`moolib`: A Platform for Distributed RL

Vegard Mella, Eric Hambro, Danielle Rothermel, Heinrich Küttler

January 10, 2022

Abstract

We present `moolib`, a library that enables the implementation of distributed reinforcement learning and other machine learning codebases. Our implementation aims to be both simple and scalable, targeting researchers with a wide range of available computing resources, e.g., from individual to hundreds of GPUs. `moolib` is build around efficient remote procedure calls (RPCs) for both tensor and non-tensor data. Together with the `moolib` library, we present example user code which shows how `moolib`'s components can be used to implement common reinforcement learning agents as a simple but scalable distributed network of homogeneous peers. Together with this whitepaper, `moolib` and its examples are provided as open source code at our repository at `github.com/facebookresearch/moolib`.

1 Introduction

Reinforcement learning (RL) has enjoyed a sustained interest in recent years as part of the broader deep learning revolution, itself a consequence of both improved hardware and new algorithms. The seminal breakthroughs for this approach were super-human results on Atari [1] and the game of Go [2], a long-standing grand challenge of Artificial Intelligence (AI).

At the same time, recent results in RL have arguably been less accessible, harder to reproduce and harder to build on than breakthroughs in other domains in Machine Learning (ML). Often, details that are critical to the success of a specific RL method's implementation are omitted from the write up [3], making the released code a requirement for reproducible research. Many results depend on highly efficient implementations which can be difficult to use or understand, which leads researchers to prefer of simpler, less efficient models where possible. Moreover, the design assumptions underpinning one algorithm's efficient implementation might be a considerable hindrance to new research directions, to the point of making it impossible to try out particular new ideas.

In this paper we present `moolib`, a library that aims to address these issues by satisfying the requirements of both the high performance and high understandability ends of the spectrum, striking a balance where they are in conflict. It does so by offering a highly performant and flexible system based on asynchronous operations for sending data within a group of peers. `moolib`'s primary target is to enable efficient implementations of RL agents, but its components are useful for the wider field of ML as well, see below. From a high-level perspective, `moolib` consists of two parts: (1) A library for distributed machines learning codebases which offers gradient accumulation, RPCs, and some additional features, some of which are RL-specific. (2) Implementations of example RL agents with the help of this library. These example agent implementations are written to be both simple and scalable, see Algorithm 1 below for an overview of a prototypical `moolib` agent.

Remote procedure calls. At the heart of `moolib` is its Remote Procedure Call (RPC) functionality, which is designed to be fast and flexible. RPCs are a convenient abstraction for distributed computation. In the context of machine learning, RPCs have recently been leveraged for building efficient distributed reinforcement learning systems, e.g. [4, 5, 6]. However, RPCs are used throughout machine learning;

```
// Setup.
accumulator := new moolib accumulator
envs := new batched environment
model := randomly initialized neural network model
optimizer := new stateful optimizer
env_future := envs.reset()
while not steps < total training steps do
    if not connected then
        wait(a_bit)
        continue
    end
    if accumulator.has_gradients() then
        optimizer.step()                                  // Apply gradients we received.
        accumulator.zero_gradients()              // Turns has_gradients() to False again.
    else if len(rollouts) $\geq$ unroll_length and accumulator.wants_gradients() then
        // Consume data, compute gradients, and start async accumulation.
        loss := compute_loss(model, rollouts)
        loss.backward()                                         // Produce local gradients.
        accumulator.reduce_gradients(local_batch_size)            // Returns immediately.
    else
        // No gradients to produce or apply.  Generate data.
        env_outputs := env_future.result()
        actor_outputs := model(env_outputs)
        rollouts.append(env_outputs, actor_outputs)                       // One timestep.
        env_future := envs.step(actor_outputs.action)             // Returns immediately.
    end
end
```
Algorithm 1: Pseudocode for an agent implemented with `moolib`. For runnable code that still avoids the complications of actual experiment code, see `examples/a2c.py` of the open source release. For code with all necessary features for experiments, see `examples/vtrace/experiment.py`.

one recent example is [7][1]; RPCs are also used in the recent [8] and [9][2]. See Section 3.3 for a discussion of `moolib`'s RPC features.

Different requirements in RL. The computational resources required for research in the field of RL vary wildly. Interesting theoretical and empirical results have been achieved with the help of computational resources at either end of a scale ranging from single machines, including laptops, to thousands of special-purpose hardware accelerators like GPUs or TPUs [10]. At each end, very different architectural choices may be required. At one end computationally intensive implementations must prioritise performance above other considerations; at the other end ease of use, understandability and flexibility have a greater priority. These priorities needn't be permanently at odds. A modern deep learning library with the right architectural design choices could in principle allow the same experiment to be run along a broad section of this spectrum. `moolib` aims to be an example of such a library. The example implementations that come as part of `moolib` try to offer simplicity by including all code relevant to research in a single file of a few hundred lines, consisting mainly of a single loop instruction that runs for the duration of the training process, see Algorithm 1 and Section 3.1 below.

[1]In [7], the authors rely on RPCs to host an index of text embeddings (> 100 GB in RAM). In this case, loading one index per GPU isn't feasible for multi-GPU training, but neither is pre-computing results per query because live access to the index is needed for backpropagating to the query vector. Instead, an 'index server' implemented with a precursor of `moolib` met all the requirements.

[2]Although neither publication mentions RPC or network communication, these techniques are an important part of their implementations (source: private communication).

2 Related Work

Many existing frameworks provide comparable functionality to `moolib`. For RPCs in RL and ML more broadly, there's Ray [11], RLgraph [12], Reverb [5], Launchpad [6]. Some core ideas that inspired `moolib` can also be found in SEED RL's set of gRPC TensorFlow ops [4, file `grpc/ops/grpc.cc`], or even in the earlier 'dynamic batching' feature of IMPALA [13]. Outside of the field of machine learning, generic communication libraries like gRPC [14] and ∅MQ [15] also offer some comparable features, and are in fact the basis of more specialized libraries such as Reverb or SEED RL. Somewhat in between is the TensorPipe [16] library, which targets ML code in the sense that it optimizes sending and receiving buffer data, including via CUDA, but is otherwise quite general; `moolib` is built on top of TensorPipe. An alternative backend could be TorchRPC or `DistributedDataParallel` (DDP), both parts of recent versions of PyTorch [17]. TorchRPC is also built on TensorPipe.

For RL agents, again, many existing open sourced implementations have comparable, if slightly different, goals to `moolib`. Among them are A3C [18], IMPALA [13], PPO in its various versions [19, 20, 21, 22], RLlib [23], Tianshou [24], SEED RL [4], rlpyt [25], TorchBeast [26], SaLinA [27], Acme [28], and Sample Factory [29]. We argue that having this multitude of options is fortunate and attests to the healthy state of research in the field of RL: The design space of agent implementation is arguably larger than, say, that of training code for computer vision classifiers, and is in fact itself an active area of study to which we are looking to further contribute. Our hope is that `moolib` adds a solution that's simpler than most of what is currently available, while still being able to scale to a large number of GPUs.

3 `moolib`

3.1 High Level Design

`moolib` enables a large number of system designs, but we ship the prototypical setup along with the library release as an example. The design aims to optimize the use of accelerators, enable high throughput, and be easy to reason about and modify. The setup consists of a group of homogeneous *peers*, each iterating through a copy of the same main loop as described in pseudocode in Algorithm 1. Each of these peers interacts with its own instance of the `Accumulator`, see Figure 3.1, which handles the asynchronous communication between peers.

This segmentation of functionality was chosen for readability and ease of modification. Using this group of homogeneous peers deviates from the original IMPALA design [13, 26], which uses a heterogeneous group of peers consisting of one *learner* and several *actor* instances. Although `moolib` allows implementing this design via RPCs, see Section 3.3, a dynamically-sized group of homogeneous peers is often easier to reason about and manipulate.

The `Accumulator` can be thought of as a state machine which transitions between three states: (1) 'wants gradients', where it waits to receive gradients from the peers; (2) 'reducing gradients', where it asynchronously reduces the gradients and (3) 'has gradients', where the accumulated gradients are available for use by the peer. This stateful design allows for the simple training loop shown in Algorithm 1, which abstracts away the necessary parallelism done in background threads without user-defined callbacks.

`moolib` optimizes for efficiently using accelerators and for overall throughput by providing "asynchronous dispatch" versions of all of its operations. In order to efficiently use an accelerator, such as a GPU, it must be consistently "kept busy", i.e., perform computations at near maximum capacity with minimal idle time. While this can be achieved with tools that enable multiple flows of control, such as threads, this can lead to complicated user-facing code. Additionally, the popular Python programming language does not support this type of parallelism well, due to its design around the global interpreter lock (GIL) mutex, among other reasons.[3] Instead, `moolib` provides "asynchronous dispatch" versions of all of its operations. From a high-level perspective, these are:

- asynchronously accumulating gradients across nodes,

[3]However, there are concrete plans to fix this situation, see `github.com/colesbury/nogil`. For a discussion of the GIL in the context of RL, see [26, Section 5.3]

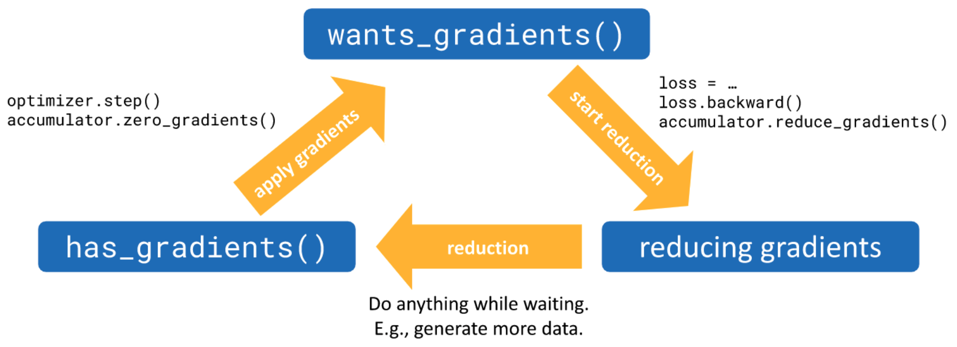

Figure 1: The `moolib Accumulator` as a state machine: From the user's perspective, the `Accumulator` transitions from (1) the 'wants gradients' state, where it waits to receive gradients from the peers, to (2) the 'reducing gradients' state, where it asynchronously reduces the gradients, to (3) the 'has gradients', where the accumulated gradients are available for use by the peer, and back to (1). Note that `wants_gradients()` and `has_gradients()` are `moolib` API calls, while the 'reducing gradients' state is active only when neither of the two functions return true.

- asynchronously stepping a full batch of environments,
- asynchronously evaluating the neural network model (implicitly, e.g., via PyTorch on CUDA).

Since all operations involving heavy computation or network communication are asynchronous, it's possible to keep all accelerators continuously occupied with model forwards or updates. `moolib` chooses the fastest available transport method for gradient accumulation to further increase throughput, and using more than one set of batched environments can additionally increase the GPU and host CPU utilization via a 'double buffering' technique.

For a thorough description of the `Accumulator` and the asynchronous operations, see Section 3.2. `moolib` also offers more primitive building blocks for arbitrary peer-to-peer communication in the form of its RPC mechanism, described in Section 3.3. The details of `moolib`'s all-reduce operation are provided in Section 3.4. Finally, some additional components of `moolib` are described in Section 3.5: the `EnvPool` for asynchronous batched environments, the `Broker`, and the `Batcher`.

3.2 Accumulator

The `Accumulator` is the primary interface that enables multi-peer and multi-node training. Its primary purpose is to all-reduce gradients such that all peers may contribute to the gradients used for training, allowing large-scale training, e.g., by having a large "virtual" batch size, or simply being able to consume training data at a higher rate.

At creation time, the `Accumulator` is given a handle to an RPC group of peers (see Section 3.3 for more details) along with a list of shared parameters and buffers. From the user's perspective, the `Accumulator` manages the transitions of a "virtual" state machine as described in Figure 3.1. Each peer interacts with its own instance of the `Accumulator`, performing operations based on the current state as seen in Algorithm 1. When the `Accumulator` is in the 'wants_gradients' state, each peer calculates losses and uses the `Accumulator` to reduce its gradients with those from the rest of the peers. When the `Accumulator` is in the 'has_gradients' state then each peer takes an optimizer step with all of the gradients provided by the `Accumulator`. During the remaining time, while the `Accumulator` is reducing the gradients, the peers continue to collect samples in anticipation of the next state change. This powerful interface abstracts away gradient manipulation and asynchronous operations, allowing for a simple training loop. It also enables a graceful cleanup operation, e.g., on a user-supplied keyboard interrupt, as well as a step-by-step 'debuggability'. The latter property is usually not present systems with fully independent heterogeneous peers such as TorchBeast [26].

This is somewhat analogous to PyTorch's `DistributedDataParallel` (DDP) interface, but with some important differences. With DDP, all peers would operate in "lock-step", i.e., they would all calculate their loss (and perform the backward pass), the gradients would be all-reduced, and they would all step their optimizer synchronously. With `Accumulator`, only *some* peers may calculate their loss function, and the `Accumulator` would determine if this is enough to fill the desired virtual batch size. If so, the gradients would effectively be all-reduced, and all peers would step their optimizers synchronously. If the virtual batch size isn't filled yet, a given peer may further contribute by calculating its loss function (and performing the backward step) on another batch of data. If a peer has not yet generated enough data to compute a loss and contribute to the gradients, it will still receive the accumulated gradients from the peers that did so. This allows the training loop to progress with minimal overhead due to waiting, even with a large number of peers or with a slow environment.

Among peers participating through an `Accumulator` object, a leader is determined. The leader is responsible for distributing its current state, among them model parameters, optimizer state, and any other state if required by the user, to other peers joining the training. This state is also distributed at a regular interval. This ensures all peers are kept synchronized and allows new peers to join ad-hoc. Note that the regular update is not usually necessary for peers to remain synchronized, but it protects against shifts due to floating point inaccuracies, and it synchronizes some state which may not remain implicitly synchronized, such as PyTorch buffers used for instance for batch normalization.

3.3 Remote Procedure Calls (RPCs)

RPCs are the core of `moolib`. While the concept has recently employed mainly in an RL context (e.g., [5, 6]), it is also used for e.g. NLP projects such as [7, 8, 9]. RPCs are increasingly used because they are easier to reason about and deal with than the underlying mechanisms. Instead of having to track the details of establishing a connection, serializing the procedure's name and arguments, sending this data via network packets, interpreting the data on the other side, calling the procedure in question and sending the return value, if any, back in the same fashion, an RPC library allows a simpler API built on top of the existing primitives.

`moolib`'s RPC functionality is build on TensorPipe [16], the transport library used in PyTorch, which allows for optimizing throughput while maintaining fault-tolerance. Via TensorPipe, `moolib`'s RPC provides automatic transport selection, which employs whichever available transport mode is fastest. The available transports are POSIX shared memory (if on the same machine), InfiniBand [30], and the generic TCP/IP. If the available hardware allows it, TensorPipe also enables `moolib` to send tensor data directly from one CUDA device to another, bypassing the host memory entirely. This allows for an effective throughput of roughly 10 GB/sec. With this throughput, this part of the program is unlikely to be the bottleneck in most ML applications. Additionally, `moolib`'s RPCs are designed to be fault-tolerant. In case of temporary connection issues, RPCs are tried again, if necessary via a different transport type.

Caller and Server API. The RPC primitives are designed with flexibility in mind, with multiple options for designing both the server and the caller.

When building a server with `moolib`'s Python API, the user can choose from three different entrypoints which provide different levels of built-in threadpool management. The simplest one, which defers all thread management to the `moolib` RPC library, is called `define`. It binds a Python function to a given name and executes the function, whenever specified by incoming calls, in a thread owned by `moolib`. A trade-off to this simplicity is that the moolib thread in question cannot service other incoming connections during that time; also, due to Python's GIL, no other Python code can run while this function call executes Python code. More fine-grained control is available via `define_queue`. Here, incoming calls add the function's arguments and a promise-like callback function to a queue. The user will need to include additional functionality in the server to service that queue, execute the corresponding function or otherwise return a result to the caller. Finally, the most general way to define a procedure for the RPC is `define_deferred`, which instead of adding the function's arguments and a callback to a queue, simply calls a super-specified function with this tuple of $(\text{callback}, \text{data})$ as arguments.

When designing the caller side, `moolib`'s API provides three possible entrypoints as well. A `sync` call is synchronous, i.e., blocks until the result of the function call has arrived at the caller, and returns that result. This is the closest to standard function calls within the same program. To make asynchronous calls, `async_` and `async_callback` are available. The former returns a `Future` object which can be waited on, cancelled, or checked for completion. The latter instead lets the user supply a callback function which will be invoked with the result once it's available.

Besides the core functionality of sending tensor data as arguments or return values, `moolib` allows sending any serializable Python object including lists, dictionaries, tuples, etc, via Python's builtin `pickle` module.

Connecting Peers. In order to make procedure calls across the peer network, it is first necessary to discover and connect all of the relevant peers in the network. `moolib` provides an ad-hoc name service where each peer must supply a globally unique *peer* (or *RPC*) *name*[4]. A given peer can find any peer known to its already known peers, i.e., name discovery works with up to one level of indirection while peers two hops away or more won't be found. It is possible to build custom peer discovery with these primitives, but `moolib` also offers a standalone program to facilitate this process: the `Broker`. See Section 3.5 for more information on the `Broker`, but at a high level it provides a central node in the peer network which allows for communication between any pair of peers within the group.

Batched RPCs. For some applications (e.g., a faithful implementation of the IMPALA [13] system like the one provided by TorchBeast [26]), it is helpful to have support for (*dynamically*) *batched* RPCs. In a batched RPC, one or more callers call the same function at approximately the same time, with each caller sending and expecting a return value of tensor data. The specified function on the server is then invoked only once, but with each argument extended by an additional *batch dimension*. Then the return value is required to have the same batch dimension size, and the caller associated with the ith input slice will receive the ith output slice. This functionality can be especially useful for building a model server which serves a ML model on an accelerator like a GPU. An early version of such a dynamic batching feature appeared in the source code release of IMPALA [13], while batched RPCs for TensorFlow 2 are directly available as part of SEED RL [4].

Various modes of batched RPC are possible, e.g., callers could supply batched arguments themselves; the server could decide whether to wait or run with the current list of inputs based on a timeout or arbitrary user logic. `moolib` provides a simple version of batched RPCs with possible extensions in the future.

3.4 All-reduce operations

In order to reduce gradient tensors, i.e., retrieve the local gradients from each peer, compute their sum or average, and send the result to each peer, `moolib` uses a method we call *forest all-reduce*. In certain other situations, `moolib` uses tree all-reduce. We begin by describing these algorithms.

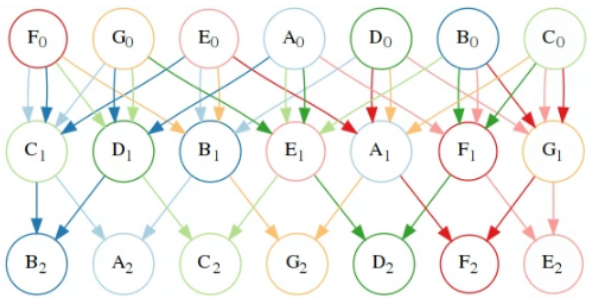

Figure 2: Illustration of the first half of forest all-reduce. Here, different colors indicate different trees, all of which operate in parallel. The color of each node is the color of the tree it is a root of. Data travels from the leaves, through every node, to the root. In the second half of the algorithm, data travels back from the root to each node (not shown).

Tree all-reduce works by first constructing a binary tree, with each peer represented as a node in the tree. Then, starting from the leaves, data is sent up the tree, the reduce operator is applied on each step, until it reaches the root. Thereafter, the final result is sent back towards the leaves to distribute the result. The time complexity at any given step of the algorithm is assumed to scale with the maximum data that any peer sends or receives in that step. Summed over all steps, we get the time complexity of the entire algorithm. This corresponds well with the assumptions that peers have individual network interfaces that can

[4]`moolib` has helper functions to produce unique identifiers (uids). In practise, we found a combination of hostname, process id and a random human-readable slug like Python's `coolname` to be most helpful. An example would be `learnmachine7:4020365:mega-dragon`.

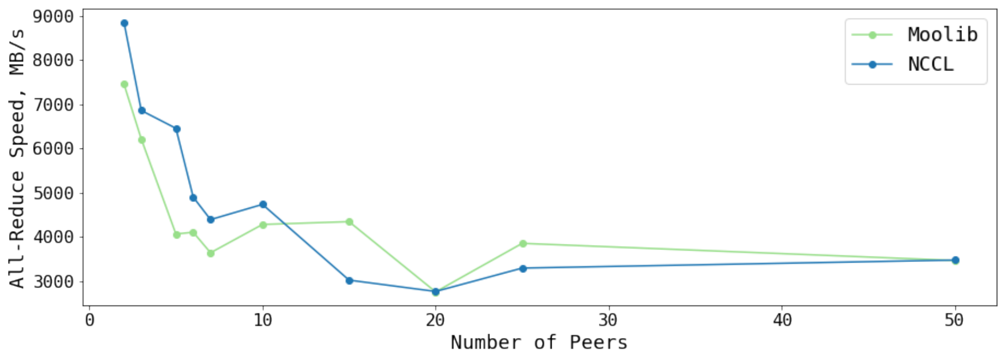

Figure 3: Comparison of Moolib vs NCCL all-reduce, reducing 128MB CUDA floating-point data over an increasing number of machines using InfiniBand interconnect.

operate in parallel and that network bandwidth
is the limiting factor. Let N be the data size that we wish to all-reduce, and P be the number of peers. Since any node in a binary tree has at most 2 children, and the depth of the binary tree is $\lfloor \log_2(P) \rfloor$, we can calculate the worst-case time complexity as $2N(\lfloor \log_2(P) \rfloor$.

The *forest all-reduce* method works by splitting the data up into T chunks, and performing one tree reduction per chunk, in parallel. We will initially assume that $T = P$. The topology of each tree is unique.

In the case of $T = P$, each peer is the root of one tree and a leaf in half of the trees. The total number of steps to finish the algorithm is still $\lfloor \log_2(P) \rfloor$, but the bandwidth required for a single data transfer is now $\frac{N}{T}$. Since a given peer is a leaf of $\frac{T}{2}$ trees, it needs to send $\frac{N}{T} \cdot \frac{T}{2} = \frac{N}{2}$ data in the first step. The amount of data received by any node in this step is the same, as there are $\frac{T}{2}$ leaves, $\frac{T}{4}$ parents of leaves, each one receives data from 2 children, and $\frac{N}{T} \cdot \frac{T}{4} \cdot 2 = \frac{N}{2}$. In the next step, since the trees are binary, the number of nodes halves and so does the required bandwidth. This proceeds until it reaches the root node, requiring a total time of $\frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \cdots \approx N(1 - \frac{1}{2^D}) \approx N$, where $D = \log_2(P)$ is the depth of the tree. Once data has reached the root node, it needs to travel back towards the leaves to distribute the results, doubling the required time to $2N$, which means the time complexity of the algorithm is constant with regards to the number of peers, and scales only with the size of the data.

When reducing tensor-like data where individual chunks can be reduced, `moolib` employs forest all-reduce. For other kinds of data, e.g., serialized Python objects, `moolib` uses tree all-reduce instead.

3.5 Other Utilities

In addition to the components listed above, `moolib` also provides utilities to facilitate specific aspects of distributed RL agent implementations, including the `EnvPool`, the `Batcher` and the `Broker`.

The `EnvPool` is an efficient implementation of 'batched environments', analogous to the 'vectorized environments' of [20]. Due to limitations of the Python programming language when it comes to parallelism, running several copies of the environment simultaneously requires using multiple operating system processes, which in turn requires inter-process communication (IPC). `moolib`'s `EnvPool` implementation is based on semaphores in shared memory, which allows for a very fast handover of observation and action buffers.

Next, the `Broker` facilitates peer discovery. In order to communicate, distributed peers within a single run of a machine learning experiment must solve the problem of *discovery*, i.e., knowing how to find each other. Since `moolib` allows the one-step discovery of all peers known to an already known peer, discovery of all peers can be achieved by supplying a single central instance which we call the `Broker`. Starting an instance of the `Broker` and pointing all peers to its known address offers a simple way to allow discovery of all peers in `moolib` programs. Note that only a single `Broker` is necessary for any number of simultaneous independent experiments within the same network and all associated peers can then be identified via a common *group name*.

Finally, to facilitate generating batched trajectories `moolib` also includes a `Batcher` class which concatenates its inputs along a given dimension. This is used to generate sequences along a 'time' dimension, as well as to produce minibatches along a 'batch' dimension.

4 Experiments

4.1 Scaling Performance

In order to assess the scalability of `moolib`, we measure the all-reduce speed as we increase the number of peers participating in the operation and compare to the performance of the Nvidia Collective Communications Library (NCCL). As seen in Figure 3, `moolib` is competitive with NCCL, with strong performance even at a large number of peers while additionally providing ease of use for distributed machine learning implementations.

4.2 Performance on Atari

In order to validate our example agent implementation, we ran experiments on the Arcade Learning Environment (ALE; also known as just 'Atari') [31]. We use the same model, comparable environment settings as well as the same V-trace loss as in [13, 26] and compare to the baseline from [26], see Figures 4 and 5. We match or exceed the performance of the TorchBeast agent on most levels. See Appendix A.1 as well as the accompanying source code release for details.

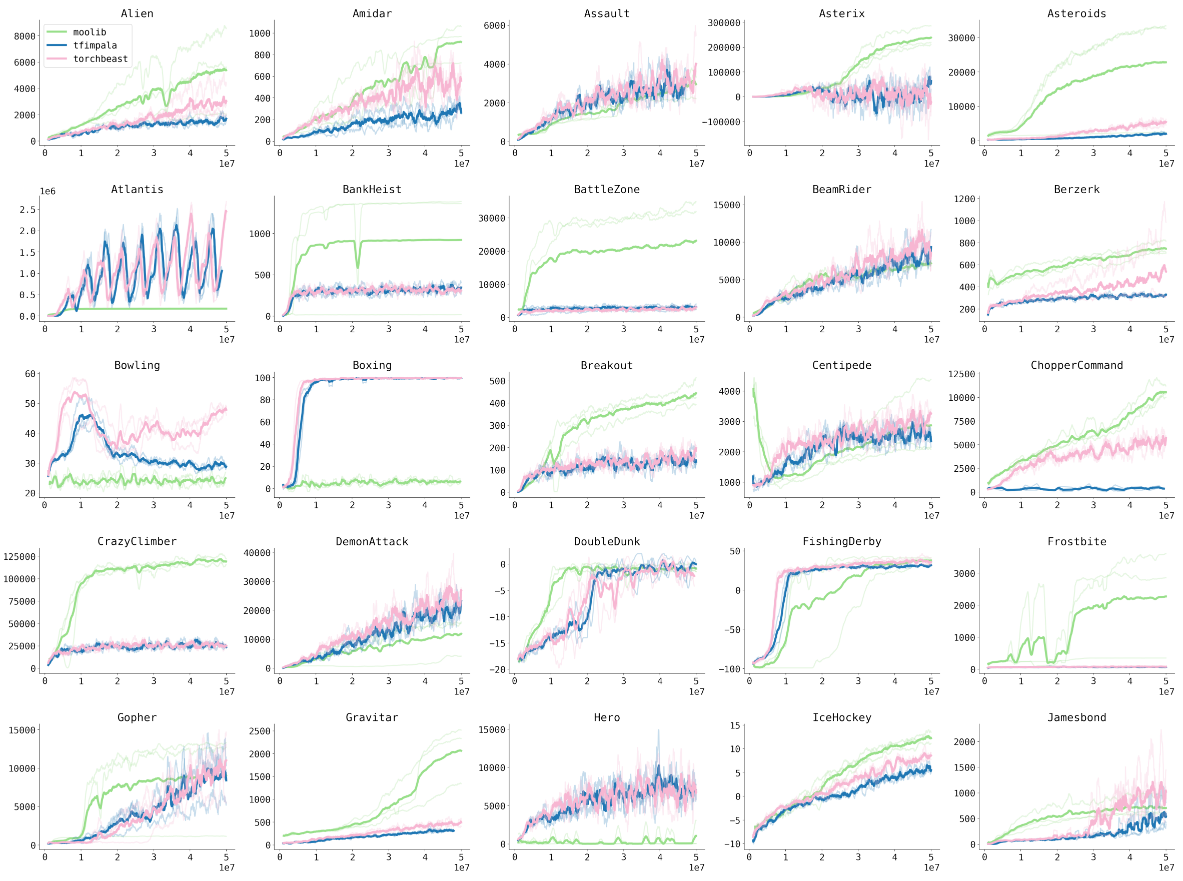

Figure 4: `moolib`'s V-trace example agent on selected Atari games (first half). Here, the x axis is agent steps (50 million agent steps corresponding to 200 million environment frames due to action repetitions) while the y axis is undiscounted episode return.

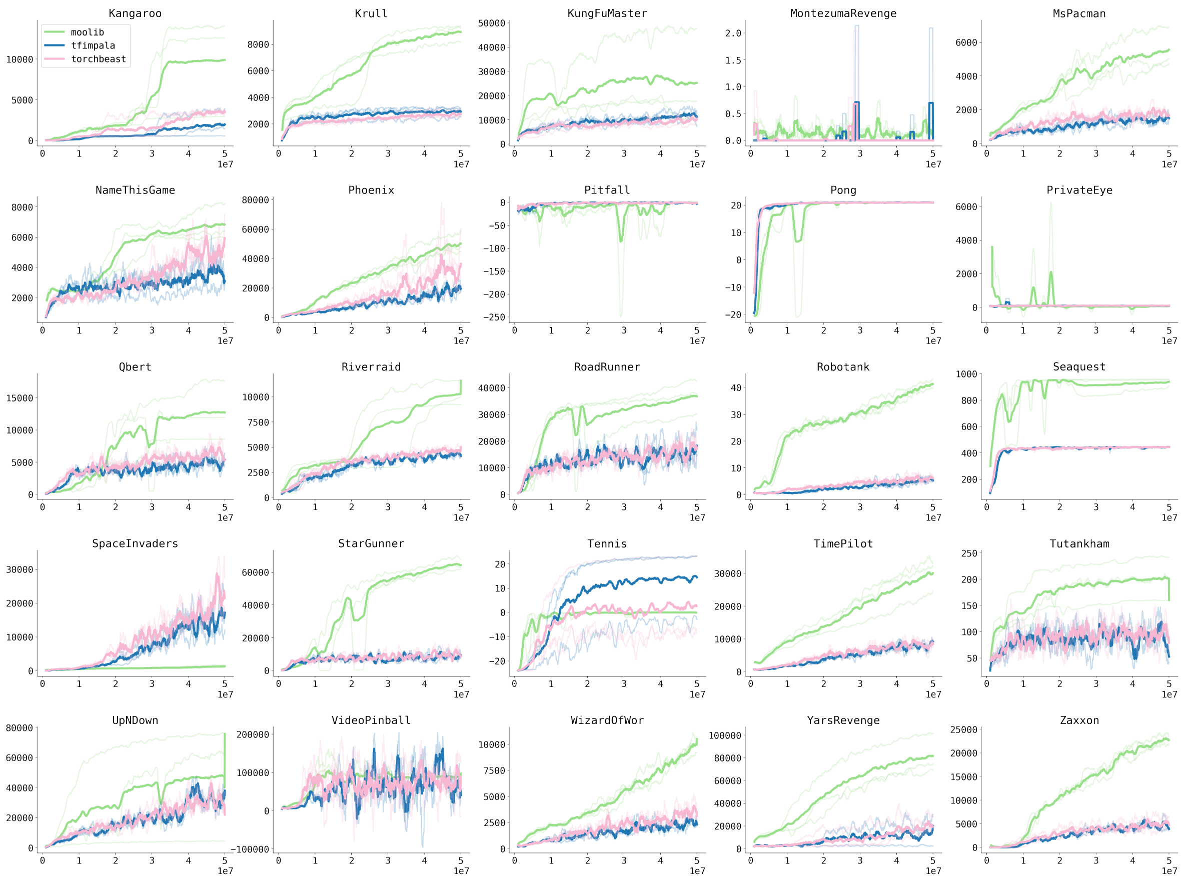

Figure 5: `moolib`'s V-trace example agent on selected Atari games (second half). Axes as in Figure 4.

5 Conclusion

We open-source `moolib`, a library that enables the implementation of distributed reinforcement learning and other machine learning codebases. Our design aims to be simple, fast, and amenable to new research needs, while being scalable to a large number of GPUs. Together with this library, we present example user code which shows how `moolib`'s components can be used to implement common reinforcement learning agents as a simple but scalable distributed network of homogeneous peers.

We discussed some aspects of `moolib`'s components and their implementation, in particular the `Accumulator` and the all-reduce operation. We also evaluated `moolib` on the Atari task suite and compared its performance to the TensorFlow IMPALA [13] implementation published by its authors as well as to TorchBeast [26]; `moolib` performs as least as good as these baselines on most tasks, much better on some. We also compared `moolib`'s raw performance with the more specialized NCCL, where `moolib` proves to be competitive.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through Deep Reinforcement Learning. *Nature*, 518(7540):529, 2015.

[2] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[3] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO, 2020.

[4] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. 2019.

[5] Albin Cassirer, Gabriel Barth-Maron, Eugene Brevdo, Sabela Ramos, Toby Boyd, Thibault Sottiaux, and Manuel Kroiss. Reverb: A Framework For Experience Replay, 2021.

[6] Fan Yang, Gabriel Barth-Maron, Piotr Stańczyk, Matthew Hoffman, Siqi Liu, Manuel Kroiss, Aedan Pope, and Alban Rrustemi. Launchpad: A Programming Model for Distributed Machine Learning Research. *arXiv preprint arXiv:2106.04516*, 2021.

[7] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks, 2021.

[8] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens, 2021.

[9] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent SIfre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher, 2021.

[10] Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable Reinforcement Learning, 2021.

[11] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications, 2018.

[12] Michael Schaarschmidt, Sven Mika, Kai Fricke, and Eiko Yoneki. Rlgraph: Modular computation graphs for deep reinforcement learning, 2019.

[13] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In *ICML*, 2018.

[14] Google Inc. gRPC: A high performance, open-source universal RPC framework. `https://grpc.io`, 2015.

[15] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013.

[16] Luca Wehrstedt, Lucas Hosseini, and Pieter Noordhuis. TensorPipe: A tensor-aware point-to-point communication primitive for machine learning. `https://github.com/pytorch/tensorpipe`, 2020.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019.

[18] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *CoRR*, abs/1602.01783, 2016.

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017.

[20] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. `https://github.com/openai/baselines`, 2017.

[21] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*, 2021.

[22] Ilya Kostrikov. PyTorch Implementations of Reinforcement Learning Algorithms. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`, 2018.

[23] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3053–3062. PMLR, 10–15 Jul 2018.

[24] Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Hang Su, and Jun Zhu. Tianshou: A Highly Modularized Deep Reinforcement Learning Library. *arXiv preprint arXiv:2107.14171*, 2021.

[25] Adam Stooke and Pieter Abbeel. rlpyt: A Research Code Base for Deep Reinforcement Learning in PyTorch, 2019.

[26] Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. TorchBeast: A PyTorch Platform for Distributed RL. *arXiv*, abs/1910.03552, 2019.

[27] Ludovic Denoyer, Alfredo de la Fuente, Song Duong, Jean-Baptiste Gaya, Pierre-Alexandre Kamienny, and Daniel H. Thompson. SaLinA: Sequential Learning of Agents, 2021.

[28] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A Research Framework for Distributed Reinforcement Learning. *arXiv preprint arXiv:2006.00979*, 2020.

[29] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning. In *ICML*, 2020.

[30] Gregory F Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, 42(617-632):10, 2001.

[31] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2013.

[32] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents, 2017.

A Appendix

A.1 Atari settings and hyperparameters

For our experiments on Atari, We use the same architecture as in [13, 26], namely a feed forward convolutional network with residual connections, followed by a fully connected layer, followed by a policy and a baseline head. We refer to the accompanying source code for the full details. We also use the V-trace same loss function as in [13, 26]. A list of hyperparameters can be found in Table 1. A list of settings for ALE can be found in Table 2. Note that we don't follow the best-practises layed out in [32] in order to stay comparable with [26].

Parameter	**Value**
Actor batch size	128
Baseline cost	0.5
Discount rate γ	0.99
Entropy cost	0.0006
Grad norm clipped at	40
Optimizer (Adam) learning rate	0.0006
Optimizer (Adam) β_1	0.9
Optimizer (Adam) β_2	0.999
Optimizer (Adam) ε	10^{-8}
Total steps	50M agent steps[5]
Unroll length	20
Learner batch size	32
Reward norm clipped at	1.0

Table 1: Settings for ALE

Setting	**Value**
ALE version	Release v0.7
OpenAI Gym version string	`ALE/`*`game`*`-v5`
Observation downsampled to	84×84
Resizing method	bilinear (`cv2.INTER_LINEAR`)
Frame stacking	4
Action repeats	4
Frame pooling	2
No-ops at start	30
Color mode	grayscale
Loss of life ends episode	false
Max frames per episode	108K
Action space	full (18 actions)
Sticky actions	No

Table 2: Settings for ALE

[5]200M environment frames due to action repeats (frame skip).