RLGRAPH: FLEXIBLE COMPUTATION GRAPHS FOR DEEP REINFORCEMENT LEARNING

Michael Schaarschmidt [* 1] **Sven Mika** [* 2] **Kai Fricke** [3] **Eiko Yoneki** [1]

ABSTRACT

Reinforcement learning (RL) tasks are challenging to implement, execute and test due to algorithmic instability, hyper-parameter sensitivity, and heterogeneous distributed communication patterns. We argue for the separation of logical component composition, backend graph definition, and distributed execution. To this end, we introduce RLgraph, a library for designing and executing high performance RL computation graphs in both static graph and define-by-run paradigms. The resulting implementations yield high performance across different deep learning frameworks and distributed backends.

1 INTRODUCTION

The recent wave of new research and applications in deep learning has been fueled by both hardware improvements and deep learning frameworks simplifying design and training of neural networks (Chen et al., 2015; Abadi et al., 2016; Seide & Agarwal, 2016; Paszke et al., 2017). Reinforcement learning (RL) algorithms combined with deep neural networks have in parallel emerged as an active area of research due to promising results in complex control tasks (Levine et al., 2016; Tobin et al., 2017; Silver et al., 2017). However, their design and execution have not found similar standardization. This is a consequence of the highly varied resource requirements, scheduling, and communication patterns found in constantly evolving RL methods. Implementations hence require a high degree of customization.

A number of RL libraries has emerged to focus on distinct aspects of managing such workloads. For example, OpenAI baselines provides a number of reference implementations meant to reproduce specific benchmark environments (e.g. Atari) (Sidor & Schulman, 2017). TensorForce provides a declarative API focusing on ease of use in applications (Schaarschmidt et al., 2018). Ray RLlib seeks to simplify distributing RL workloads by moving from hand-designed distributed communication to actor-based centralized execution on Ray (Liang et al., 2018; Moritz et al., 2017).

While these libraries serve different purposes, many of them suffer from similar design problems leading to difficulties in testing, distributed execution, and extensibility. The root cause of these difficulties lies in a lack of separation of con-

cerns. Composition of logical components defined within an RL algorithm is tightly coupled with code fragments specific to a deep learning framework (e.g. TensorFlow calls to define placeholders and variables). This leads to ill-defined APIs and also makes the reuse and testing of components difficult. Similarly, the often complex dataflow within RL algorithms is intertwined with control flow regulating (distributed) execution, environment interaction, and device management. In many of the frameworks we surveyed, this typically results in distributed execution and local device strategies being tied to specific algorithms.

The central contribution of this paper is **RLgraph**, an RL framework addressing many of these issues. RLgraph achieves this by separating logical component composition, creation of operations, variables and placeholders, and finally the local and distributed execution of the component graph (Fig. 1). At the core of our design is a novel meta graph architecture responsible for assembling and connecting algorithmic components, such as buffers or neural networks, and for exposing their functionality to a common API. Importantly, this meta graph exists independently of implementation specific notions (e.g. TensorFlow variables), and instead relies on generalized space objects and operations. This means the meta graph can both be built for static graphs and define-by-run execution, and RLgraph currently supports both TensorFlow (TF) and PyTorch (PT).

The meta graph is built into a backend-dependent computation graph via a graph builder which generates operations, variables, device assignments, and a registry for the API of the model. The meta graph build also includes type checking and shape inference. Developers are hence freed from tedious manual placeholder and variable definition, as they only need to specify type and shape of input spaces to an algorithm's outermost container component (root compo-

---

[*]Equal contribution [1]University of Cambridge [2]rlcore.ai [3]Helmut-Schmidt-University. Correspondence to: Michael Schaarschmidt <michael.schaarschmidt@cl.cam.ac.uk>.

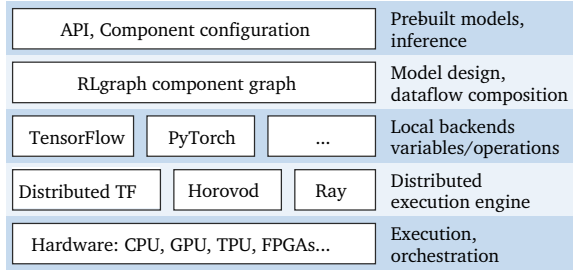|	
API, Component configuration	Prebuilt models, inference
RLgraph component graph	Model design, dataflow composition
TensorFlow · PyTorch · ...	Local backends variables/operations
Distributed TF · Horovod · Ray	Distributed execution engine
Hardware: CPU, GPU, TPU, FPGAs...	Execution, orchestration

Figure 1. RLgraph stack for using and designing RL algorithms.

nent). They can then rely on the RLgraph utilities to handle most aspects of connecting complex models, e.g. splitting and merging complex nested spaces. A resource aware graph executor expands the meta graph to add operations for local and distributed device strategies, e.g. by creating subgraph replicas for GPUs. At runtime, the graph executor (e.g. for TensorFlow) serves requests to the agent API by determining relevant input placeholders and operations from the op-registry and batching together all relevant operations into a single session call. Our design provides numerous advantages over many existing libraries:

1. **Distributed execution.** By separating concerns of design and execution, resulting agents can be implemented towards any distributed execution paradigm, e.g. using distributed TensorFlow (Abadi et al., 2016), Ray (Moritz et al., 2017), and plugins such as Uber's Horovod (Sergeev & Balso, 2018).

2. **Static and define-by-run backends.** The meta graph design does not impose restrictions on its own execution. This means it can not only support end-to-end static graphs including control flow (Yu et al., 2018), but also enables define-by-run semantics (e.g. PyTorch (Paszke et al., 2017)) through a unified interface.

3. **Fast development cycles and testing.** The separation of component composition and variable/op creation drastically simplifies graph creation, component reuse, and testing. RLgraph's space utilities are particularly helpful as they can nest, split and merge complex spaces across batch and time ranks, which normally requires tedious manual adjustments. As a result, users can focus on algorithm logic during development.

In the remainder of the paper, we first analyze design problems in RL and survey existing libraries (§2). Following this motivation, we discuss the design of RLgraph (§3, §4). In the evaluation, we compare RLgraph against reference implementations using different execution paradigms (§5). Our results show RLgraph can improve sample throughput over existing implementations by up to 180%. In related work, we discuss emerging approaches in programming

models and optimization (§6). RLgraph is available as open source[1].

2 MOTIVATION

2.1 RL workloads

The central difficulty of executing RL workloads lies in the need for frequent interaction with the problem environment during training to evaluate and update the model. Environments may take the form of expensive physical systems (robots), 3D scene simulators, games, or generally any system exposing a state representation and an action interface. This is in contrast to supervised workloads, where training data is typically entirely available in advance, thus enabling straightforward batching and synchronization strategies (Sergeev & Balso, 2018). As a consequence of fast moving and empirically driven research, RL algorithms vary across all dimensions of execution (recently discussed by Liang et al. (Liang et al., 2018)).

State management. Sample trajectories are often collected in a distributed fashion where workers interact with dedicated (simulation) environment copies. Algorithms manage synchronization of model weights between one or multiple learners and sample collectors, employing synchronous and asynchronous strategies. In addition, they must process and transmit samples to learners efficiently, sometimes involving hierarchies of local and distributed shared buffers to split post-processing tasks (Horgan et al., 2018).

Resource requirements and scale. Recent successes in applying RL at scale in gaming (e.g. OpenAI Five DOTA (OpenAI, 2018), AlphaGo (Silver et al., 2017)) were enabled by training models on up to tens of thousands of CPU cores and hundreds of GPUs. In contrast, models for environments which are not easily parallelized may be executed on single CPUs, but might have stringent latency requirements.

Models and optimization strategies. Neural networks used to represent policies range from small multi-layer perceptrons to complex hierarchical representations (Silver et al., 2016; Wayne et al., 2018). Learning approaches vary from small incremental updates to expensive but infrequent policy optimizations over large batches, making effective use of hardware accelerators difficult.

2.2 Existing abstractions

Reference implementations. Many libraries primarily serve as reference implementations to help reproduce research results. For example, **OpenAI baselines** (Sidor & Schulman, 2017) and Google's **Dopamine** (Bellemare et al., 2018) provide collections of well-tuned algorithms on

---

[1] https://github.com/rlgraph/rlgraph

benchmarks such as OpenAI gym (Brockman et al., 2016) or ALE (Bellemare et al., 2013). **Nervana Coach** (Caspi et al., 2017) contains a similar collection but with added tools for visualizing progress, and facilities for hierarchical learning and distributed training. **TensorFlow Agents** (Hafner et al., 2017) contains a single well-tuned implementation of a state-of-the-art algorithm (proximal policy optimization (PPO) (Schulman et al., 2017)) where all control flow and environment stepping have been merged into one end-to-end TensorFlow graph. Reference implementations share some components between algorithms (e.g. network architectures) but typically ignore many practical considerations in favour of concise code. Retooling them to different execution modes, environment semantics, or device strategies (e.g. multi-GPU support) requires significant work due to hard-coded, tightly coupled designs.

Centralized control. Ray RLlib (Liang et al., 2018) defines a set of abstractions for scalable RL. It relies on Ray's actor model (Moritz et al., 2017) to execute RL algorithms via centralized control. At the core of RLlib's hierarchical task parallelism approach lies a set of optimizer classes. Each optimizer implements a *step()* function which distributes sampling to remote actors, manages buffers, and updates weights. For example, an *AsyncReplayOptimizer* implements distributed prioritized experience replay (Horgan et al., 2018). Each step, the optimizer loop pulls samples from actors, inserts them into replay buffers, and performs training on an asynchronous learner thread.

The core claim of RLlib is the separation of the execution plane in the optimizer from the definition of the RL algorithm within a policy graph. However, each optimizer encapsulates both local and distributed device execution. This means for example that only the dedicated multi-gpu optimizer class supports splitting input batches synchronously over multiple GPUs. RLgraph solves this issue by not driving control flow from the optimizer, but via a graph executor, which can impose any device strategy onto the metagraph, irrespective of distributed coordination.

Another disadvantage of optimizer driven control flow is that RLlib mixes Python control flow, Ray calls, and TensorFlow calls in its components. Algorithms implemented in RLlib are hence not easily portable as training is principally meant to be executed only on Ray. In contrast, RLgraph can construct end-to-end computation graphs including in-graph control-flow (depending on the backend), then distribute them via Ray, distributed TF, or any other mechanism.

Fixed end-to-end graphs. TensorForce (Schaarschmidt et al., 2018) is a TF library providing a declarative interface to a number of RL algorithms. TensorForce focuses on applied use cases where control flow is driven by external application contexts, not simulation environments. TensorForce's design disadvantage stems from a tightly coupled

component design within an end-to-end TF graph. Moving control flow into the graph accelerates execution by avoiding unneeded context switches between Python interpreter and TF runtime (Yu et al., 2018). However, TensorForce's object design, in which all models inherit along a single hierarchy and must conform to its custom interface, makes composing new models or retooling execution difficult.

3 FRAMEWORK DESIGN

3.1 Design principles

In the absence of a single dominant design pattern, frameworks must resolve the tension between flexible prototyping, reusable components, and scalable execution mechanisms. RLgraph's design is driven by a number of insights:

Separating algorithms and execution. RL algorithms require complex control flow to coordinate distributed state and sample collection on one hand, and internal training logic on the other hand. Separating these aspects is difficult but essential to avoid re-implementing execution strategies. RLgraph manages local execution via graph executors which expose a clear interface between a high level API and the component graph. Distributed coordination is delegated to dedicated distributed executors (e.g. on Ray), or as part of the graph build in the executor (distributed TF).

Shared components with strict interfaces. Deep learning frameworks enable quick prototyping of neural networks by exposing APIs to combine different types of layers with compatible interfaces. Providing a similar set of interchangeable components towards RL is complicated by the multitude of learning and execution semantics. This is exacerbated by implementations containing definitions in multiple execution contexts, e.g. Python control flow interleaved with calls to TF runtime. Tight coupling of components, and in turn a lack of well-defined interfaces and component boundaries, means that re-usability is severely constrained. RLgraph addresses this problem via its component meta graph. Components may only interact via declared API methods, which offer various utilities for defining dataflow.

Sub-graph testing. An undesirable consequence of incorporating stochastic approximations at all levels are numerical sensitivity and non-determinism (Nagarajan et al., 2018). RL algorithms can require an overwhelming number of hyperparameters (often in excess of 25). This has created severe issues for robustness and reproducibility (Henderson et al., 2017; Mania et al., 2018). Implementations are notoriously difficult to debug and test in part because generating and verifying inputs and outputs of partial dataflow is tedious (e.g. manually creating tensors of required shapes). RLgraph enables sub-graph testing by allowing users to send example data from input spaces through arbitrary components and component combinations.

3.2 Components and meta graph

Components. Next, we discuss the design of RLgraph's component graph. For simplicity, we use TensorFlow as the primary backend and describe the implementation of other backends (e.g. PyTorch) in §4.2. RLgraph's core abstraction is the *Component* class which encapsulates arbitrary computations via *graph functions*. RLgraph components are conceptually similar to DeepMind Sonnet's components (DeepMind, 2017) but offer more advanced notions of composition. Consider a replay buffer component which exposes
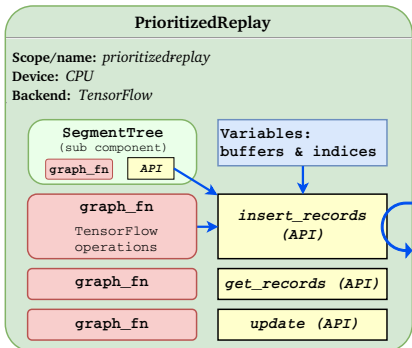
Figure 2. Example memory component with three API methods.

functionality to insert experiences and sample mini-batches according to priority weights. Implementing this buffer in an imperative language such as Python is straight-forward, but including it as part of a TensorFlow graph requires creating and managing many variables through control flow operators (e.g. to update priorities). Composing multiple such components in a re-usable way is difficult due to an impedance mismatch between class-based programming in a driver language, and functional transformations within a dataflow graph. Using a define-by-run framework (e.g. PyTorch) eases development but can create difficulties in large scale distributed execution and program export.

Existing high-level APIs for neural networks such as Sonnet, (DeepMind, 2017), Keras (Chollet et al., 2015), Gluon (Rochel et al., 2018), or TF.Learn (Tang, 2016) focus on assembly and training of neural networks. Implementing RL workloads in these frameworks usually means mixing imperative Python control flow with deep learning graph objects, leading to the design issues discussed before.

When building for a static graph backend, RLgraph's component API enables fast composition of end-to-end differentiable dataflow graphs with in-graph control flow. The graph builder and executor automatically manage burdensome tasks such as variable and placeholder creation, scopes, input spaces, and device assignments.

Example component. In figure 2, we show a simplified prioritized replay buffer component. All components inherit from the generic *Component* class and assemble logic by

combining their own sub-components. The buffer has a segment tree sub-component to manage priority orders. It exposes API methods to insert, sample, and update priorities which under the hood map to three graph functions. The difference between simple object methods and RLgraph API methods is that registered API methods are identified and managed in the build. Input shapes can be inferred automatically via dataflow from inputs to the root component.

Developers can declare methods as API methods by calling a register function (or in the future using a decorator). Technically, not all functionality of a component needs to be registered as an API method. Users can also implement helper functions or utilities, e.g. using TensorFlow operations without including them as API methods, if they do not need to be called from external components. Implementing such utilities as RLgraph components with API methods is nonetheless useful because these features can then be built and tested as sub-graphs.

Components can call arbitrary sub-components (and their sub-components). Soft restrictions in creating edges between components exist with regard to input-completeness. A component may have multiple API methods where input spaces to one method depend on outputs of its other methods. In strict build mode, this is a violation because the component cannot be built safely before knowing all its spaces. These violations point to problematic component-design. However, users can turn off strict input checks for specific methods.

3.3 Building component graphs

RLgraph models are assembled in three distinct phases:

1. **Component composition phase** in which component objects are defined and combined, including arbitrary nesting of sub-components.

2. **Assembly phase** in which a type- and dimension-less dataflow graph (the meta-graph) is created. This is achieved by calling each of the root component's API methods once to traverse its call graph. The API-methods of the root component define the externally visible API of the meta graph.

3. **Graph compilation/building phase** in which all computation operations are defined for each component. Inside a component class, these definitions are placed in special functions called *graph functions*. Graph functions are the only places in the code where backend dependent objects are used (e.g. TF ops).

There are also intermittent sub-phases for the initialization of execution aspects (e.g. session management) which are explained in §4. A wide range of off-the-shelf component

implementations such as buffers, optimizers, neural networks, or nested space splitters and mergers means that most users will only need to define few components to prototype new algorithms (e.g. loss function, network architecture).

```python
1  def init(self, *args, **kwargs):
2    self.register_api("update", self.update)
3
4  def update(self, batch_size):
5    sample = self.call(memory.sample, batch_size)
6    s, a, r, next_s, t = self.call(splitter.split, sample)
7    loss = self.call(dqn_loss.get_loss, s, a, r, next_s, t)
8    variables = self.call(policy.get_variables)
9    update = self.call(optimizer.step, loss, variables)
10   return update, loss
11
12 # Envisioned API: Implicit 'call' via decorators.
13 # Use space-hints to auto-split, merge nested spaces.
14 @rlgraph.api_method(split=True)
15 def observe(records)
16   self.memory.insert(records)
```

Listing 1. Example API methods show framework-independent control-flow with implicit and explicit use of *call*.

1. Component composition and nesting. All components needed by a model are defined as Python objects. Components are logically organized as nested subcomponents of a root container component, which exposes the external API. Note that an agent can in theory define multiple root components to act and learn different policies in parallel.

2. Assembling the meta-graph. Next, users define the dataflow through the model via connections between components. Neither data types nor shape information (i.e. tensor rank and dimensions) are necessary at this stage. Each component comes with a set of API-methods (Listing 1). The data (tensors) are interpreted inside these methods as abstract meta-graph operator objects, and their shapes and types will be inferred at build time. In the prototype implementation we explicitly use the component *call* method for all API invocations. We are currently implementing a more user-friendly decorator version (bottom example).

Return values of an API call can now be passed into other API-methods, a sub-component's API-method, or into a graph function for numerical manipulation. RLgraph spaces can conveniently nest, merge, split and fold time and batch dimensions of tensors through components. In our experience, these utilities drastically reduce development times as the different build phases automatically detect problems when manipulating complex spaces, e.g. records containing multiple states and actions with batch and time dimensions.

A simplified meta graph assembly procedure is shown in Algorithm 1. The root component exposing the external interface (e.g. *act*, *observe*, *update*) and the input spaces for the external API are passed to the meta graph builder. This builder generates the backend-independent dataflow graph and the API by iterating over all API methods defined

---

Algorithm 1 Meta graph build procedure

Input: component $root$, input_spaces $spaces$
$api = dict()$
// Call all api methods once, generate op columns.
for $method, record$ **in** $root.api$ **do**
  $in\_ops\_records = list()$
  // Create one input record per API input param.
  for $param$ **in** $record.input\_args$ **do**
    $in\_ops\_records.append(Op(param.space))$
  end for
  // Traverse graph from root for this method.
  $out\_ops\_records = method(in\_ops\_records)$
  // Register method with graph inputs and output ops.
  $api[method] = [in\_ops\_records, out\_ops\_records]$
end for
return MetaGraph(root, api)

---

in the root component. For each method, a meta graph op is created for each of its parameters and looked up in the input graph (type checks and default argument handling omitted). The component graph is then traversed by calling *call* which infers parameters and return values for each API called through the graph, and these are stored as records in the meta graph. Finally, the API method is registered in an API registry which contains the input spaces and final output ops (identified through the traversal).

3. Building computation graphs. All numerical operations occur in the third phase inside graph functions which implement each component's API. Operations defined in a graph function include for example defining loss functions, or sampling from a buffer. As sometimes the component's variables must be accessed to complete these operations (e.g. a neural network layer must read its weights), RLgraph ensures that any such computation function is only called after all variables of a component have been defined. For example, the memory component in Fig. 2 can only define its buffers (e.g. TensorFlow variables) once it receives shapes and types of buffer contents.

This barrier is enforced during the build, and custom components only need to override a generic method for variable creation. The method is called automatically and receives types and shapes of variables as input arguments. Developers thus only need to specify the external input spaces to the program (e.g. int/float boxes with batch and time ranks, container spaces to describe nested data). In practice, these are primarily state and action layouts.

We briefly describe the intuition behind the main build algorithm. The build begins by calling all API methods defined at the root from the provided input spaces until a component is input-complete, i.e. all spaces for all its computations are available. It then executes a completion function which

calls the component's *create_variables*, and subsequently its graph functions under the correct device and scope to define its operations. Once a component is complete, the outputs of its graph functions become available as input spaces for subsequent components/graph functions. We then simply perform breadth-first-search until there are no more components to build or a constraint violation is detected. Input placeholders and op names are created and stored for all ops and output combinations defined in the API.

```
1  state_space = FloatBox(shape=(64,), add_batch_rank=True,
2                          add_time_rank=True)
3  # Dict space: 1 discrete, 1 continuous action.
4  action_space = Dict(discrete=IntBox(),
5                       cont=FloatBox(), add_batch_rank=True)
6  policy = Policy("recurrent_policy.json", action_space)
7
8  # Construct sub graph from spaces, auto-gen placeholders.
9  test = ComponentTest(policy, dict(nn_input=state_space),
10        action_space=action_space))
11  # Test with any inputs in the input space.
12  action = test.test("get_action", state_space.sample())
```

Listing 2. Testing sub-graphs from arbitrary spaces.

Testing sub-graphs. Consider the RLgraph test class example in Listing 2. Here, we build a *Policy* component (with subcomponents for a recurrent network and action selection) for the specified state and action spaces (with options for batch and time ranks). The test helper builds the sub-graph for the policy via the phases describe above. Users can then run the test and call an API method (e.g. by sampling an input from the input space). This call is delegated to a graph executor which retrieves and executes the corresponding op.

3.4 Agent API

Pre-built models can be configured via declarative configurations similar to TensorForce (Schaarschmidt et al., 2018). Configurations are provided as e.g. JSON documents specifying an algorithm and its components (network with list of layers, buffer, optimizers, device strategy etc.). The agent interface defines a set of abstract methods which agents must support to access certain execution modes (Listing 3).

The main difference between RLgraph and existing APIs lies in strictly enforced component boundaries and more explicit execution semantics. Fine-grained device control is managed via a device map where each components operations and variables can be assigned separately and selectively. Components may only exchange data along edges of the component graph where an edge corresponds to a call to a declared API method. This ensures well-defined APIs and also avoids a common case where two components are implemented to always be used together, thus making individual reuse difficult. In RLgraph, all components can be used and built individually from any input spaces. Next, we discuss graph execution.

```
1  abstract class rlgraph.agent:
2    # Build with default devices, variable sharing, ..
3    def build(options)
4    def get_actions(states, explore=True, preprocess=True)
5    # Update from internal buffer or external data.
6    def update(batch=None)
7    # Observe samples for named environments.
8    def observe(state, action, reward, terminal, env_id)
9    def get_weights, def set_weights
10   def import_model, def export_model
```

Listing 3. High level agent API.

4 EXECUTING GRAPHS

4.1 Graph executors.

All build phases are called from a *graph executor* which serves as the execution bridge between the component graph and a backend framework. Graph executors expose an *execute()* method which takes name and arguments of an API method and returns the result. They further manage any backend-specific initialization, monitoring, and devices. For example, the TensorFlow executor assumes the following tasks. First, it initializes the TensorFlow session and variables, and builds hooks for summaries or profiling. For operation execution, it fetches input placeholders and op names from the graph operation registry and assembles session inputs. Importantly, there is no other interaction between user programs and graph other than through API operations defined in the root component.

Device management. Graph executors also handle device assignments by interleaving the build with a phase to initialize device strategies. Upon initialization, local device information (e.g. CUDA visible devices) is read and compared against user-defined device maps or synchronization modes. Consider a synchronous multi-GPU strategy where each input batch is split to one graph copy per GPU, with gradients averaged for the final update. Managing this synchronization requires the creation of additional operations and variables for the device copies and the splitting and averaging logic. The graph executor does this by creating core component copies and connecting them through generic input space splitters before building the meta graph. Since no variables or placeholders have been created at this stage, components or sub-graphs can straightforwardly be modified, replaced or extended. In addition to generic device strategies, users can define a device map which specifies a device assignment for each component's ops and variables.

Distributed execution. RLgraph can be executed in distributed mode either with framework-specific mechanisms or by using any other means to create and synchronize agents. For example, when using distributed TensorFlow, the TF graph executor can create the necessary parameter server and global/local synchronization operations. It can also plug-in third party tools such as Uber's Horovod to
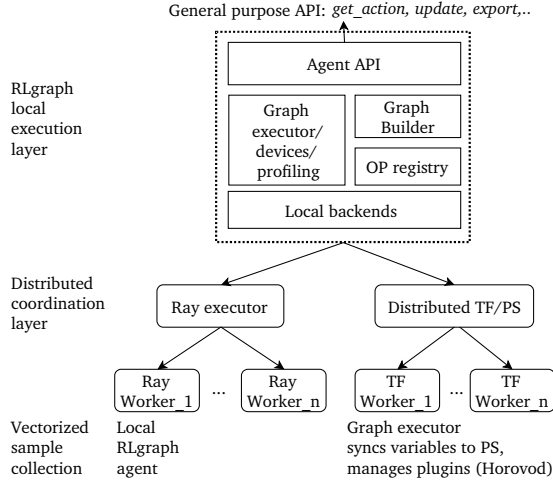
General purpose API: *get_action, update, export,..*

Figure 3. RLgraph execution stack.

assume specific aspects of distributed communication for a backend (e.g. ring all-reduce (Sergeev & Balso, 2018)). To demonstrate RLgraph's flexibility, we also built a Ray executor which can execute arbitrary RLgraph implementations on Ray's centralized execution model (Moritz et al., 2017). In the evaluation, we show that our implementation outperforms Ray's native library RLlib (Liang et al., 2018).

Figure 3 illustrates how local execution and distributed coordination are separated. When using Ray, we simply pass an agent configuration to our Ray executor which creates Ray workers, each locally generating their component graph and graph executor to interact with environments. When using non-centralized control (e.g. distributed TensorFlow), the graph executor creates the necessary session and server objects and handles parameter server synchronization on each worker. Common features such as fault tolerance are delegated to the underlying execution engines via a configuration interpreted by the respective graph executors.

4.2 Backend support and code generation

Define-by-run graphs. While there are many advantages to defining models as end-to-end computation graphs, define-by-run semantics and eager execution are increasingly popular for ease of use. RLgraph's component graph can be executed both in static graph construction or define-by-run mode. To implement a PyTorch backend, we only had to modify the build procedure as follows. As there are no placeholders, we simply create torch tensors during the build phase as artificial placeholders to push through the dataflow graph for shape and type inference of variables (e.g. tensors used to store state). This is because even in define-by-run mode, automatically dealing with nesting and splitting of complex spaces across time and batch dimensions can be handled via space utilities. Next, after building the meta graph, we change the execution mode flag for API methods from 'build' to 'define-by-run'. In this mode, instead

of returning operation objects used for graph construction, RLgraph simply directly evaluates a call-chain of graph functions to retrieve the result. RLgraph hence provides a unified interface for executing its component graph API in define-by-run and static-graph mode.

Autograph and graph optimization. An emerging but early trend in deep learning frameworks are prototype features to automatically convert imperative code to static computation graphs. Examples of this include TensorFlow's AutoGraph (Google Inc., 2018) mechanism and PyTorch's JIT tracing. We encountered (likely temporary) limitations e.g. in TensorFlow on stateful operations which would need to convert list manipulation to TF variable updates. In the future, we plan to merge backend-dependent component implementations into single-stream Python functions, which will then be auto-converted via such features.

RLgraph's separation of concerns also opens up opportunities for optimization at all stages. Emerging approaches in optimizing execution (e.g. via automated device placement (Mirhoseini et al., 2018)), or backend-dependent compilations can be integrated at the graph executor stage. As the graph is assembled in one place through well-defined phases, including external tools at each stage is straight-forward.

5 EVALUATION

We evaluate RLgraph's performance using different distributed execution engines, local deep learning backends, and device strategies. Our aim is not to benchmark the underlying frameworks but to show RLgraph can perform competitively compared to native implementations.

5.1 Results

Build overhead. We begin by comparing one-time build overhead of RLgraph's abstractions on different backends. Recall there are two sources of overhead during initialization. First, the component meta graph is created by traversing the graph from the root component. Second, creating variables and potentially static graph operations by letting input spaces flow through the meta graph requires moving through components in the iterative build procedure.

In Figure 4a, we show meta graph and main build overhead on a single memory component and a common RL architecture (DQN) for the TensorFlow (TF) (v1.11) and PyTorch (PT) (v0.4) backends. Overhead here refers to the time spent on top of creating variables and operations, which would have to be done irrespective of using RLgraph. The overhead for both build phases to build a single component (e.g. for modular performance testing and debugging) is less than 100 ms. For a common architecture (dueling DQN with prioritized replay, 43 components), the combined overhead is about 1 s for TF and 650 ms for PT. PyTorch builds

only require few milliseconds as variables (e.g. to represent memory state) are simple Python lists or numpy arrays.
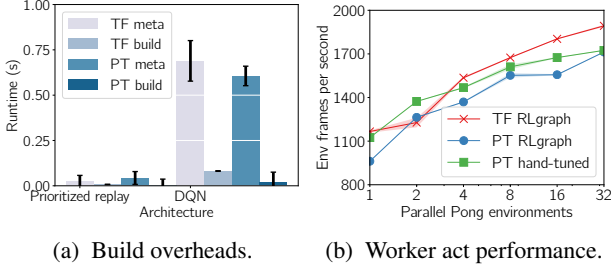

(a) Build overheads. (b) Worker act performance.

Figure 4. TensorFlow and PyTorch backend comparison.

We also compare backend runtime performance by testing act (inference) throughput on a single-threaded worker acting on a vector of environments (Fig. 4b). We use the Atari Pong environment and a standard 3-layer convolutional architecture followed by a dueling network for action selection. The TF backend (*TF RLgraph*) does not incur runtime overhead because the component graph is discarded after building. The graph executor simply looks up the name of the operation in the op registry and executes the corresponding TF op. In define-by-run mode using PyTorch (*PT RLgraph*), RLgraph incurs some overhead when calls are routed through components. To understand this overhead, we also implemented a bare-bones PyTorch actor, including fine-tuning OpenMP and MKL settings (*PT hand-tuned*).

TensorFlow outperforms both PyTorch variants as batch-size increases on a CPU, thus making it more suitable for performing batch-acting and batch-postprocessing. RLgraph's PT backend carries overhead due to requiring additional lookups when traversing the graph via component "calls". This overhead becomes negligible as batch size increases and runtime is dominated by the network forward passes. To reduce this overhead when traversing the component graph, we implemented some initial fast-path calls. For some cases, the graph builder can identify edge-contractions (where calls are edges and components are vertices), so define-by-run execution through the relevant sub-graph requires no intermediate component calls.

To the best of our knowledge, RLgraph represents the first common interface to TF and PT on a component level. High-level APIs like Keras only support static graph approaches. Libraries such as Ray RLlib support TF/PT backends but only on an algorithm level where entire implementations can be parallelized via RLlib's distributed abstractions. Using RLgraph, developers assemble logic via the backend-independent API, and can then implement e.g. a new loss component in their backend of choice.

Distributed execution on Ray. Next, we evaluate RLgraph on the distributed execution engine Ray in comparison to Ray's naive library, RLlib. We implemented distributed pri-

oritized experience replay (Ape-X (Horgan et al., 2018)), a state-of-the-art distributed Q-learning algorithm on our Ray executor. We further implemented a vectorized environment worker for sample collection, including all heuristics described in the Ape-X paper and found in RLlib's implementation (e.g. worker-side prioritization).

All experiments were performed on Google Cloud with the learner being hosted on a GPU instance with 1 active V100 GPU, 24 vCPUs and 104 GiB RAM. Sample collection nodes had 64 vCPUs and 256 GiB RAM. Figure 5 shows
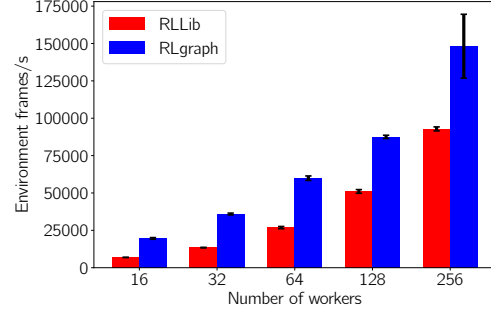

Figure 5. Distributed sample throughput on *Pong*.

sampling performance on the Pong environment. The x-axis represents the number of policy-evaluators/Ray-workers respectively (RLlib, RLgraph), each initialized with a single CPU, and y-axis shows environment frames per second (including frame skips). Each worker executed 4 environments, and we used 4 instances of replay memories to feed the learner (we did not observe improvements using more memories due to limited vCPUs on GPU nodes). All settings were ran with 8 sample nodes except 256 workers (16 sample nodes) to ensure sufficient memory. RLgraph outperforms RLlib by a large margin (185% on 16, 60% on 256 workers) despite implementing the same algorithm with equivalent hyper-parameters, model size, and environments. Performance for 16 workers is highest due to all tasks fitting on a single node, and because RLgraph also completes its learning tasks faster due to improved implementations.

The reason for RLgraph's better performance is systematic component analysis yielding insights into efficient sample processing. For example, RLlib's policy evaluators execute multiple session calls to incrementally post-process batches. RLgraph instead splits post-processing in incremental and batched parts to minimize calls to the TensorFlow runtime. These effects can also be observed at the scale of a single task across different task lengths and number of environments (called sequentially). Figure 6a shows the requested number of samples versus the achieved frames per second using a single RayWorker (RLgraph) versus a policy evaluator (RLlib). Both use the same agent and configuration as in the distributed setting (10 warm up runs, mean across 50 runs). RLgraph is not only more effective on a single en-
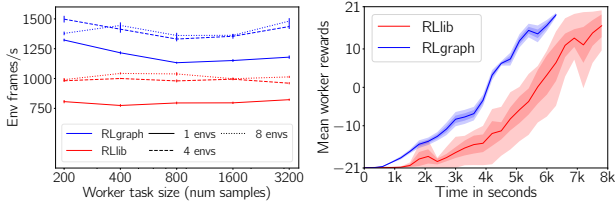
(a) Single worker throughput. (b) Training times for *Pong*.

Figure 6. Single task throughput and learning comparison.

Figure 7. RLgraph synchronous multi-GPU device strategy

vironment, it also scales better on vectorized environments due to faster accounting across environments and episodes.

We show learning results (Fig. 6b) to confirm that RLgraph's throughput is not at the cost of training performance. We use RLlib's provided tuned *Pong* configuration (32 workers). In reinforcement learning, the same code using the same hyper-parameters can vary drastically across runs, so reliably comparing learning is difficult (Henderson et al., 2017). We ran 10 random seeds and average across the 3 best runs (both libraries did not learn anything for some seeds as expected). In line with throughput, RLgraph learns to solve (reward 21) Pong substantially faster than RLlib.

Note that RLlib's published results on Ape-X throughput do not include updating without stating this explicitly, and later reported results including updates[2] are up to 130 k frames per second on 256 workers (versus 170 k max for RLgraph). Some performance differences may be attributed to different hardware setup (fewer CPUs on head node, GCP GPU latency). During experimentation, we experienced difficulties with the Ray engine. Actor tasks terminated without specific errors, and the object store ran out of memory both for RLlib and us on large clusters. Due to cost, we could not fully investigate these issues beyond tuning object store settings. Experimental versions of Ray's new backend include improved garbage collection of which RLgraph would benefit to the same extent as RLlib.

Overall, our results show RLgraph's advantage both on a single-task and distributed level. While RLlib could adopt more efficient implementations to match our results, our main insight is that RLgraph can be used on Ray via simple wrapper classes. Implementing other distributed semantics on Ray with RLgraph only requires extending our generic Ray executor to implement a coordination strategy.

Multi-GPU strategies. We also implemented device strategy prototypes where the meta graph is automatically expanded during the build phase (e.g. to create sub-graph replicas). When using a synchronous GPU replica strategy, the update batch is internally split into multiple sub-batches, and gradients are averaged across towers. Fig. 7 contains Ape-X results using 1 and 2 V100 GPUs. We observe the
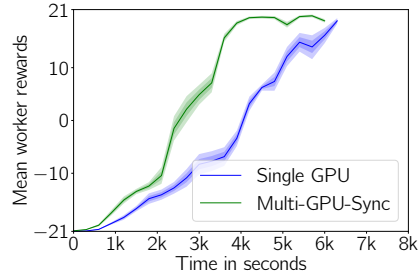
expected speed-up in convergence. We did not compare to RLlib as its multi-gpu optimizer does not support Ape-X semantics, we make no claim regarding its performance.

Distributed TensorFlow. Finally, we evaluate RLgraph using the distributed TensorFlow backend on DeepMind's (DM) importance-weighted actor-learner architecture (IM-PALA) (Espeholt et al., 2018). The authors have open-sourced an optimized implementation[3]. IMPALA perhaps best represents the end-to-end computation graph paradigm, where even environment interaction is fused into the TF graph. We implemented IMPALA in RLgraph to demonstrate its ability to generate such graphs. To this end, RLgraph provides generic execution components for graph-fused environment stepping. IMPALA executes updates by letting each actor perform a rollout step (100 samples) and input its samples into a globally shared blocking queue. The learner dequeues rollouts and uses a staging area to hide GPU latency. DM's implementation is not expected to
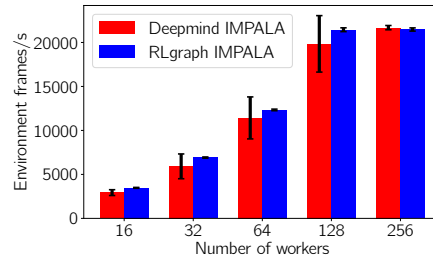
Figure 8. IMPALA throughput comparison on *seekavoid_arena_01*

match published results due to missing custom TF ops for environment interaction. Figure 8 compares throughput using the large network described in the paper on a DM lab 3D task (which are more expensive to render than Atari tasks). We again use a single V100 GPU for the learner and let 4 workers share a 8 vCPU instance. RLgraph achieves about 10-15% better mean throughput (5 runs) for fewer workers until both implementations are limited by updates. DM's implementation exhibited higher variance due to subtle differences in preprocessing tensors after unstaging. DM's code also carried out unneeded variable assignments in the

---

actor. Removing these yielded 20% improvement in a single-worker setting for RLgraph. Emerging tools in graph optimization (e.g. TF XLA, TVM (Chen et al., 2018)) can help with optimizing data layouts and numerical transformations. RLgraph helps most in improving high level dataflow as it enforces well-defined interactions between components (c.f. supplementary material). This in turn improves reasoning about the complex design patterns found in RL algorithms.

5.2 Discussion

Our results show that using RLgraph to define algorithms via combining backend-independent components yields high-performing implementations across backends. Assembling the graph through the different build phases only adds up to one second of build overhead. Using Ray as a distributed backend, we demonstrated that RLgraph run on its own Ray executor outperforms Ray RLlib. This indicates that wide-spread mixed backend (e.g. Python/TF) RL implementations can be significantly accelerated via careful dataflow analysis, and without changing algorithm logic. Using distributed TF, we found that RLgraph can help improve dataflow in end-to-end static computation graphs. Our main take-away is that developers can use RLgraph to focus on logical component composition in RL independently of the underlying execution paradigm.

6 RELATED WORK

RLgraph builds upon the experiences of many prior libraries which we discussed in §2.2. Here, we discuss emerging trends in programming models and optimizations.

6.1 Programming models

The success of deep learning frameworks has given rise to several higher-level learning APIs seeking to free users from dealing with lower level tensor operations. Keras (Chollet et al., 2015) is a popular framework for quick assembly and training of deep learning models with support for multiple static-graph backends (e.g. TensorFlow (Abadi et al., 2016), CNTK (Seide & Agarwal, 2016), MXNet (Chen et al., 2015)). Gluon provides a concise API for imperative, dynamic neural networks on top of MXNet (Rochel et al., 2018). TF.Learn offers a high level API for constructing symbolic TensorFlow graphs (Tang, 2016). Among these frameworks, TensorFlow's programming model (Abadi et al., 2017) is distinct because it supports in-graph control-flow (Yu et al., 2018). A key issue in the high-level APIs above is that they typically assume control-flow to be implemented in the driver language (frequently Python). RLgraph bridges this gap by providing Sonnet-style (DeepMind, 2017) composable components, with the addition of API methods handling in-graph control flow, or define-by-run graph execution.

6.2 Graph optimizations

Programming models for machine learning and in particular deep learning typically prioritize high level APIs and usability over performance. Code is written in a multitude of languages and libraries, and is executed over a variety of backends, e.g. prototyped on a personal laptop, and deployed on a cluster of GPUs. Frameworks like Weld (Palkar et al., 2017) optimize performance by integrating library calls into a common intermediate representation which is then mapped to efficient multi-threaded code. Mirhoseini et al. identify TensorFlow effective graph device placements via hierarchical reinforcement learning (Mirhoseini et al., 2018). FlexFlow further improves parallelization strategies for specific deployments using an execution simulator and fine-grained randomized search across different execution dimensions (Jia et al., 2018). TVM is a compiler stack to optimize tensor operations across diverse hardware backends (Chen et al., 2018). In general, RLgraph constructs component graphs irrespective of execution semantics. Graph optimizations can be performed at the level of graph executors by including them at the appropriate build stage.

6.3 Graph conversion and code generation

The proliferation of deep learning frameworks which often focus development efforts on Python front-ends has also created the need for new shared representations. The aim of standard formats such as ONNX (Facebook Inc., 2017) is to enable model interoperability, define a common route from prototyping to production deployment, and to create a shared runtime for optimizations. From a RL perspective, the aim of deployable graphs is more difficult to achieve due to the extensive control-flow and state management in RL workloads. Implementing code using e.g. TensorFlow control flow operators allows immediate export, optimization and deployment of the entire RL program but can slow down development. Framework developers have recognized this tension and are designing tools such as AutoGraph in TF for automatic graph generation. The TF for Swift project is an early stage example of integrating first-class language support for automatic graph program extraction from imperative programs. As these initiatives mature, we expect RLgraph components to merge currently backend-dependent graph functions by leveraging automatic graph generation.

7 CONCLUSION

RLgraph is a new open source framework for designing and executing computation graphs for reinforcement learning. RLgraph's meta graph abstraction allows developers to separate the composition of logical components in a RL algorithm from their local backend and distributed execution. The resulting implementations achieve state of the art performance across different distributed execution paradigms.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pp. 265–283, 2016.

Abadi, M., Isard, M., and Murray, D. G. A computational model for tensorflow: An introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2017, pp. 1–7, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5071-6. doi: 10.1145/3088525. 3088527. URL http://doi.acm.org/10.1145/3088525.3088527.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013.

Bellemare, M. G., Castro, P. S. C., Gelada, C., Kumar, S., and Moitra, S. Dopamine. https://github.com/google/dopamine, 2018.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

Caspi, I., Leibovich, G., and Novik, G. Reinforcement learning coach, December 2017. URL https://doi.org/10.5281/zenodo.1134899.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL http://arxiv.org/abs/1512.01274.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: An automated end-to-end optimizing compiler for deep learning. *http://arxiv.org/abs/1802.04799v2*, 2018.

Chollet, F. et al. Keras. https://keras.io, 2015.

DeepMind. Sonnet: TensorFlow-based neural network library. https://github.com/deepmind/sonnet, 2017.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018. URL http://arxiv.org/abs/1802.01561.

Facebook Inc. ONNX- Open Neural Network Exchange Format. https://onnx.ai, 2017.

Google Inc. AutoGraph. https://www.tensorflow.org/guide/autograph, 2018.

Hafner, D., Davidson, J., and Vanhoucke, V. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *CoRR*, abs/1709.02878, 2017. URL http://arxiv.org/abs/1709.02878.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2017. URL http://arxiv.org/abs/1709.06560.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., and Silver, D. Distributed prioritized experience replay. *CoRR*, abs/1803.00933, 2018. URL http://arxiv.org/abs/1803.00933.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *http://arxiv.org/pdf/1807.05358v1*, 2018.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. End-to-end training of deep visuomotor policies. *J. Mach. Learn. Res.*, 17(1):1334–1373, January 2016. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2946645.2946684.

Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., and Stoica, I. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pp. 3059–3068, 2018.

Mania, H., Guy, A., and Recht, B. Simple random search provides a competitive approach to reinforcement learning. *CoRR*, abs/1803.07055, 2018. URL http://arxiv.org/abs/1803.07055.

Mirhoseini, A., Goldie, A., Pham, H., Steiner, B., Le, Q. V., and Dean, J. Hierarchical planning for device placement. 2018. URL https://openreview.net/pdf?id=Hkc-TeZ0W.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017. URL http://arxiv.org/abs/1712.05889.

Nagarajan, P., Warnell, G., and Stone, P. The impact of nondeterminism on reproducibility in deep reinforcement learning. *https://arxiv.org/abs/1809.05676*, 2018.

OpenAI. OpenAI Five DOTA. Website, June 2018. URL https://blog.openai.com/openai-five/.

Palkar, S., Thomas, J. J., Shanbhag, A., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S., Zaharia, M., and InfoLab, S. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Rochel, S. et al. Gluon - A clear, concise, simple yet powerful and efficient API for deep learning. https://github.com/gluon-api/gluon-api, 2018.

Schaarschmidt, M., Kuhnle, A., Ellis, B., Fricke, K., Gessert, F., and Yoneki, E. Lift: Reinforcement learning in computer systems by learning from demonstrations. *CoRR*, abs/1808.07903, 2018. URL http://arxiv.org/abs/1707.06347.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Seide, F. and Agarwal, A. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2135–2135. ACM, 2016.

Sergeev, A. and Balso, M. D. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018. URL http://arxiv.org/abs/1802.05799.

Sidor, S. and Schulman, J. Openai baselines. website, 2017. URL https://blog.openai.com/openai-baselines-dqn/.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

Tang, Y. Tf.learn: Tensorflow's high-level module for distributed machine learning. *CoRR*, abs/1612.04251, 2016. URL http://arxiv.org/abs/1612.04251.

Tobin, J., Zaremba, W., and Abbeel, P. Domain randomization and generative models for robotic grasping. *CoRR*, abs/1710.06425, 2017. URL http://arxiv.org/abs/1710.06425.

Wayne, G., Hung, C., Amos, D., Mirza, M., Ahuja, A., Grabska-Barwinska, A., Rae, J. W., Mirowski, P., Leibo, J. Z., Santoro, A., Gemici, M., Reynolds, M., Harley, T., Abramson, J., Mohamed, S., Rezende, D. J., Saxton, D., Cain, A., Hillier, C., Silver, D., Kavukcuoglu, K., Botvinick, M., Hassabis, D., and Lillicrap, T. P. Unsupervised predictive memory in a goal-directed agent. *CoRR*, abs/1803.10760, 2018. URL http://arxiv.org/abs/1803.10760.

Yu, Y., Abadi, M., Barham, P., Brevdo, E., Burrows, M., Davis, A., Dean, J., Ghemawat, S., Harley, T., Hawkins, P., Isard, M., Kudlur, M., Monga, R., Murray, D., and Zheng, X. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pp. 18:1–18:15, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5584-1. doi: 10.1145/3190508.3190551. URL http://doi.acm.org/10.1145/3190508.3190551.

A   VISUALIZING COMPONENT GRAPHS

To illustrate how RLgraph helps organize computation graphs, we visualize the IMPALA implementation discussed in the evaluation.

Figure 9 illustrates the RLgraph implementation using TensorBoard. As each component's scope and variables are managed by RLgraph during the build, device assignments and dataflow are easy to visualize. Here, green components are on the GPU while blue components are on the CPU. The dataflow from bottom to top clearly illustrates how tensors are moved from the shared queue, preprocessed, then moved to a staging area. The prior batch is taken from the staging area, then preprocessed and passed to the policy and loss function. The optimizer and policy interact with a *Shared* scope as learner and workers share policy variables.

Mixed colours imply that a component has multiple sub-components on different devices. For example, the IMPALA loss-function computes an importance correction on the CPU as it is difficult to parallelize, so the loss-component has mixed assignments. We found that RLgraph specifically helps to make more effective use of existing tools like TensorBoard, as clean visualizations are key to identifying problems (e.g. with device assignments).

In Figures 10 and 11 (split due to size in TensorBoard), we show for comparison a visualization of DeepMind's open source IMPALA implementation. As is common for self-contained RL scripts, scopes, devices, and operation names are handled on an ad-hoc basis. The resulting graph, while implementing the same logic as our code, is highly fragmented in the visualization, making insights into potential problems much more difficult.
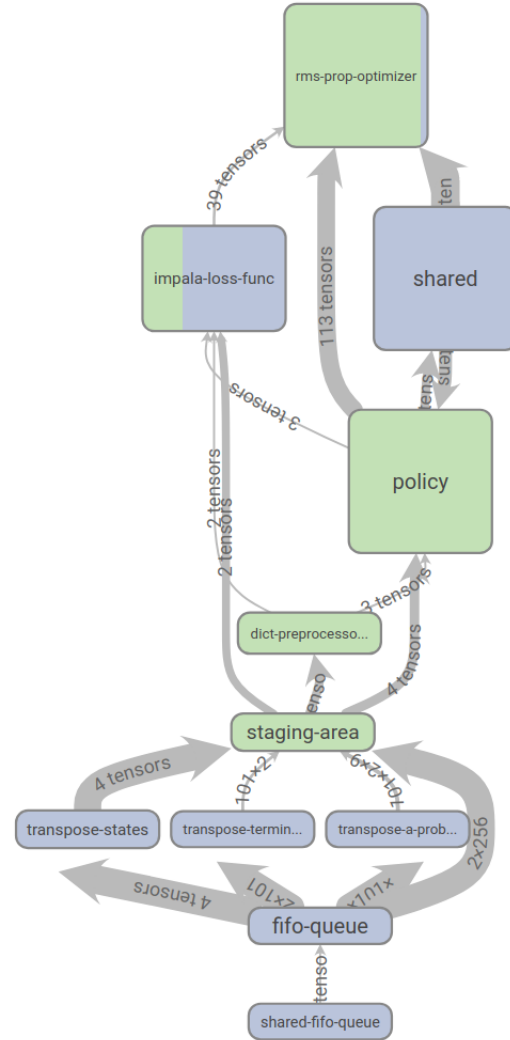

Figure 9. TensorBoard visualization of RLgraph's IMPALA learner. As all operations and variables are organized in components under separate scopes, dataflow between components is clear.

Figure 10. TensorBoard visualization of DeepMind's IMPALA learner (left).

Figure 11. TensorBoard visualization of DeepMind's IMPALA learner (right).