Replication of Experience Replay for Continual Learning

Anonymous Author(s)
Affiliation
Address
email

Abstract

In this work, we investigate the results reported in the paper "Experience Replay for Continual Learning" [Rolnick et al., 2018] through a replication study of the CLEAR method. More specifically, we review the contributions of this paper and report a detailed specification of our replication process for sequential task learning and implementing the CLEAR method for experience replay. Through our investigation, we found that the authors did not release a codebase of their contributions, and only the baseline, IMPALA [Espeholt et al., 2018], was available. We also found that the CLEAR method required many modifications to the baseline code with vague details not well described in the paper. To this end, we include our solutions to fixing the IMPALA codebase and how we adapted it for the CLEAR method. Lastly, we describe our attempts to implement the CLEAR method with and without behavioral cloning (Figure 2 in the original paper) and report our replicated graph of IMPALA for sequential task learning.

1 Introduction

When interacting with a complex environment, agents must be able to continually learn and adapt as task specifications change. One desiderata of an intelligent agent is that after it has learned how to perform one task, it should be able to perform more effectively in the future when exposed to similar tasks. However, one major issue in Deep Reinforcement Learning (RL) is the problem of catastrophic forgetting, in which an agent overrides previously learned information after being trained on a new task. Catastrophic forgetting has plagued Deep RL agents from transferring previously learned policies to new scenarios, making continual learning in RL a difficult hurdle to overcome. Simply put, catastrophic forgetting is a symptom of the agent adapting too quickly to new experiences, which destabilizes the learning process.

One common method of overcoming catastrophic forgetting is to teach the agent tasks in a simultaneous manner, which prevents the agent from forgetting critical information for any given task since the agent will be exposed to all the tasks consistently throughout training. However, while this method is feasible when access to simulated training data is abundant, it does not apply well in domains where computational resources for training data is limited and scarce, or when not all tasks are known beforehand, such as robotics. Therefore, the motivation of this paper is to enable Deep RL agents to leverage past experiences (experience replay) in order to overcome catastrophic forgetting when learning sequential tasks.

Rolnick et al. [2018] contributes a method for **C**ontinual **L**earning with **E**xperience **A**nd **R**eplay, termed the CLEAR method. The CLEAR method enables Deep RL agents to utilize both on-line learning methods to adapt new experiences into the learning process, while also using off-line learning through replay to stabilize learning and prevent catastrophic forgetting in sequential task learning scenarios. The authors also compare their method of implementing CLEAR with and without

Submitted to 33rd Conference on Neural Information Processing Systems (NeurIPS 2019). Do not distribute.

behavioral cloning, and report the results in Figure 2 of the original paper. The reason why this figure is of interest is because it demonstrates that even without behavioral cloning, the CLEAR method is able to do very well in overcoming catastrophic forgetting, making it an attractive method for enabling continual learning in Deep RL agents. It was this figure that we originally sought out to replicate and verify that the CLEAR method truly resolves the issue of catastrophic forgetting for continual learning. However, as noted in the following article, we encountered many issues in both fixing the original code base and implementing the CLEAR method as described by the authors. Therefore, our final results include our replication of Figure 1 in the original paper, where the baseline was ran on several different domains in a sequential manner. By replicating this figure, we demonstrate that we were able to fix the codebase to get the baseline code working, which is useful and necessary for any future researchers interested in replicating the CLEAR results.

2 The CLEAR Method

The CLEAR method [Rolnick et al., 2018] uses actor-critic training on a mixture of new and replayed experiences. In this actor-critic training method, there are many distributed actors each using different parameterized policies to choose actions for their given task. Each of these actors feed their experiences into a global buffer, which a single learner uses to update global parameters that the actors pull from. Because of this global buffer, the learner uses delayed experiences in the learning process, requiring an off-policy algorithm for integrating in the experiences from the actors.

For this actor-critic method, the value function is updated with an L2 loss, and the policy is regularized by reducing the entropy loss:

$$L_{value} := (V_\theta(h_s) - v_s)^2$$

$$L_{entropy} := \sum_a \pi_\theta(a|h_s) log \pi_\theta(a|h_s)$$

CLEAR uses the V-Trace off-policy learning algorithm [Espeholt et al., 2018] and adds two loss terms to induce behavioral cloning between the network and its past self.

$$L_{policy-cloning} := \sum_a \mu(a|h_s) log \frac{\mu(a|h_s)}{\pi_\theta(a|h_s)}$$

$$L_{value-cloning} := ||V_\theta(h_s) - V_{replay}(h_s)||_2^2$$

Note that the policy-cloning loss function reflects a minimization of the KL divergence between μ and π_{theta}, while the value-cloning loss attempts to minimize the L2 norm between the replay value function and the value function induced by the current policy θ.

3 Replication Procedures

For the DMLab experiments, Rolnick et al. [2018] modified the network in IMPALA [Espeholt et al., 2018], the source code for which is available at `github.com/deepmind/scalable_agent`. We also started from that codebase and made modifications to it in our attempt to replicate CLEAR.

3.1 Setup

3.1.1 Using the provided docker file

We started with building the scalable_agent repository using the docker file provided by the developers of the repositories. Details of our efforts and the errors faced for the first attempt are as follow:

- One of the pre-requisites for setting up the scalable_agent repository was to set up the deepmind-lab repository (`https://github.com/deepmind/lab`). The first round of errors were related to building up the deepmind-lab repo and revolved around **pip issues**, and **linking .so files using the Bazel compiler**

- We used our college's jupyter-repo2docker open source project to attempt to resolve the dependency issues inherent in their docker file as suggested. This route did not resolve all the issues. Even the solutions provided on the closed issues of the github repositories were not useful

Figure 1: Closed issue regarding the dockerfile not working

- We then switched to making our own virtual environment with manual installations of the dependencies to build the repository from scratch
- Figuring this out took us around a day

3.1.2 Manual Installation

Details of our efforts and the errors faced while manually installing the dependencies and building the scalable_agent repository are as follow:

- The deepmind-lab repository with the tasks/environments to run by the agent is compiled with a Bazel compiler. The instructions to install Bazel were wrongly provided in the readme of repository. Closed issues of the github repo were able to better guide us in making the Bazel compiler work

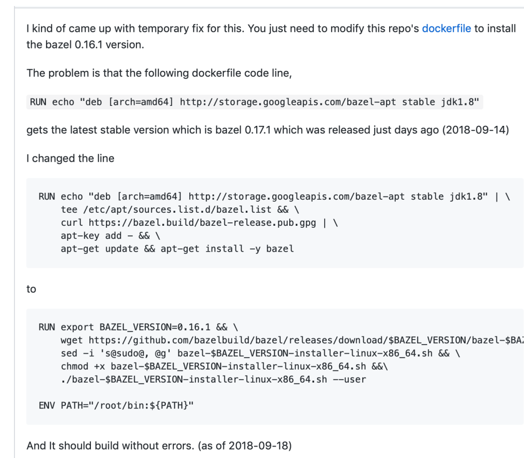

Figure 2: Closed issue regarding solution to making the Bazel compiler work

  Instead of doing deb and apt-get we had to wget a stable script file release and update the environment path. This solved the Bazel error we were getting

- For the pip issues in the docker file, we had to search and look at the Python.BUILD script. Although the script suggested that the Scalable_agent repository supported both python2 and python3, the python2 functions had function incompatibility issues, leading us to reinstall all the previous dependencies for python3. We had to install a specific version of Tensorflow==1.15.0 and dm-sonnet==1.23 that made the repository work

- For resolving the linking issues; the dynamic batching module had to be re-run with the right version of the GCC library (gcc=4.8)

After spending around a week purely on installation, we were able to get the scalable_agent repository up and running, and have a requirements.txt file detailing the correct versions of all the necessary dependencies for the scalable_agent codebase. We are happy to release it for people who want to work on this project in the future.

3.2 Implementation

3.2.1 Replay Buffer

Rolnick et al. [2018] described the implementation of a replay buffer to store necessary information for the off-policy V-trace algorithm. Each actor needed a separate replay buffer of $capacity = number_of_frames/(2 * number_of_actors)$. Every actor forwarded a tuple to the learner, containing the current unroll (trajectory) and the replay history. The learner selected samples from the replay buffer and the current unroll using (50%, 50%) probability. Once the buffers were filled up to the capacity, the new unrolls were added using reservoir sampling, as mentioned by Isele and Cosgun [2018], so that the buffer contains uniform random samples up to the present point and not just recent unrolls. In reservoir sampling, each experience is assigned a random value, which serves as the key in priority queue where the experiences are preserved with the highest key values.

For the implementation of the replay buffer, we had to figure out where in the code should the buffer be populated (whether it was inside the $build_actor$ function or the $train$ function). The $build_actor$ function generates the output containing the agent state, agents output and environments output, while in the $train$ function the $build_actor$, $build_learner$ and $run.session$ loop is called.

We made multiples attempts at designing the data structure for the replay buffer since the authors provided no information on this. When asked about the implementation of the replay buffer as a GitHub issue, the authors provided a vague answer that they used a custom C++ implementation with a promise of release around a year back. The only information provided was to use tf_py function to make wrappers for the replay buffer to increase the speed. A screenshot of the issue is provided in Figure 3

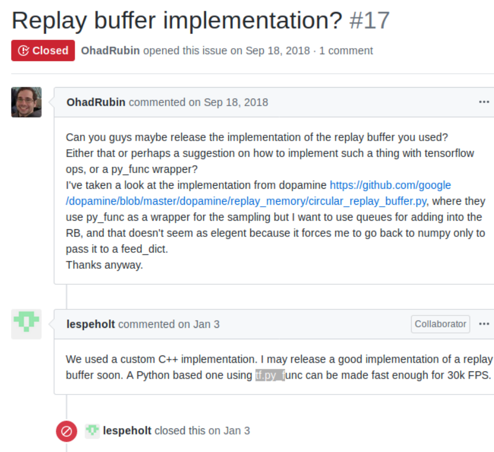

Figure 3: Github issue - Replay buffer implementation

The order in which the data structures were implemented for the replay buffer and the errors we ran into implementing each of them are as follows:

3.2.1.1 Replay buffer as list per actor

The replay buffers were implemented as a list of lists data structure where each replay buffer was implemented as a list per actor, and all the replay buffers per actor were put into one list. This did not

work since the samples of unrolls had to be reorganized into a specific internal structure initialized by the developers of the authors, and the samples from the replay buffers could not be packed into this specific internal structure to be fed to the learner.

The exact error we faced is shown in Figure 4.

```
Traceback (most recent call last):
  File "experiment_queues.py", line 791, in <module>
    tf.app.run()
  File "/home/ifrah/.local/lib/python3.6/site-packages/tensorflow/python/platform/app.py", line 125, in run
    _sys.exit(main(argv))
  File "experiment_queues.py", line 785, in main
    train(action_set, level_names)
  File "experiment_queues.py", line 626, in train
    sample_replay_buffer = nest.pack_sequence_as(structure, sample_replay_buffer)
  File "/home/ifrah/.local/lib/python3.6/site-packages/tensorflow/python/util/nest.py", line 316, in pack_sequence_as
    % (len(flat_structure), len(flat_sequence), structure, flat_sequence))
ValueError: Could not pack sequence. Structure had 12 elements, but flat_sequence had 24 elements:   Structure: ActorOutput(level_name=<tf.Tensor 'StopGradient:0' shape=() dtype=string>, agent_state=LSTMSta
teTuple(c=<tf.Tensor 'StopGradient_1:0' shape=(256,) dtype=float32>, h=<tf.Tensor 'StopGradient_2:0' shape=(256,) dtype=float32>), env_outputs=StepOutput(reward=<tf.Tensor 'StopGradient_3:0' shape=(101,)
dtype=float32>, info=StepOutputInfo(episode_return=<tf.Tensor 'StopGradient_4:0' shape=(101,) dtype=float32>, episode_step=<tf.Tensor 'StopGradient_5:0' shape=(101,) dtype=int32>), done=<tf.Tensor 'StopGr
adient_6:0' shape=(101,) dtype=bool>, observation=[<tf.Tensor 'StopGradient_7:0' shape=(101, 72, 96, 3) dtype=uint8>, <tf.Tensor 'StopGradient_8:0' shape=(101,) dtype=string>]), agent_outputs=AgentOutput(
```

Figure 4: Error - Replay Buffer as a list data structure per actor

3.2.1.2 Replay buffer as queue per actor

Implementing each replay buffer as a queue per actor led to issues with TensorFlow 's serializability. Initializing multiple queue runners to enqueue the unrolls in the replay buffers caused errors, as shown in Figure 5.

```
INFO:tensorflow:Error reported to Coordinator: <class 'tensorflow.python.framework.errors_impl.UnknownError'>, EOFError: Ran out of input
Traceback (most recent call last):

  File "/home/ifrah/.local/lib/python3.6/site-packages/tensorflow/python/ops/script_ops.py", line 158, in __call__
    ret = func(*args)

  File "/home/ifrah/lab/scalable_agent/py_process.py", line 86, in py_call
    result = self._out.recv()

  File "/usr/lib/python3.6/multiprocessing/connection.py", line 251, in recv
    return _ForkingPickler.loads(buf.getbuffer())

EOFError: Ran out of input


         [[Node: scan/while/flow_environment_step/step = PyFunc[Tin=[DT_STRING, DT_INT32], Tout=[DT_FLOAT, DT_BOOL, DT_UINT8, DT_STRING], _class=["loc:@scan/while/TensorArrayWrite_8/TensorArrayWriteV3"],
token="pyfunc_3", _device="/job:local/replica:0/task:0/device:CPU:0"](scan/while/flow_environment_step/step/input_0, scan/while/GatherV2)]]2019-11-30 18:57:16.577932: W tensorflow/core/framework/op_kernel
.cc:1306] Unknown: UnpicklingError: invalid load key, '\x00'.
Traceback (most recent call last):

  File "/home/ifrah/.local/lib/python3.6/site-packages/tensorflow/python/ops/script_ops.py", line 158, in __call__
    ret = func(*args)

  File "/home/ifrah/lab/scalable_agent/py_process.py", line 86, in py_call
    result = self._out.recv()

  File "/usr/lib/python3.6/multiprocessing/connection.py", line 251, in recv
    return _ForkingPickler.loads(buf.getbuffer())

_pickle.UnpicklingError: invalid load key, '\x00'.
```

Figure 5: Error - Replay Buffer as a queue data structure per actor

Asynchroniscity among multiple threads lead to multiple blocking statements, and the above errors of running out of input to fill the receiving buffer. As stated by the documentation of TensorFlow, using a coordinator object with the queue runners could potentially resolve the issues, but the scalable_agent codebase was written to optimize for speed concerns specifically, and these optimizations were so interwoven in the code that it made it hard for us to incorporate the coordinator object.

3.2.1.3 Replay buffer as list per actor - second attempt

We went back to the implementation of replay buffer as a list, with the goal of overcoming the structure packing issues described in Section 3.2.1.1. We were able to overcome the structure issue by separating the replay buffer into separate replay buffers for each actor correctly, but the code processed this structure before sending it to the learner. This structure processing can be found in lines $551 - 571$ of the original code in the scalable_agent repository, and we started facing a new error during this processing, as shown in Figure 6.

```
INFO:tensorflow:Error reported to Coordinator: <class 'tensorflow.python.framework.errors_impl.UnknownError'>, EOFError: Ran out of input
Traceback (most recent call last):

  File "/home/ifrah/.local/lib/python3.6/site-packages/tensorflow/python/ops/script_ops.py", line 158, in __call__
    ret = func(*args)

  File "/home/ifrah/lab/scalable_agent/py_process.py", line 86, in py_call
    result = self._out.recv()

  File "/usr/lib/python3.6/multiprocessing/connection.py", line 251, in recv
    return _ForkingPickler.loads(buf.getbuffer())

EOFError: Ran out of input


         [[Node: scan/while/flow_environment_step/step = PyFunc[Tin=[DT_STRING, DT_INT32], Tout=[DT_FLOAT, DT_BOOL, DT_UINT8, DT_STRING], _class=["loc:@scan/while/TensorArrayWrite_8/TensorArrayWriteV3"],
token="pyfunc_3", _device="/job:local/replica:0/task:0/device:CPU:0"](scan/while/flow_environment_step/step/input_0, scan/while/GatherV2)]]2019-11-30 18:57:16.577932: W tensorflow/core/framework/op_kernel
.cc:1306] Unknown: UnpicklingError: invalid load key, '\x00'.
Traceback (most recent call last):

  File "/home/ifrah/.local/lib/python3.6/site-packages/tensorflow/python/ops/script_ops.py", line 158, in __call__
    ret = func(*args)

  File "/home/ifrah/lab/scalable_agent/py_process.py", line 86, in py_call
    result = self._out.recv()

  File "/usr/lib/python3.6/multiprocessing/connection.py", line 251, in recv
    return _ForkingPickler.loads(buf.getbuffer())

_pickle.UnpicklingError: invalid load key, '\x00'.
```

Figure 6: Error - Replay Buffer as a list data structure per actor - transpose error

We believe that this error came up because our replay buffer was a list of lists, while the way the unroll was processed by the developers of the code was a list of a custom tensor object class. We tried converting our replay buffer list into a tensor object but ran into the error of incompatible types.

3.2.1.4 Replay buffer as list per task

We went back to the implementation of replay buffer as a list, but this time implemented it as a list per task to overcome the type error, as mentioned in 3.2.1.3. This implementation led us to the error of not being able to pack the sequence into the specific internal structure, as shown in Figure 7.

```
Traceback (most recent call last):
  File "experiment_queues.py", line 791, in <module>
    tf.app.run()
  File "/home/ifrah/.local/lib/python3.6/site-packages/tensorflow/python/platform/app.py", line 125, in run
    _sys.exit(main(argv))
  File "experiment_queues.py", line 785, in main
    train(action_set, level_names)
  File "experiment_queues.py", line 626, in train
    sample_replay_buffer = nest.pack_sequence_as(structure, sample_replay_buffer)
  File "/home/ifrah/.local/lib/python3.6/site-packages/tensorflow/python/util/nest.py", line 316, in pack_sequence_as
    % (len(flat_structure), len(flat_sequence), structure, flat_sequence))
ValueError: Could not pack sequence. Structure had 12 elements, but flat_sequence had 4 elements.  Structure: ActorOutput(level_name=<tf.Tensor 'StopGradient:0' shape=() dtype=string>, agent_state=LSTMSta
teTuple(c=<tf.Tensor 'StopGradient_1:0' shape=(256,) dtype=float32>, h=<tf.Tensor 'StopGradient_2:0' shape=(256,) dtype=float32>), env_outputs=StepOutput(reward=<tf.Tensor 'StopGradient_3:0' shape=(101,)
dtype=float32>, info=StepOutputInfo(episode_return=<tf.Tensor 'StopGradient_4:0' shape=(101,) dtype=float32>, episode_step=<tf.Tensor 'StopGradient_5:0' shape=(101,) dtype=int32>), done=<tf.Tensor 'StopGr
adient_6:0' shape=(101,) dtype=bool>, observation=[<tf.Tensor 'StopGradient_7:0' shape=(101, 72, 96, 3) dtype=uint8>, <tf.Tensor 'StopGradient_8:0' shape=(101,) dtype=string>]), agent_outputs=AgentOutput(
action=<tf.Tensor 'StopGradient_9:0' shape=(101,) dtype=int32>, policy_logits=<tf.Tensor 'StopGradient_10:0' shape=(101, 9) dtype=float32>, baseline=<tf.Tensor 'StopGradient_11:0' shape=(101,) dtype=float
32>)), flat_sequence: [[<tf.Tensor 'StopGradient:0' shape=() dtype=string>, <tf.Tensor 'StopGradient_1:0' shape=(256,) dtype=float32>, <tf.Tensor 'StopGradient_2:0' shape=(256,) dtype=float32>, <tf.Tensor
 'StopGradient_3:0' shape=(101,) dtype=float32>, <tf.Tensor 'StopGradient_4:0' shape=(101,) dtype=float32>, <tf.Tensor 'StopGradient_5:0' shape=(101,) dtype=int32>, <tf.Tensor 'StopGradient_6:0' shape=(10
1,) dtype=bool>, <tf.Tensor 'StopGradient_7:0' shape=(101, 72, 96, 3) dtype=uint8>, <tf.Tensor 'StopGradient_8:0' shape=(101,) dtype=string>, <tf.Tensor 'StopGradient_9:0' shape=(101,) dtype=int32>, <tf.T
ensor 'StopGradient_10:0' shape=(101, 9) dtype=float32>, <tf.Tensor 'StopGradient_11:0' shape=(101,) dtype=float32>], [<tf.Tensor 'StopGradient_36:0' shape=() dtype=string>, <tf.Tensor 'StopGradient_37:0'
 shape=(256,) dtype=float32>, <tf.Tensor 'StopGradient_38:0' shape=(256,) dtype=float32>, <tf.Tensor 'StopGradient_39:0' shape=(101,) dtype=float32>, <tf.Tensor 'StopGradient_40:0' shape=(101,) dtype=floa
t32>, <tf.Tensor 'StopGradient_41:0' shape=(101,) dtype=int32>, <tf.Tensor 'StopGradient_42:0' shape=(101,) dtype=bool>, <tf.Tensor 'StopGradient_43:0' shape=(101, 72, 96, 3) dtype=uint8>, <tf.Tensor 'Sto
pGradient_44:0' shape=(101,) dtype=string>, <tf.Tensor 'StopGradient_45:0' shape=(101,) dtype=int32>, <tf.Tensor 'StopGradient_46:0' shape=(101, 9) dtype=float32>, <tf.Tensor 'StopGradient_47:0' shape=(10
1,) dtype=float32>], [<tf.Tensor 'StopGradient_72:0' shape=() dtype=string>, <tf.Tensor 'StopGradient_73:0' shape=(256,) dtype=float32>, <tf.Tensor 'StopGradient_74:0' shape=(256,) dtype=float32>, <tf.Ten
sor 'StopGradient_75:0' shape=(101,) dtype=float32>, <tf.Tensor 'StopGradient_76:0' shape=(101,) dtype=float32>, <tf.Tensor 'StopGradient_77:0' shape=(101,) dtype=int32>, <tf.Tensor 'StopGradient_78:0' sh
ape=(101,) dtype=bool>, <tf.Tensor 'StopGradient_79:0' shape=(101, 72, 96, 3) dtype=uint8>, <tf.Tensor 'StopGradient_80:0' shape=(101,) dtype=string>, <tf.Tensor 'StopGradient_81:0' shape=(101,) dtype=int
32>, <tf.Tensor 'StopGradient_82:0' shape=(101, 9) dtype=float32>, <tf.Tensor 'StopGradient_83:0' shape=(101,) dtype=float32>], [<tf.Tensor 'StopGradient_108:0' shape=() dtype=string>, <tf.Tensor 'StopGra
dient_109:0' shape=(256,) dtype=float32>, <tf.Tensor 'StopGradient_110:0' shape=(256,) dtype=float32>, <tf.Tensor 'StopGradient_111:0' shape=(101,) dtype=float32>, <tf.Tensor 'StopGradient_112:0' shape=(1
01,) dtype=float32>, <tf.Tensor 'StopGradient_113:0' shape=(101,) dtype=int32>, <tf.Tensor 'StopGradient_114:0' shape=(101,) dtype=bool>, <tf.Tensor 'StopGradient_115:0' shape=(101, 72, 96, 3) dtype=uint8
>, <tf.Tensor 'StopGradient_116:0' shape=(101,) dtype=string>, <tf.Tensor 'StopGradient_117:0' shape=(101,) dtype=int32>, <tf.Tensor 'StopGradient_118:0' shape=(101, 9) dtype=float32>, <tf.Tensor 'StopGra
dient_119:0' shape=(101,) dtype=float32>]].
```

Figure 7: Error - Replay Buffer as a list data structure per task

3.2.1.5 Replay buffer as queue per task

The methods of the tensor class Queue were able to handle the type errors and sequence packing errors by themselves. The reason we were more inclined to a list structure than a queue structure as used by developers of the Scalable agent code was that we didn't want to dequeue from the replay buffer; instead, we wanted to keep appending to it to maintain the history of trajectories. Nevertheless, we again attempted to use the queue data structure, but this time per task. This implementation again did not work and gave us the same error as Section 3.2.1.2, shown in Figure 5.

We had to try all these attempts because the authors were not clear about the details for the implementation of the replay buffer. We imagine that for researchers wanting to use this codebase as a baseline for their research, spending so much in implementing the original code is not an effort well-spent. Replication of the replay buffer would have been more feasible if we at least had an idea of what underlying data structures were used and a better understanding of the optimizations of the code. The latter could have been achieved by providing better documentation of the scalable_agent repository.

3.2.2 Sequential Training

The IMPALA (scalable_agent) codebase has settings for single-task and multi-task training, corresponding to the separate and simultaneous training shown in Figure 1 in Rolnick et al. [2018]. However, the setting used in Rolnick et al. [2018] is sequential training (the right-most image in Figure 1), to demonstrate the network's performance for continual learning. Rolnick et al. [2018] did not specify how they modified the scalable_agent codebase to have the network trained on sequences of tasks, therefore, we operated solely on our guesses of how sequential training was implemented.

Sequential training is essentially single-task training where the task switches from time to time. Therefore, we initially sought to modify the single-task training setting to switch the training task after a number of frames. In the single-task setting, all actors are initialized with the same task. We modified the code to reinitialize the actors with a new task after a number of frames, and in effect switching the training task. However, trying to reinitialize the actors gave us an error stating that the Tensorflow graph is finalized and cannot be modified.

As re-initializing the actors was not feasible, we decided to initialize all actors at the beginning with all the tasks we wanted to train the network on, and have separate queues to hold the output from the actors for a specific task. However, the way the learner was accessing the actors' output was unclear and not well-documented in the codebase, and we were unsuccessful in modifying the code to switch the source of training data (and thus the training task) for the network.

3.2.3 Additional Loss Functions

In order to implement policy-cloning and value-cloning as detailed in the CLEAR method, we wrote two different python functions for each of the loss functions. The python code for these two loss functions can be found in Figure 8. Although we were unable to apply these loss functions in the training of the network, as the functions rely on the replay buffer being correctly implemented, we were able to write the loss functions based on the details in the paper, and include them here for posterity.

```
##########################################################################
# New Functions:
# Applied only for replay experiences
# The weights were added as described in Section A.4.
# The motivation for behavioral cloning is to prevent network output on
# replayed tasks from drifting while learning new tasks.
##########################################################################

def compute_policy_cloning_loss(logits, observed_exp_logits):
    policy = tf.nn.softmax(logits)
    observed_policy = tf.nn.softmax(observed_exp_logits)
    log_frac = tf.math.log(tf.math.divide(policy, observed_policy), axis=1)
    loss = tf.reduce_sum(- observed_policy * log_frac, axis=-1)
    return - 0.01 * loss

def compute_value_cloning_loss(replay_advantages):
    return 0.005 * tf.reduce_sum(tf.square(tf.l2_normalization(replay_advantages)))
```

Figure 8: Additional loss functions

4 Results

With our implemented corrections to the IMPALA codebase, we were able to replicate the first figure in the CLEAR paper where actors were trained on separate tasks. Our replicated figure and the figure from the original paper are included in Figure 9. This figure is important because it helps differentiate between the effects of catastrophic forgetting and interference. Destructive interference is when two tasks that are being learned are in conflict with one another in terms of learned behavior. Catastrophic forgetting is when newly learned experiences override previous experiences. While these two are not mutually exclusive, interference can either be destructive (hurtful) or constructive (helpful), while catastrophic forgetting is often present whenever the agent switches between learning tasks. Figure 1

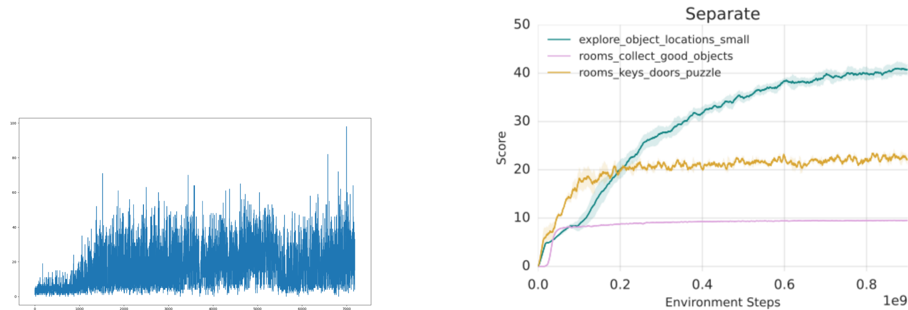

Figure 9: (Left) Our replicated graph. (Right) Figure 1 from the CLEAR paper, separate tasks. Note that we only replicated the explore_object_location_small curve

201 attempts to address the differences in these two failure modes by comparing the baseline method,
202 IMPALA, on separate, simultaneous, and sequential task learning paradigms.

203 We note that the original figure ran for one billion environment steps, but we were unable to run our
204 model to completion under the same conditions due to our limited compute capability. We also note
205 that in our figure, we only report the training curve for the explore_object_location_small curve.

206 Overall, our graph has more variance than the figure in the original paper, but this is accounted for by
207 the fact that we only report a single training instance of our model, while the graph from the paper
208 appears to have averaged results over many runs (However, there is no mention in the original paper
209 of how many times they ran their models, nor do they specify whether the graph includes standard
210 deviation or confidence intervals). It is promising to note that our model starts off poorly (as expected
211 when starting from scratch), and begins to learn such that the score begins to rise over time. It appears
212 that our replicated graph has an average score that is around 20, which is similar to the actual figure
213 at around the same amount of environment steps. Therefore, we find our replicated figure to positively
214 suggest that our implementation results similarly reflects those found in the original paper.

215 # 5 Conclusion

216 This article describes our process of replicating the paper "Experience Replay for Continual Learning"
217 and what results we gathered. We found issues with the original codebase and report detailed
218 descriptions of how we approached these problems. In addition, many important implementation
219 details for the model and figures were not included in the paper, and we therefore include our methods
220 of implementing the CLEAR method based on our best guesses. In the end, while we were not able
221 to successfully implement the replay buffer and sequential task training, which was necessary for
222 replicating Figure 2 in the original paper, we were able to get the original baseline working on the
223 separate tasks and write the loss functions for policy cloning and value cloning. Therefore, while we
224 were not able to confirm that results of this paper, we contributed many corrections that are are useful
225 for any other researchers who wish to implement the CLEAR method.

226 **Acknowledgments**

227 We would like to give special thanks to Neev Parikh, our assigned Teacher Assistant mentor, for
228 discussions regarding our replication. We would also like to give thanks to Deniz Bayazit, who was
229 originally on our team and contributed to our coding replication.

230 # References

231 Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam
232 Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with
233 importance weighted actor-learner architectures. In *Proceedings of the International Conference*
234 *on Machine Learning (ICML)*, 2018.

235 David Isele and Akansel Cosgun. Selective experience replay for lifelong learning. *CoRR*, 236 abs/1802.10269, 2018. URL `http://arxiv.org/abs/1802.10269`.

237 David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy P Lillicrap, and Greg Wayne. Experience 238 replay for continual learning. *arXiv preprint arXiv:1811.11682*, 2018.