Autodiff, PyTorch, and JAX

Machine Learning Journal Club

2024/05/08

Chatipat Lorpaiboon

Table of contents

Autodiff 101

Autodiff (automatic differentiation)

- Define the derivative of each "primitive" function
- The derivative of a function implemented using the primitives are automatically defined using the chain rule
- Two extremes for computing applying the chain rule:
  - Forward-mode
  - Reverse-mode

$$w = e^{-x^2} \qquad \frac{dw}{dx} = -2xe^{-x^2}$$

$$\begin{array}{ccc}
y = x^2 & \longrightarrow & \frac{dy}{dx} = 2x \\
\downarrow & & \\
z = -y & \longrightarrow & \frac{dz}{dy} = -1 \\
\downarrow & & \\
w = e^z & \longrightarrow & \frac{dw}{dz} = e^z
\end{array}$$

Forward-mode autodiff

- Key operation: Jacobian-vector product
$$\sum_j \frac{df_i(x)}{dx_j} v_j$$
- Single forward pass for both values and derivatives
- No need to store all intermediates in memory
- Use when output size > input size

$$w = e^{-x^2} \qquad \frac{dw}{dx} = -2xe^{-x^2}$$

$$\begin{tikzcd}
y = x^2 \arrow[r] \arrow[d] & \frac{dy}{dx} = 2x \arrow[d] \\
z = -y \arrow[r] \arrow[d] & \frac{dz}{dx} = -\frac{dy}{dx} \arrow[d] \\
w = e^z \arrow[r] & \frac{dw}{dx} = e^z \frac{dz}{dx}
\end{tikzcd}$$

Reverse-mode autodiff (backpropagation)

- Key operation: vector-Jacobian product
  $$\sum_i v_i \frac{df_i(x)}{dx_j}$$
- Two passes
  - Forward pass to compute values
  - Backward pass to compute derivatives
- Intermediate values from forward pass are stored in memory for backward pass
- Use when input size > output size

$$w = e^{-x^2} \qquad \frac{dw}{dx} = -2xe^{-x^2}$$

$$\begin{tikzcd}
y = x^2 \arrow[r] \arrow[d] & \frac{dw}{dx} = 2x\frac{dw}{dx} \\
z = -y \arrow[r] \arrow[d] & \frac{dw}{dy} = -\frac{dw}{dz} \arrow[u] \\
w = e^z \arrow[r] & \frac{dw}{dz} = e^z \arrow[u]
\end{tikzcd}$$

PyTorch

PyTorch basics

https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html

- Numpy-like API, but many differences
- PyTorch has very good tutorials and documentation

How PyTorch autodiff (mostly) works

- Only reverse-mode is supported in normal PyTorch
- PyTorch creates computational graphs on-the-fly
  - Forward pass creates the graph and saves intermediates for the backward pass
  - Backward pass computes derivatives and may delete the graph

https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html

```python
class LinearFunction(Function):

    @staticmethod
    def forward(input, weight, bias):
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    @staticmethod
    def setup_context(ctx, inputs, output):
        input, weight, bias = inputs
        ctx.save_for_backward(input, weight, bias)

    @staticmethod
    def backward(ctx, grad_output):
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None

        if ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias
```

PyTorch packages

- torch.nn — neural networks
- torch.optim — optimizers
- torch.func — function transforms (similar to JAX)
  - Support for forward-mode autodiff (but incomplete)
  - Support for higher-order derivatives (but incomplete)

Optimization

- Asynchronous execution — PyTorch operations queue work for GPUs and return immediately
- Use composite operations (PyTorch has 1200+ operations!)
- torch.compile — JIT compile a function or model
  - https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html
- torch.jit — convert a PyTorch module into a TorchScript model (which can be run faster)
  - https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html
  - torch.jit.trace — trace a PyTorch module
    - Less impact on code, advanced Python features can still be used
  - torch.jit.script — translate the source code of a PyTorch module
    - Supports Python control flow, limited to a supported subset of Python (similar to numba)

Ecosystem

- https://pytorch.org/ecosystem/
- PyTorch Lightning —
  - https://github.com/Lightning-AI/pytorch-lightning
  - Removes boilerplate, makes organization nicer
  - Frequent API breaking changes
- PyG (PyTorch Geometric) — graph neural networks
  - https://github.com/pyg-team/pytorch_geometric

JAX

JAX basics

https://jax.readthedocs.io/en/latest/quickstart.html

https://jax.readthedocs.io/en/latest/notebooks/Common_Gotchas_in_JAX.html

- Use jax.numpy and jax.scipy instead of numpy and scipy
- JAX arrays are immutable
  - Instead of *x[idx] = y* use *x = x.at[idx].set(y)*
  - https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html
- jax.random — JAX explicitly passes random number generator state
  - https://jax.readthedocs.io/en/latest/random-numbers.html
- jax.nn — activation functions, etc.
- jax.lax — primitives (use jax.numpy and jax.scipy instead if possible)

JAX transformations

- Autodiff
- jax.jit — compile functions to make them faster
- jax.vmap / jnp.vectorize — vectorize functions over batch dimensions (gufuncs in numpy terminology)

Autodiff

Forward-mode autodiff (use when #outputs > #inputs)

- jax.jacfwd — Jacobian wrt first argument
- jax.jvp — Jacobian-vector product (for a single vector)
- jax.linearize — Jacobian-vector product (stores graph so it can be evaluated for multiple vectors)

Reverse-mode autodiff (use when #inputs > #outputs)

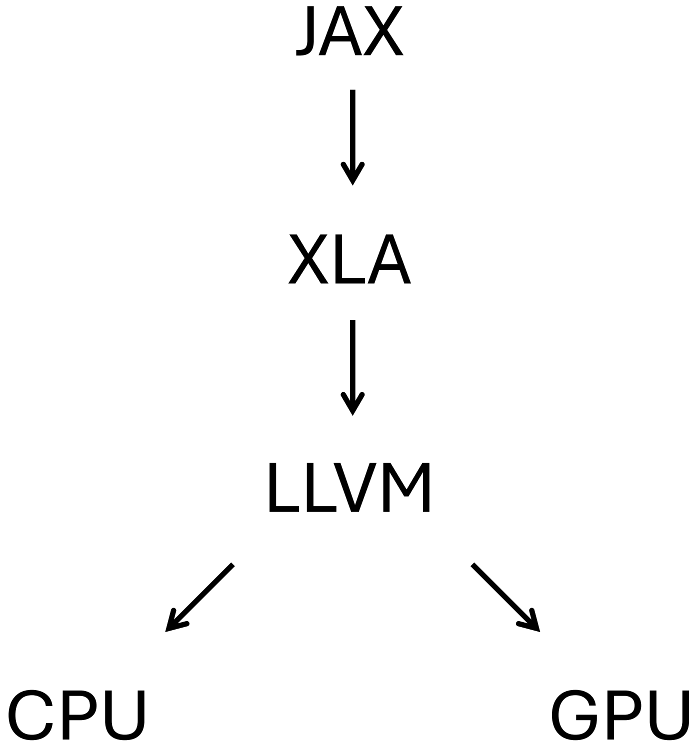
- jax.grad — gradient of scalar function wrt first argument
- jax.value_and_grad — value and gradient of scalar function wrt first argument
- jax.jacrev — Jacobian wrt first argument
- jax.vjp — vector-Jacobian product (stores graph so it can be evaluated for multiple vectors)

Other

- jax.hessian — Hessian wrt first argument
- jax.lax.stop_gradient — identity function, but with the derivative set to zero

How JAX works

- JAX creates and transforms (e.g., autodiff, vectorization) computational graphs
- XLA optimizes and evaluates computational graphs
- XLA uses LLVM to generate code for CPU and GPU



JAX assumes functions are *pure*

- Applying a function to the same inputs must always return the same outputs
  - Any variable that can change should be an input argument
- Functions cannot have side effects
  - No output including printing (but see jax.debug)
  - No mutating inputs — return a modified copy instead
  - Mutating objects created inside the function is okay
- Using JAX transformations (grad, jit, etc.) with functions that aren't pure is undefined behavior

Tracing a function

- Tracing does **not** use the source code of the function
- Input arrays are replaced with tracers
  - Tracers have shapes and types but don't have values
- The function is evaluated and operations on the tracers are recorded
  - Computation that doesn't involve the tracers are assumed to be constant (this is why the function must be pure)
- The result of tracing is a computational graph of the function

Optimization

- Asynchronous dispatch
  - https://jax.readthedocs.io/en/latest/async_dispatch.html
  - Operations on JAX arrays return immediately
  - Computation is done in the background
  - JAX waits for the computation
- Jit compilation: jax.jit
  - Function must be traceable
  - Must use JAX control flow (no if/for/while statements) for conditions that depend on values in JAX arrays
  - Use on the outermost function so JAX can optimize more

Ecosystem

JAX does not include neural networks or optimizers

- Flax — most popular neural network library
    - Complicated implementation; errors messages can be difficult to understand
    - JAX transformations don't work *inside* neural networks; Flax transformations must be used instead
- Equinox — more elegant neural network library
    - JAX transformations work properly
    - Models easier to manipulate and debug
    - Shared parameters and stateful operations (e.g., batch norm) can be painful
- Optax — gradient-based optimizers
- Orbax — checkpointing
- Chex — asserts, debugging, etc.
- Jraph — very barebones graph neural network library (use PyTorch instead if possible)
- Distrax — normalizing flows
- BlackJAX — MCMC sampling

See https://github.com/n2cholas/awesome-jax for more

Gotchas

- Functions are assumed to be pure: undefined behavior if they aren't
- Tracing through large loops = compiling takes forever
  - Use JAX control flow instead
- Out-of-bounds indexing is undefined behavior (instead of raising an exception like in numpy)
- 64-bit is disabled by default (enable by running *jax.config.update("jax_enable_x64", True)* before any JAX code)
- JAX automatically chunks intermediates if they take too much CPU/GPU memory (but not inputs or outputs)

Escape hatches

- jax.pure_callback — use regular Python (e.g., numpy) to compute values inside a JAX transform
  - Derivatives can be implemented using jax.custom_jvp, etc.
- jax.debug.print — try to print an intermediate (may be not printed or printed multiple times)
- jax.debug.callback — try to call a function (e.g., save intermediate to disk)