2024.12.17

Keras first impressions

design principle:
progressive disclosure of complexity

only one per import

Backends: JAX,
Tensorflow, PyTorch

works with NumPy, Pandas,
Tensorflow `Dataset`,
PyTorch `DataLoader`
regardless of backend

packaged with popular
dataset ~~downloaders~~
Pipelines

models are similar to
layers but can have
component models

the layer abstraction

`init` with configuration,
`build` if it has
persistent data/params
depending on input
shape, `call`

with `Input` layer
builds automatically,
without it build
manually with batch
input shape

model-centric

```
model = keras.Sequential([   ...list of layers...   ])
model.compile(loss, optimizer, metrics)
callbacks =
   [ModelCheckpoint(..), EarlyStopping(..)]
model.fit(x_train, y_train, batch_size,
          epochs, validation_split, callbacks)
score = model_evaluate(x_test, y_test)
model.save("final_model.keras")
predictions = model.predict(x_test)
```

2024.12.18

in OCANNL, a tensor expr. function shares weights, a layer / block with `~config` does not

reusing a layer expression in different models does not share weights, in same model shares weights

composing with a model shares weights

upcoming:
- training and eval
- distributed training

model inputs can be a list, outputs can be a dictionary

functional API = layer expressions instead of `Sequential`

saved model includes:
- architecture (layer expression)
- weight values (params)
- training config
- optimizer and its state

for cyclic or recursive computations: subclass `Model`

Keras styles: Sequential, functional, subclassing

auto-propagated `call` args

`mask`

`training`

can mix-and-match Sequential, layer expressions and subclassing -- via composing (sub)models

bool tensor if model input shape

train vs. inference, handled by built-in train, eval, predict loops

layers and models have a `trainable` flag

individual weights can also be non-trainable

regenerated per-call

layers can `add_loss` to models that use them

2024.12.19

Keras training

sample weights: per-sample influence on loss

class weights: balance classes without resampling

Data sources i.e. input pipelines are iterator-based (except NumPy), offer batching and shuffling, keras-specific one is mulitcore.

Callback class has methods specific to: begin/end of whole/batch/epoch of train / test i.e. eval / predict i.e. infer

Dynamic learning rate schedules are callbacks that modify the optimizer.

Ideas for callbacks: checkpointing, early stopping, changing learning rate when plateau, fine-tuning of top layers when plateau, emailing on performance thresholds, TensorBoard, CSVLogger.

For saving/loading, custom layers etc. must define get_config, usually captures init arguments.

Progressive intervention into a model's training:
- override train_step and/or test_step (of eval) using model's forward-call and loss initerface;
- as above but inline loss;
- write the training and/or eval loop from scratch.

allows e.g. subclassing a GAN model

examples generate the derivative at each train step

JAX example jit-compiles the full train step

2024.12.20

Keras distributed

same as `tensorflow.dtensor`

In OCANNL, better fit to link DeviceMesh with a routine rather than a tensor.

OCANNL's `DeviceMesh : dev`

grid configured manually but sharding done by program search

per-cluster mesh config passed to the mesh backend functor

`DeviceMesh`

organizes devices into N-dim grid with axis_names

`TensorLayout`

assigns axes of any tensor (positionally) to sharded on a given mesh axis, or replicated

tied to a device_mesh (might initially be unset)

is synchronous

no events

`DataParallel`

automates setting the layout

`ModelParallel`

batch_dim_name specifies the data parallel part of the layout

`LayoutMap`

can contain the default device_mesh

a fuzzy (regexp) map from parameters (weight variables) to TensorLayout

2024.12.21

to be continued

JAX distributed

counter-based PRNGs are better for parallelism

sharding mesh + PartitionSpec (like TensorLayout) = device-like

`Array`: like DTensor

layout propagation / inference

inferred layout of outputs minimizes copying

error when explicit layouts of inputs disagree

default layout inputs can be moved and resharded automatically to fit other inputs

`with_sharding_constraint` redirects layout inference

`shard_map` takes a mesh and partition specs for inputs and output

unassigned input axes are replicated / tiled as in DTensor

partitions tensors preserving the ranks (i.e. nums of axes)

mapped func result shape must have rank sufficient for concatenation of shareding axes in output partition spec

unassigned output axes are un-replicated: result is selected from just a subset of devices, assuming that it's the same on other groups of devices

caller can pick mesh axes that are propagated rather than set manually on inputs / output

2024.12.22

JAX collectives

- communicate across devices from within shard_map
  - to overlap comp. and comm. reshape to add an axis and loop over it inside the map
    - if not overlapped by XLA
- all_to_all
  - transposes blocks along an on-device and a cross-device axis
- ppermute
  - sends tensor(s) by permuting a mesh axis
- all_gather
  - concatenates blocks along an axis, replicating a tensor
- psum
  - replicates the summed axis
- psum_scatter
  - = ppermute + add, no replication
- for best shift perf on TPUs, split blocks in half and shift bidirectionally

NN parallel patterns in JAX

- data parallel
  - shard data on a batch axis, pmean the loss
    - other sharding is automatic
- FSDP
  - also shard params, on the batch mesh axis
    - all_gather inside predict, jax.remat to re-gather on backward pass
- tensor parallel
  - shard data and params on corresponding features axis, psum_scatter activations
- FSDP + TP
  - explicit psum for features (in TP automatic sum->psum)
- SPMD pipeline parallel
  - for same structure layers: shard_map over concatenated params, ppermute to advance the pipeline

2024.12.23

JAX distributed multi-host

processes must agree on per-device sizes

very restrictive approach:
- SPMD: all processes same computations
- all processes same number of devices
- all devices the same (e.g. H100)

control flow must not diverge, watch out: length of training loop, iteration order

but allows running `shard_map` etc. without changes

death of any process kills others

each JAX process runs independently, no one controller but one coordinator

NVIDIA backend: Collective Communications Library NCCL

JAX integrates with tf.data.Dataset

sometimes the storage locality disagrees with computation locality -- load `jax.Array` with storage sharding, and add `with_sharding_constraint` for efficient resharding

2024.12.26

XLA instruction set

- `XlaOp` = tensor
- control-flow-like
  - `Conditional`
  - `While`
  - `Scatter`, `SelectAndScatter`: non-deterministic loop of updates
- `Clamp` to min/max
- `OptimizationBarrier`
- `AfterAll` for sequencing (like tensor-centric events)
- cross-replica: `AllGather`, `AllReduce`, `AllToAll`, `CollectivePermute`, `ReduceScatter`
- persisted autotuning: cache on disk for speed and determinism
- `Recv` and `Send`: communicate via shared channel
- domain- or algo-specific
  - `BatchNormTraining/Grad/Infer`
  - `ConvWithGeneralPadding`
  - `Fft` forward and inverse Fourier
  - `Cholesky`, `TriangularSolve`
- `CompositeCall`: to define composite functions
- can define asynchronouns funcs: start, update loop, done
- `Infeed`: reads a tensor from an implicit channel on a device
- `Iota`: constant literal initialized on device without transfer
- vectorized: `Reduce`, `Map`
- tensor structure
  - `Gather` general idea: convert a list of offsets into tensors into a tensor with a new batch dimension
  - `Concatenate`
  - `Broadcast`
  - `Collapse`
  - `Transpose`: permute axes

Also arithmetic

2024.12.27

XLA Emitters

3 compilation routes: libraries like cuBLAS & cuDNN; tiling followed by Triton; Emitters

Partitioning: tensors are emitted in a single function when they interact pointwise without duplication.

Subkernel function inputs: "inflow" tensors and indices of "outflow" tensors; outputs: "outflow" values at the indices. Kernel function: takes both "inflow" and "outflow" tensor args.

Only single-call functions are inlined.

loop traversals linear in output tensors for coalesced writes, with boundary checks inside

tensors flattened to 1D as in memory

loop unrolling

Transpose and Reduction emitters, using shared memory

two loops: coalesced reads to shared mem; then `sync_threads`; then coalesced writes

Loop emitter is default (no "hero")

Other emitters: Concatenate, Dynamic Update Slice, Input slices, Scatter

symbolically computes indexing maps between tensors, e.g. input<->output

for reasoning on mem. coalescing and tiling propagation

for emitting index transformations (transpose, broadcast, reshape, slice, reverse)

only contiguous accesses get inlined as transfer reads