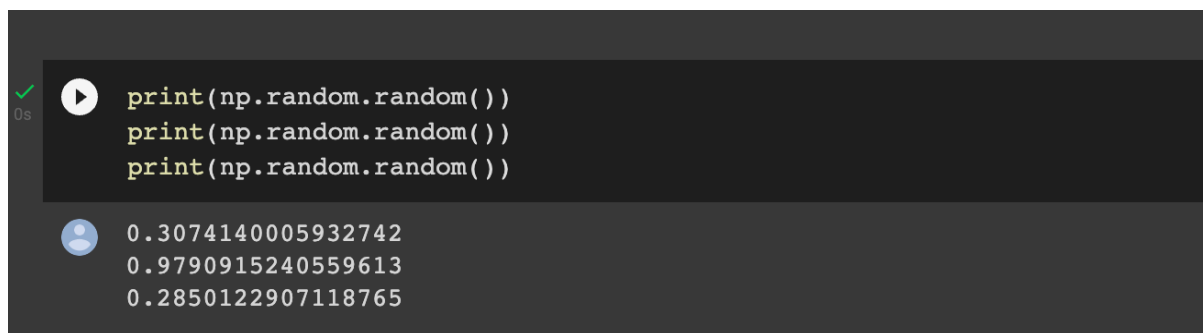
Random number generation is one main difference between JAX and NumPy. JAX is meant to be used with functional programs. JAX expects these functions to be pure. A **pure function** has no side effects and expects the output to only come from its inputs. JAX transformation functions expect pure functions.

Therefore, when working with JAX, all input should be passed through function parameters, while all output should come from the function results. Hence, something like Python's print function is not pure.


A pure function returns the same results when called with the same inputs. This is not possible

with `np.random.random()` because it is stateful and returns different results when called several times.

```
print(np.random.random())
print(np.random.random())
print(np.random.random())
```



A screenshot of a Jupyter Notebook cell. The code cell contains three lines of Python code: `print(np.random.random())`, `print(np.random.random())`, and `print(np.random.random())`. The code is highlighted in yellow. To the left of the code is a play button icon and a green checkmark. Below the code cell, the output is displayed, showing three floating-point numbers: `0.3074140005932742`, `0.9790915240559613`, and `0.2850122907118765`. Each output line is preceded by a blue user icon.

JAX implements random number generation using a random state. This random state is referred to as a **key** . JAX generates pseudorandom numbers from the pseudorandom number generator (PRNGs) state.

```
seed = 98
key = jax.random.PRNGKey(seed)
jax.random.uniform(key)
```

You should, therefore, not reuse the same state. Instead, you should split the PRNG to obtain as many sub keys as you need.

```
key, subkey = jax.random.split(key)
```

Using the same key will always generate the same output.

```
✓ [44] key, subkey = jax.random.split(key)
0s

✓ [45] subkey
0s
      DeviceArray([3614062411, 3294896607], dtype=uint32)

✓ [46] jax.random.uniform(subkey)
0s
      DeviceArray(0.95996785, dtype=float32)

✓ [47] jax.random.uniform(subkey)
0s
      DeviceArray(0.95996785, dtype=float32)
```

# Pure functions

We have mentioned that the output of a pure function should only come from the result of the function. Therefore, something like Python's `print` function introduces impurity. This can be demonstrated using this function.

```
def impure_print_side_effect(x):
    print("Executing function") # This is a
    side-effect
    return x

# The side-effects appear during the first
run
print ("First call: ", jax.jit(impure_print_side_effect)(4.))

# Subsequent runs with parameters of same
type and shape may not show the side-effect
# This is because JAX now invokes a cached
compilation of the function
print ("Second call: ", jax.jit(impure_print_side_effect)(4.))
```

```
nt_side_effect)(5.))
```

```
# JAX re-runs the Python function when the  
type or shape of the argument changes  
print ("Third call, different type: ", ja  
x.jit(impure_print_side_effect)(jnp.array  
([5.])))
```

```
Executing function  
First call:  4.0  
Second call: 5.0  
Executing function  
Third call, different type:  [5.]
```

We can see the printed statement the first time the function is executed. However, we don't see that print statement in consecutive runs because it is cached. We only see the statement again after changing the data's shape, which forces JAX to recompile the function. More on `jax.jit` in a moment.

# JAX NumPy operations

Operations on JAX arrays are similar to operations with NumPy arrays. For example, you can `max`, `argmax`, and `sum` like in NumPy.

```
matrix = matrix.reshape(4,4)
jnp.max(matrix)
jnp.argmax(matrix)
jnp.min(matrix)
jnp.argmin(matrix)
jnp.sum(matrix)
jnp.sqrt(matrix)
matrix.transpose()
```

```
✓ 0s ▶ jnp.argmax(matrix)
      DeviceArray(15, dtype=int32)

✓ 0s [38] jnp.min(matrix)
      DeviceArray(17, dtype=int32)

✓ 0s [39] jnp.argmin(matrix)
      DeviceArray(0, dtype=int32)

✓ 0s [40] jnp.sum(matrix)
      DeviceArray(392, dtype=int32)

✓ 0s [41] jnp.sqrt(matrix)
      DeviceArray([[ 4.1231055,  4.2426405,  4.358899 ,  4.472136 ],
                   [ 4.582576 ,  4.690416 ,  4.7958317,  4.8989797],
                   [ 5.          ,  5.0990195,  5.196152 ,  5.2915025],
                   [ 5.3851647,  5.477226 ,  5.5677643,  5.656854 ]], dtype=float32)
```

However, JAX doesn't allow operations on non-array input like NumPy. For example, passing Python lists or tuples will lead to an error.

```
try:
    jnp.sum([1, 2, 3])
except TypeError as e:
    print(f"TypeError: {e}")
    # TypeError: sum requires ndarray or scalar arguments, got <class 'list'> at position 0.
```

---

# JAX arrays are immutable

Unlike in NumPy, JAX arrays can not be modified in place. This is because JAX expects pure functions.

```
scores = [50, 60, 70, 30, 25]
scores_array = jnp.array(scores)
scores_array[0:3] = [20, 40, 90]
# TypeError: '<class 'jaxlib.xla_extension.DeviceArray'>' object does not support item assignment.
# JAX arrays are immutable. Instead of ``x[idx] = y``, use ``x = x.at[idx].set(y)`` or another .at[]
# method: https://jax.readthedocs.io/en/latest/\_autosummary/jax.numpy.ndarray.at.html
```

Array updates in JAX are performed using `x.at[idx].set(y)`. This returns a new array while the old array stays unaltered.

```
try:
    jnp.sum([1, 2, 3])
except TypeError as e:
    print(f"TypeError: {e}")
    # TypeError: sum requires ndarray or scalar arguments, got <class 'list'> at position 0.
```

## Out-of-Bounds Indexing

NumPy usually throws an error when you try to get an item in an array that is out of bounds. JAX doesn't throw any error but returns the last item in the array.

```
matrix = jnp.arange(1,17)
matrix[20]
# DeviceArray(16, dtype=int32)
```

JAX is designed like this because throwing errors in accelerators can be challenging.

# Data placement on devices in JAX

JAX arrays are placed in the first device, `jax.devices()[0]` that is, GPU, TPU, or CPU. Data can be placed on a particular device using `jax.device_put()`.

```
from jax import device_put
import numpy as np
size = 5000
x = np.random.normal(size=(size, size)).as
```

```
type(np.float32)
x = device_put(x)
```

The data becomes committed to that device, and operations on it are also committed on the same device.

## How fast is JAX?

JAX uses asynchronous dispatch, meaning that it does not wait for computation to complete to give control back to the Python program. Therefore, when you perform an execution, JAX will return a future. JAX forces Python to wait for the execution when you want to print the output or if you convert the result to a NumPy array.

Therefore, if you want to compute the time of execution of a program you'll have to convert the result to a NumPy array using `block_until_ready()` to wait for the execution to complete. Generally speaking, NumPy will outperform JAX on the CPU, but JAX will outperform NumPy on accelerators and when using jitted functions.

## Using `jit()` to speed up functions

`jit` performs just-in-time compilation with XLA. `jax.jit` expects a pure function. Any side effects in the function will only be executed once. Let's create a pure function and time its execution time without jit.

```
def test_fn(sample_rate=3000, frequency=3):
    x = jnp.arange(sample_rate)
    y = np.sin(2*jnp.pi*frequency * (frequency/sample_rate))
    return jnp.dot(x,y)
%timeit test_fn()
# best of 5: 76.1 µs per loop
```

Let's now use jit and time the execution of the same function. In this case, we can see that using jit makes the execution almost 20 times faster.

```
test_fn_jit = jax.jit(test_fn)
%timeit test_fn_jit().block_until_ready()
# best of 5: 4.54 µs per loop
```

In the above example, `test_fn_jit` is the jit-compiled version of the function. JAX then created code that is optimized for GPU or

TPU. The optimized code is what will be used the next time this function is called.

## How JIT works

JAX works by converting Python functions into an intermediate language called `jaxpr` (JAX Expression).

The `jax.make_jaxpr` can be used to show the `jaxpr` representation of a Python function. If the function has any side effects, they are not recorded by `jaxpr`. We saw earlier that any side effects, for example, printing, will only be shown during the first call.

```
def sum_logistic(x):  
    print("printed x:", x)  
    return jnp.sum(1.0 / (1.0 + jnp.exp(-  
x)))
```

```
x_small = jnp.arange(6.)  
print(jax.make_jaxpr(sum_logistic)(x_small))
```

JAX creates the `jaxpr` through tracing. Each argument in the function is wrapped with a tracer object. The purpose of these tracers is to record all JAX operations performed on them when the function is called. JAX uses the tracer records to rebuild the function, which leads to `jaxpr`. Python side-effects don't show up in the `jaxpr` because the tracers do not record them.

JAX requires arrays shapes to be static and known at compile time. Decorating a function conditioned on a value with `jit` results in error. Therefore, not all code can be jit-compiled.

```
@jax.jit
def f(boolean, x):
    return -x if boolean else x

f(True, 1)
# ConcretizationTypeError: Abstract tracer
value encountered where concrete value is
expected: Traced<ShapedArray(bool[], weak
_type=True)>with<DynamicJaxprTrace(level=
0/1)>
```

There are a couple of solutions to this problem:

- Remove conditionals on the value.
- Use JAX control flow operators such as `jax.lax.cond`.
- Jit only a part of the function.
- Make parameters static.

We can implement the last option and make the boolean parameter static. This is done

by

specifying `static_argnums` or `static_argnames` .

This forces JAX to recompile the function when the value of the static parameter changes. This is not a good strategy if the function will get many values for the static argument. You don't want to recompile the function too many times.

You can pass the static arguments using Python's `functools.partial` .

```
from functools import partial

@partial(jax.jit, static_argnums=(0,))
def f(boolean, x):
    return -x if boolean else x

f(True, 1)
```

# Taking derivatives with `grad()`

Computing derivatives in JAX is done using `jax.grad`.

```
@jax.jit
def sum_logistic(x):
    return jnp.sum(1.0 / (1.0 + jnp.exp(-
x)))

x_small = jnp.arange(6.)
derivative_fn = jax.grad(sum_logistic)
print(derivative_fn(x_small))
```

The `grad` function has a `has_aux` argument that allows you to return auxiliary data. For example, when building machine learning models, you can use it to return loss and gradients.

```

@jax.jit
def sum_logistic(x):
    return jnp.sum(1.0 / (1.0 + jnp.exp(-
x))), (x + 1)

x_small = jnp.arange(6.)
derivative_fn = jax.grad(sum_logistic, has
_aux=True)
print(derivative_fn(x_small))
# (DeviceArray([0.25          , 0.19661194, 0.
10499357, 0.04517666, 0.01766271,
#
0.00664806], dtype=float32),
DeviceArray([1., 2., 3., 4., 5., 6.], dtype=float32))

```

You can perform advanced automatic differentiation

using `jax.vjp()` and `jax.jvp()`.

# Auto-vectorization with vmap

vmap(Vectorizing map) allows you write a function that can be applied to a single data and then `vmap` will map it to a batch of data. Without `vmap` the solution would be to loop through the batches while applying the function. Using jit with for loops is a little complicated and may be slower.

```
mat = jax.random.normal(key, (150, 100))
batched_x = jax.random.normal(key, (10, 100))
def apply_matrix(v):
    return jnp.dot(mat, v)

@jax.jit
def vmap_batched_apply_matrix(v_batched):
    return jax.vmap(apply_matrix)(v_batched)

print('Auto-vectorized with vmap')
%timeit vmap_batched_apply_matrix(batched_x).block_until_ready()
```

*In JAX, the `jax.vmap` transformation is designed to generate a vectorized implementation of a function automatically. It does this by tracing the function similarly to `jax.jit`, and automatically adding batch axes at the beginning of each input. If the batch dimension is not the first, you may use the `in_axes` and `out_axes` arguments to specify the location of the batch dimension in inputs and outputs. These may be an integer if the batch axis is the same for all inputs and outputs, or lists, otherwise. Matteo Hessel, JAX author.*

## Parallelization with pmap

The working of `jax.pmap` is similar to `jax.vmap`. The difference is

that `jax.pmap` is meant for parallel execution, that is, computation on multiple devices. This is applicable when training a machine learning model on batches of data.

Computation on batches can occur in different devices then the results are aggregated. The `pmap` ed function returns a `ShardedDeviceArray` . This is because the arrays are split across all the devices.

There is no need to decorate the function with jit because the function is jit-compiled by default when using `pmap` .

```
x = np.arange(5)
w = np.array([2., 3., 4.])

def convolve(x, w):
    output = []
    for i in range(1, len(x)-1):
        output.append(jnp.dot(x[i-1:i+2], w))
    return jnp.array(output)
```

```
convolve(x, w)

n_devices = jax.local_device_count()
xs = np.arange(5 * n_devices).reshape(-1,
5)
ws = np.stack([w] * n_devices)
jax.pmap(convolve)(xs, ws)
# ShardedDeviceArray([[ 11.,  20.,  29.],
#
# .....
#
# [326., 335., 344.]],
dtype=float32)
```

You may need to aggregate data using one of the collective operators, for example, to compute the mean of the accuracy or mean of the logits. In that case, you'll need to specify an `axis_name`. This name is important to achieve communication between devices.

# Debugging NaNs in JAX

By default, the occurrence of NaNs in JAX program will not lead to an error.

```
jnp.divide(0.0,0.0)
# DeviceArray(nan, dtype=float32, weak_type=True)
```

You can turn on the NaN checker and your program will error out at the occurrence of NaNs. You should only use the NaN checker for debugging because it leads to performance issues. Also, it doesn't work with `pmap` , use `vmap` instead.

```
from jax.config import config
config.update("jax_debug_nans", True)
jnp.divide(0.0,0.0)
# FloatingPointError: invalid value (nan)
  encountered in div
```

# Double (64bit) precision

JAX enforces single-precision of numbers. For example, you will get a warning when you create a `float64` number. If you check the type of the number, you will notice that it's `float32`.

```
x = jnp.float64(1.25844)
# /usr/local/lib/python3.7/dist-packages/j
ax/_src/numpy/lax_numpy.py:1806: UserWarni
ng: Explicitly requested dtype float64 req
uested in array is not available, and will
be truncated to dtype float32. To enable m
ore dtypes, set the jax_enable_x64 configu
ration option or the JAX_ENABLE_X64 shell
environment variable. See https://github.
com/google/jax#current-gotchas for more.
# lax_internal._check_user_dtype_supported
(dtype, "array")
# DeviceArray(1.25844, dtype=float32)
```

You can use double-precision numbers by setting that in the configuration using `jax_enable_x64`.

```
# set this config at the beginning of the program
from jax.config import config
config.update("jax_enable_x64", True)
x = jnp.float64(1.25844)
x
# DeviceArray(1.25844, dtype=float64)
```

## What is a pytree?

A pytree is a container that holds Python objects. In JAX, it can hold arrays, tuples, lists, dictionaries, etc. A Pytree contains leaves. For example, model parameters in JAX are pytrees.

```

example_trees = [
    [1, 'a', object()],
    (1, (2, 3), ()),
    [1, {'k1': 2, 'k2': (3, 4)}, 5],
    {'a': 2, 'b': (2, 3)},
    jnp.array([1, 2, 3]),
]

# Let's see how many leaves they have:
for pytree in example_trees:
    leaves = jax.tree_leaves(pytree)
    print(f"{repr(pytree):<45} has {len(leaves)} leaves: {leaves}")

# [1, 'a', <object object at 0x7f280a01f6d0>] has 3 leaves: [1, 'a', <object object at 0x7f280a01f6d0>]
# (1, (2, 3), ())
has 3 leaves: [1, 2, 3]
# [1, {'k1': 2, 'k2': (3, 4)}, 5]
has 5 leaves: [1, 2, 3, 4, 5]
# {'a': 2, 'b': (2, 3)}
has 3 leaves: [2, 2, 3]
# DeviceArray([1, 2, 3], dtype=int64)
has 1 leaves: [DeviceArray([1, 2, 3], dtype=int64)]

```

# Handling state in JAX

Training machine learning models will often involve state in areas such as model parameters, optimizer state, and stateful Layer such as BatchNorm. However, jit-compiled functions must have no side effects. We, therefore, need a way to track and update model parameters, optimizer state, and stateful layers. The solution is to define the state explicitly.

# Loading datasets with JAX

JAX doesn't ship with any data loading tools. However, JAX recommends using

data loaders.....