Introduction to Jax

Foreword

My first encounter with `Jax` was in 2018. I was told that `jax` can be very useful for large scale trying to write a numerical relativity simulation in `jax` such that it can leverage modern accelerators to scale the compute. At the time the development experience was horrible, a lot of the documentation was pretty arcane and it was quite hard for me to debug the code. In the end the code was made, but it was nothing more than a fun side project for me to try out `jax`. Over the years, `jax` has seen a lot of community effort in improving its usability and expanding its capability. There is a growing community which is interested in adding `jax` into their scientific computing workflow, and I basically use `jax` for most of my deep learning and scientific computing projects. In this tutorial, hopefully I will be able to convert you to switch to `jax`, since I think that is straightly beneficial.

What is Jax

python on steroid

`jax` is a code transformation library that can do a lot of magic behind the scene, and we will get to that very soon. Syntax-wise `jax` is `python`, even though there are patterns that are encouraged and discouraged in `jax` that requires a different mindset comparing to a normal `python` code, you are allowed to write `numpy`-like code and enjoy all the benefits `jax` has to offer. This is a huge advantage comparing to other options such as writing `CUDA` code, since it basically does not require you to learn a new language.

XLA

One of the original advantage of `jax` is it is codeveloped with `XLA`, which stands for accelerated linear algebra. `XLA` is a compiler that is designed to compile efficient code for accelerators, such as GPUs and TPUs. The idea is to take a high level code, then compile it into a low level code that can be executed on the accelerator. It provides almost hand-optimized `CUDA` code performance in a majority of cases, while not requiring the user to learn how to program in `CUDA`. These days `XLA` has moved on to a standalone project which in principle can be integrated with `PyTorch`, but I think `jax` integration is still the tightest.

Core features

jit

The first feature you may want to use in `jax` is its just-in-time compilation feature. Let's look at an example first:

```
import numpy as np
import jax
import jax.numpy as jnp

def selu(x, alpha=1.67, lambda_=1.05):
  return lambda_ * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)

def selu_numpy(x, alpha=1.67, lambda_=1.05):
  return lambda_ * np.where(x > 0, x, alpha * np.exp(x) - alpha)
```

```
selu_jit = jax.jit(selu)
x = jnp.arange(1000000)
%timeit selu(x).block_until_ready()
%timeit selu_jit(x).block_until_ready()

y = np.arange(1000000)
%timeit selu_numpy(y)
```

Here we create three version of the `selu` function, which is a commonly used activation function in deep learning. The first one is the `jax` version without any optimization, the second one is the `jax` version with `jit` optimization, and the last one is the `numpy` version. Similar to `julia`, the first time you call a jit function will have the compilation overhead. Unlike `julia`, you can choose to compile a function or not, whereas in `julia`, everything is compiled.

Benchmark the three functions, and we can see that the `jit` version is faster than the non-optimized version, and the `numpy` version is slower than the non-optimized version. Why is this the case? There are mainly two factors why this could be the case. The first reason is `jax` can interface seamlessly with `XLA`, which is a compiler that is designed to compile efficient code for accelerators. This compilation produces more optimized code by fusing operations such that the execution time and memory footprint are reduced. Another reason is without changing the source code, your program can run on a GPU or TPU, which makes large and dense computation much faster.

While jitting generally provide performance boost, it is not always the case. For example, if the function is very simple, the communication overhead between the CPU and the accelerator may outweigh the performance gain from the compilation. But this is usually a corner case that is not very practical, so while keeping that in mind, try jitting whenever you can.

grad

Another awesome feature jax provides is automatic differentiation. Again, let's first look at how it works:

```
import jax
import jax.numpy as jnp

def tanh(x):
  return (jnp.exp(x) - jnp.exp(-x)) / (jnp.exp(x) + jnp.exp(-x))

d_tanh = jax.grad(tanh)
assert d_tanh(1.57) == (1./jnp.cosh(1.54))
```

Given a function `f` that is compatible with `jax`, then `jax.grad(f)` returns a function that will return the gradient of `f` with respect to its arguments. If this is not magical enough, the runtime of the gradient function is usually linear in the number of arguments instead of geometric like finite differencing, and the accuracy is usually as accurate as the function itself! How is this even possible?

The idea behind autodiff is basically just chain rules. Say I have a function `sin(x)`, we know exactly it's derivative, which is `cos(x)`. Now if I build on top of the function and I want its gradient with respect to its input, then I just need to apply chain rule and multiply the jacobian I get at each stage. There are a lot of infrastructure behind the scene to make the efficient, but this is the crux of autodiff.

vmap

The third vanilla feature we will learn here is `vmap`, and here is how it looks like:

```
def my_function(x, y):
  return x@y

x = jnp.arange(100)
y = jnp.eye(100)

my_function(x, y)

x_batch = x[None].repeat(10, axis=0)
y_batch = y[None].repeat(10, axis=0)

jax.vmap(my_function)(x_batch, y_batch)
```

`jax.vmap` takes a function as an argument then return a function that add a batch dimension to all of the function's argument, and return that new function as its result. If you want to map over different axis of your input, you can use the `in_axes` argument in `vmap` You can find more detail in the link. If you have ever come across the function `np.vectorize`, you can read from their documentation that `np.vectorize` is actually just a for-loop. While `vmap` achieve the same thing in terms of function signature, the underlying computation is completely different. Since `np.vectorize` is essentially a for-loop, the runtime of the vectorized function is linear in the batch dimension. However, `vmap` add extra annotation to the function before dispatching it to the `XLA`, telling the runtime the newly added batch axis can be parallelized, a vmapped function could be executed just as fast as the original function in most cases, as long as you are not memory bounded by your resource. This means we can acheive much higher throughput while not paying extra time cost. The cost we have to be mindful here is memory, since you are essentially parallelizing over one axis, you are computing more stuff at one time, meaning you will need more memory to store the input and the result.

Sharp bits

While `jax` offers a lot of good features that you should use, it does have some sharpbits that could be a bit unintuitive if you have not worked with these kinds of code transformation libraries.

Jax only

Now you may be wondering how the compiler knows about the gradient of the function you have just coded up, while you are pretty this is a completely original piece code.The answer lies in the fact that ultimately the functions or "primitives" you are allowed to use to construct your original function be known to `jax` already. For example, if you have a function that uses `np.sin`, you must replace it with something `jax` is aware of its internel, in this case `jnp.sin`. If you have functions which `jax` do not know the internel of the function, then `jax` will not be able to convert the function automatically.

The implication of this that needed to be transform has to be in `jax`. If you find a package online that is pretty neat, but it does not use `jax` as its dependencies, you either need to expose the internal of the code to `jax`, or just rewrite the thing in `jax`.

Introduction to Jax

Control flows

Writing control flows in `jax` could be a bit tricky as well. Some of the control flow such as `if` is not allowed if you want to `jit` a function. You may want to consult this page if you want to add control flows to your function. In general, you should compose control flows using the primitives provided in here.

Dynamic allocation is prohibited

In order to efficiently work with accelerators, `jax` aggressively hates any dynamic allocation, i.e. functions that do not know how much memory it will need at compile time. This means in your function, you are almost never allow to something that will change in shape if you want to `jit` it. For example, say you want to sample from a distribution using MCMC, you may be tempted to write something along the line:

```
def step(x, n):
  for i in jnp.arange(n):
      x += i
  return x
```

This will fail to compile if you try to `jit`, the reason is the size of the `samples` list is not known at compile time, and the compiler cannot allocate memory for it. The way to fix this is to use `jax.lax.scan`, which we will see an example of this later.

Long compilation with for loop

While `jax` can provide really good run time performance, and most deep learning practitioners will not be acutely aware of the compilation overhead, the compilation overhead can be pretty annoying when it comes to scientific computing. The difference between deep learning and scientific computing is despite being compute heavy, deep neural network are very simple program. The core of many neural network are just matrix multiplication, so there are not many lines to the program. On the other hand, scientific programs often come with a wide variety of forms, and there could be many interdependent modules, hence often way more lines of code involved. At its core, `jax`'s `jit` system is still running on `python`, and overhead related to `python` could make compilation slow. To make it worse, for loops are quite common in scientific computing, and if one naively write a for loop and trying to compile it in `jax`, `jax` will unroll the whole for loop and inline everything. This means the compilation time of your program scales linear as the number of loops you have. One example is when I first tried `jax` in 2018, I was writing a numerical relativity simulation in `jax`, hoping to capitalize on the use of accelerator from `jax`. While the simulation runtime is much faster than alternatives (30 minutes vs a couple of hours), the compilation time is about 2 hours long, which completely defeat the point of having a fast code. In a later section, we will see what is the recommended way to write such operation.

Ecosystem

jaxtyping

`jaxtyping` is a package to add type hints for `jax` arrays and `pytree`. Instead of annotating an array as `jnp.ndarray`, you can do something like `Float[Array, "n m"]`, which is a type hint for a 2D array with `n` rows and `m` columns. This is very useful since shape mismatch is one of the most

common error in deep learning workflows, and having a type hint for the shape of the array can help you catch the error early. You can find more information about `jaxtyping` here.

Equinox

`Equinox` is a library that allows you to build neural network like you would in `PyTorch`. Before I discovered `Equinox`, I was using `Flax`, which is not very intuitive to me, since model parameters are external to the model, and calling functions from a particular layer is not very ergonomic. `Equinox` mirror `PyTorch` interface quite well, while preserving some of the perks one may want to use in `jax`. For example:

```
import equinox as eqx

class MyLayer(eqx.Module):
  weights: Float[Array, "n_out n_in"]
  biases: Float[Array, " n_out"]

  def __init__(self,
    weights: Float[Array, "n_out n_in"]
    biases: Float[Array, " n_out"]
  )
    self.weights = weights
    self.biases = biases

  def __call__(self, x: Float[Array, " n_in"]):
    return self.weights @ x + self.biases

weights = jax.random.uniform(jax.random.PRNGKey(0), shape = (5, 10))
biases = jax.random.uniform(jax.random.PRNGKey(1), shape = (5))
layer = MyLayer()

test_input = jnp.arange(10)
layer(test_input)
```

This is a very simple example to code your own layer, and here the variable `layer` are fully compatible with `jit`, `grad` and `vmap`. Further down the road if you continue to use `jax`, your model is also fully compatible with all the distributed training tools that `jax` provides.

optax

`optax` is a library that provides a lot of optimization algorithms for `jax`. You will find a lot of the common optimization algorithms such as `adam` in `optax`. Using optax is slightly different from `PyTorch`, in the sense that the optimizer states need to be initialized and passed around explicitly, and you have to do something like `params = optax.apply_updates(params, updates)` to change the parameers. We will see an example of how to use `optax` in the next section.

diffrax

`diffrax` is library for solving differential equations in `jax`. There are a bunch of cool features and solvers in `diffrax`. And here is the coolest thing about having `jax` libraries: most of the code should be composable with all the `jax` features we have seen. This means your *solvers* and *optimizer* are vmappable, jittable, and probably differentiable. This implies you can train multiple neural networks at the same time on a single GPU, and many more cool stuffs.

Introduction to Jax

Modeling a pendulum

Modeling dynamic system with deep neural network is getting a lot of attention because of its implications. Many infrastructures in our society are designed with very principle models, from turbines to electrical transformer (Not to be confused with the deep learning architecture). If deep learning can improve these models performance, it will lead to better infrastructure hence has huge societal impact.

In this section, we are going to attempt to solve one of simplest dynamic problem: forward modeling a pendulum system. Now instead of providing the position and velocity of the pendulum to you, let say your lab mate is also very deep learning brain, and the person provides you videos of the pendulum instead. We are going to simulate a dataset of image of pendulum, then model the system with a neural network. We are going to build a basic CNN to emulate the system. There is also some note on a more involved solution that combines a VAE and a neural ODE to model the system, but I figure we will not have enough time in the class so I am leaving it as optional.

Step 1: Solving the dynamics

A pendulum system can be modeled as the following second order ODE:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin\theta$$

where g is the acceleration due to gravity, l is the length of the pendulum, and θ is the angle of the pendulum. When implementing a ODE solver, we often favor solving a coupled first order ODE instead of the second order ODE, so we are going to rewrite the equation as:

$$\frac{d\theta}{dt} = \omega$$
$$\frac{d\omega}{dt} = -\frac{g}{l}\sin\theta$$

We are now ready to solve this system forward in time with `diffrax`. Take a look of the example here here, and implement the `simulate_pendulum` function in `generate_data.py` in your template code.

The function signature of `simulate_pendulum` is given to you, and your task is to create the relevent objects in the function and fill out the line

```
sol = dx.diffeqsolve(
  ...
)
```

Make sure to include `args` such as the length of the pendulum and gravity, and remember to create `dx.Saveat` object to save the trajectory of the pendulum.

Step 2: Rendering an animation

The next step is to render the simulation into images which we will use as our data. Instead of actually generating the image and save as a png, we are only going to save the output of the simulation as a `jax.numpy` array. This allows us to `vmap` over the rendering function later. You are required to fill out the `render_pendulum` function in the `generate_data.py` file.

Introduction to Jax

Once again the function signature of `render_pendulum` is given to you. Here is the pseudo code of the function:

1. Generate a grid denotating the position of each pixel with `jnp.meshgrid`
2. Given the angle of the pendulum, compute the coordinate of the pendulum with respect to the center of the image
3. Compute the distance from each pixel to the pendulum with `jnp.linalg.norm`, and set the pixel's value to be 1 if the distance is less than the radius of the pendulum, and 0 otherwise.
4. (Optional) Plot the image to see whether the pendulum is rendered correctly.

Step 3: Creating our data

Now we have the simulation code and rendering code, we simply have to loop over them to generate a dataset for our machine learning model. You are required to fill out the `generate_dataset` function in the `generate_data.py` file. You may notice the function signature is a bit odd, that we require two input frames and only predict one output frames (as seen in the channel). Think a bit about why this is the case, and you are also welcome to modify the function to take one input frame and output one frame and see what happens.

The thing you have to pay attention to in this function is you should use `vmap` to render the pendulum frames instead of a for loop. Another thing is since we don't necessary have to jit this complete function[1], you can use non-`jax` object such as a list.

Finally, to ensure reproducibility, the random number system in `jax` works a bit differently than `numpy`. You should use `jax.random.PRNGKey` to generate random number in `jax`. Instead of `np.random.uniform`, you have to provide an additional key to the random generator, which can be generated using `jax.random.PRNGKey(your_integer)`, which acts as a seed. And instead of manually inputing all the key, you can use `jax.random.split` to generate a list of keys, then use the keys in your function. You can find more information about the random number system in `jax` here.

Building an emulator with `Equinox`

Now we have our data from the previous section, let's build an emulator to model this problem. Our task is defined to be: given a snapshot of the current system, predict how would it look like in the coming time step. Since our data is in the form of images, let's try to build a convolutional neural network to model our dataset.

Step 1: Buidling the model

Building a model in `Equinox` is very similar to building a model in `PyTorch`. You should find the starter code in `models.py`. Let's see what the essential components for our model:

1. You should see the variable `layers` as class variable. This is where you can define relevant variables to use in your class. For this tutorial, `layers` should be sufficient.
2. In the `__init__` function, you should put the code which creates the layers of your model. For this specific tutorial, we will use the `eqx.nn.Conv2d` layer and `jnp.tanh` as our activation. Checkout the documentation for more information. You may also need Lambda

[1] You are welcome to jit the whole thing, but that is a bit more involved so we are not going to do it here.

After that, you just need to implement the `__call__` function, which is the forward pass of your model. Since we have access to `jit`, it is actually okay to just loop through the layers.

Step 2: Writing the training loop

You should be able to find the template code in the `train_models.py` file. There are a couple of things you have to implement for the training loop to work:

1. Implement a mean square error as your loss function in `loss_fn`. There is a tricky bit related to how you should handle a batch of data. It is not too difficult to figure out so I will leave that as a puzzle.
2. In the main train function, you need to define an optimizer. We will use `optax` to define our optimizer, you can find information here.
3. You will have to implement the `make_step` function. This could be a bit involved, so I leave a lot of hints in the body of the function. The idea here is this function will take your model, optimization state, and a batch of data as input, then spit out a new model, new optimization state, and the loss.
4. Finally, you have to loop over a certain number of epochs to train you model.

Step 3:Train and evaluation

Once you have completed the training loop, it is time to generate the data and train your model. Generate a small amount of training data using functions you have defined in `generate_data.py`, then train your model with the training loop. After you have trained your model, try making a simulation of the pendulum and compared it with a true simulation.

A slight rant

Since I have a physics background, and I have worked on some simulations before, I have a stroke everytime there are some obviously oblivious "machine learning experts" claim their network can accelerate simulations for orders of magnitudes. I once believed in that, only to later find out the field of deep learning is filled with bogus claims and these experts have no idea what they are talking about. The setting they provide is usually follow the pattern shown here:

1. There are some benchmark datasets they either download from some where or generated from some codes
2. The claimed SOTA code is slow.
3. They use some neural networks to model the simulation end-to-end, i.e. emulating the simulations.
4. Loss goes down, and they show some pictures of the simulations saying they look the same.
5. Claim victory.

If you are coming from a machine learning background, this is basically standard practice. Propose some new architecture, show improvement on benchmark metrics, then you can publish in a big conference. As long as you have enough buzzwords in your paper, you should make it to a big conference with relative ease. And as a data science aligned person, you may have the impulsion of slapping a deep learning model to solve any problem, may it be CNN, graph, normalizing flow, or LLM. After all, that's what you learn in school. Unfortunately, despite all the hype you have heard about deep learning, as of September 2024, there are almost no deep learning solutions that are state-of-the-art, and in fact, most ML solutions are far from SOTA[2].

[2]For interested students, there is a fun recent paper about deep learning in CFD.

Introduction to Jax

The problem is, these days training neural networks has become so fast and easy that people forgot neural network has the following disadvantage:

1. They are not accurate, especially in a small data regime.
2. They are expensive. A GPU is often needed to iterate through a dataset.
3. They are often uncontrollable. I have full control and guaruntee over a RK4 solver, but I cannot say the same about even an MLP.

These properties are often overlooked when an ML team is given a problem. We raise a whole generation of students and engineers that can only solve problem through neural network, GPUs, and more data. But real life problems come with constraints that may not allow you to do so. GPUs cost money, electricity costs money, collecting data costs money, and ML engineers are massively overpriced. Let's go back to the pendulum system for a second: the inaccuracy in ML makes the CNN solution unreliable. You may argue you just need more data and maybe data augmentation to make the CNN works better, but that costs money and time. A competent engineer who actually care about solving the problem should think about how to combine all the tools one has access to and solve the problem in the fastest and cheapest way possible. In this case, running the experiment is very cheap, now instead of collecting the images, I rather just measure the position and velocity of the pendulum, probably through attaching a string to readout through a smooth pulley. Once I have the data, I don't even need a computer to create a model that will certainly beat any of the machine learning model here, it's called physics buddy.

Best practices

Think like a GPU

If you intend to run something on the GPU, it is helpful to think like a GPU. The way GPU (or other typical accelerators) work is through "packing" compute, meaning they usually manipulate a large batch of data at once. The clock speed of a GPU core is usually slower than a CPU core, but the GPU has many more cores. So when you write a program in jax, you should thinkg about how to execute as many unit of compute in parallel as possible. Say if your GPU can do 100 units of compute per cycle, if you do not saturate that bandwidth, the extra cycles are simply "wasted". So when you write a `jax` program, you should always ask yourself whether the way you write it is designed for a GPU.

Avoiding in-place mutation

Speaking of think like a GPU, you should avoid in-place mutation. In-place mutation is when you change the value of a variable without creating a new variable, usually happens when you mutate the values within an array, i.e. `x[0] = 1`. You can see how this is not very GPU friendly, since it requires the computer to go to a very specific location and operate locally, instead of applying a lot of changes everywhere. In `jax`, in-place mutation is not allowed, and if you try to do so, you will get an error. If you have to mutate some values, instead of `x[0] = 1`, you have to do `x = x.at[0].set(1)`.

Scan your for loop

We have mentioned writing a for-loop can result in long compilation time when `jit`-ing the function. One way to avoid this is to use `jax.lax.scan`. `jax.lax.scan` is a function that allows you to write a for-loop in a way that is more friendly to the compiler. Essentially it is telling the compiler it doesn't

not have to unroll the loop and just compile the body of the loop once. The syntax of `jax.lax.scan` is a bit weird. While the official documentation is here, here is an example of how you can use it:

```
import jax
import jax.numpy as jnp

def body(carry, x):
  return carry + x, carry

def scan_example(x):
  return jax.lax.scan(body, 0, x)

scan_example(jnp.arange(10))
```

The output of the function is `(45, array([0, 1, 3, 6, 10, 15, 21, 28, 36, 45]))`. The first element of the tuple is the aggregated value of the loop, and the second element is the value of the loop at each step.

jit at top level

When you transform a function with `jit`, it looks through your code and try to optimize the code before compiling it. So if you `jit` at the highest level possible, the compiler has more information to compile more efficient code. Because of this, usually I save the `jit` until the very last moment.

check your gradient

Sometime you may have some wacky result in script and you have checked everything you have thought off. You may have printed the value of the function and see not problem at all. The thing is sometimes the function itself can be well defined by the gradient can be `nan`. For example, if you use the function `nansum`, which sum the array while ignoring `nan`, the gradient of the final function can be broken and may not be as useful as intended.