Learning Motor Policies with Time Continuous Neural Networks

Titouan Renard [1]

[1] *MT-RO, 272257*

Abstract *Time-Continuous Neural Networks provide an effective framework for the modeling of dynamical systems [5] and are natural candidates for continuous-control tasks. They are closely related to the dynamics of non-spiking neurons, which gives further justification to investigate their use in control [11]. The following document describes the work and results obtained while investigating the use of such neural networks during a semester project jointly supervised by EPFL's BIOROB and LCN labs.*

Introduction

We first consider time-continuous neural networks, and the main ideas of reinforcement learning, then we discuss implementation details, and finally we go go through the results obtained during the project.

1 Time-Continuous Neural Networks

1.1 Formulating a Neural Network model for Continuous-Time Processes

In the following discussion, we only worry about time-independent (sometimes also refered to as autonomous) systems, as those are more relevant for control, but most of those approaches generalize well to time-dependent systems as well. Time-Continuous Neural Networks model dynamical systems model the evolution of the hidden states (which we denote as $\boldsymbol{x}_t$ at a given time t) of a neural network by equations of the form:

$$\frac{\partial \boldsymbol{x}_t}{\partial t} = D(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta). \tag{1}$$

Where D denotes some kind of model function that estimates the time-derivative of $\boldsymbol{x}_t$. D is a function of $\boldsymbol{x}_t$, which denotes the hidden state of the neuron, $\boldsymbol{I}_t$ which denotes the inputs of the neuron and a learnable parameter vector θ. Updates of the (hidden) state $\boldsymbol{x}_t$ are the computed using some ODE solver which integrates $\frac{\partial \boldsymbol{x}_t}{\partial t}$ over some time-step Δt to compute $\boldsymbol{x}_{t+\Delta t} = \int_t^{t+\Delta t} \frac{\partial \boldsymbol{x}_t}{\partial t} + \boldsymbol{x}_t$.

$$D(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta) = f(\boldsymbol{I}_t, \theta). \tag{2}$$

It is worth noting that in a supervised learning context, this formulation has the advantage of being able to represent irregularly sampled time-sequences. For the control applications we consider here, the sample-rate is imposed by the hardware of our robot and is likely regular. In the original formulation of Chen et al, neural ODEs are not recurrent but a natural extension to recurrent neural networks can be formulated as:

$$D(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta) = f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta). \tag{3}$$

Where the flow D is not only a function of the input $\boldsymbol{I}_t$ but also of the inner-state $\boldsymbol{x}_t$. Such a model can be though of as a recurrent neural network where the recurrent connection are implemented by an integrator.

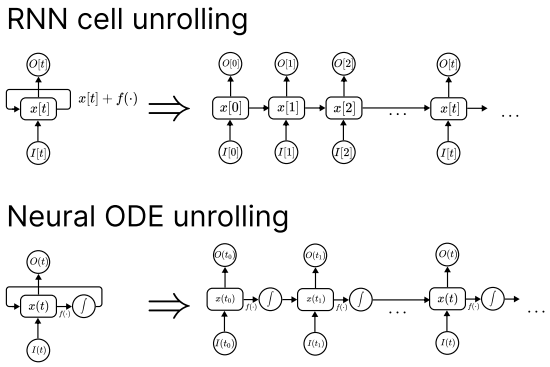

Fig. 1: An illustration of the difference between Neural ODE unrolling v.s. RNN unrolling.

The most straight forward approach to that problem is directly using a neural network to model the flow D, this is the approach chosen by [5] (which is often referred to as "Neural-ODEs"). An alternative provided by an earlier contribution is the so called "Continuous-Time Recurrent Neural Network" model (CT-RNNs) first proposed by Funahashi and Nakamura in [1], which pick D as:

$$D(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta) = -\frac{\boldsymbol{x}_t}{\tau} + f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta). \tag{4}$$

where τ is a fixed time constant (according to our formalism it is an element of the vector A) and f a non-linear activation function. The fixed time-constant is is introduced to induce a decay in the behavior of the neurons which is meant to enable recurrent connections which avoiding unstable neuron dynamics. In their original paper, Funahashi and Nakamura propose to use $f = \sum_{j=1}^{m} w_{i,j} \cdot \sigma\left(x_{i,t}\right) + I_i(t)$, where the weights wi, j are elements of the vector θ.

More recently, one suggested implementation that seems to give better performance is proposed by Hasani and Lechner though so-called "Liquid Time-Constant networks (LTCs)" [12]. In that case the activation f affects both the time-constant and the non-linearity

(hence make the time-constant "liquid"), this approach corresponds to the following time-derivative model:

$$D(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta) = -\left[\frac{1}{\tau} + Af(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta, A)\right]\boldsymbol{x}_t + f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta). \quad (5)$$

Where A is a so called *bias* parameter which con-trolls the non-linearity in the synaptic response. In practice, Hasani and Lechner propose to use the following activation function: $f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta, A) = \tanh(w^{inner}\boldsymbol{x} + w^{inputs}\boldsymbol{I} + \mu)$, which is loosely connected to the non-linearity observed in synaptic dynamics between biological neurons [11].

Table 1 Time-Continuous Neural Network Classes

	Hidden state equation	Recurrent?
CT-RNN	$\frac{\partial \boldsymbol{x}_t}{\partial t} = -\frac{\boldsymbol{x}_t}{\tau} + f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta)$	Yes
LTC	$\frac{\partial \boldsymbol{x}_t}{\partial t} = -\left[\frac{1}{\tau} + f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta)\right]\boldsymbol{x}_t + f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta)$	Yes
Neural-ODE	$\frac{\partial \boldsymbol{x}_t}{\partial t} = f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta)$	No
RNN-ODE	$\frac{\partial \boldsymbol{x}_t}{\partial t} = f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta)$	Yes

1.2 Training

Compared to multi-layer perceptrons and RNNs computing gradients on continuous time neural networks is way less obvious because of the integrator step. In the following section we discuss two approaches to the computation of such gradients together with their pros and cons. Two main approaches are possible *backpropagation through time* (BPTT, which is recommanded by Hasani et al. [12]) and the *adjoint sensitivity method* (which is recommended by Chen et al [5]).

Backpropagation through time works by directly computed the gradient of a loss function through the ODE solver (it requires our ODE solver to build a computation graph and then we use *autograd* to compute a gradient).

TODO: Finish write up on BPTT

The *adjoint sensitivity method* required saving a so-called adjoint state $\boldsymbol{a}(t) = \frac{\partial L}{\partial \boldsymbol{z}(t)}$ throughout the unrolling of the neural network.

TODO: Finish write up on the adjoint method

1.3 Continuous Time Neural Network Cells

In order to deal with the motor task that we consider in this project we choose to investigate small fully connected neuron cells that take a n-dimensional input, contain k inner-neurons and return d outputs (we call the outputs "*motor neurons*") We provide a visual representation of such a cell in figure 2. Most of our experiments are performed with LTC cells, so we will explicitly derive the forward pass equations for an LTC, but we can build equivalent cells for RNN-ODE and CT-RNN cells in a very similar fashion.

For a given inner-neuron i, the flow of it's state is computed according to the following equation:

$$\frac{dx_i}{dt} = f_i(\mathbf{x}, \mathbf{I}, \theta)$$
$$= -\frac{x_i}{\tau_i} + \tanh\left(\sum_{j=1}^{n} w_{ij}^{inputs} I_j + \sum_{j=1}^{n} w_{ij}^{inner} x_j + \mu_i\right)(A_i - x_i).$$

The states of each neurons are then passed through a linear layer to compute the output values as follows. Each output value's is computed as:

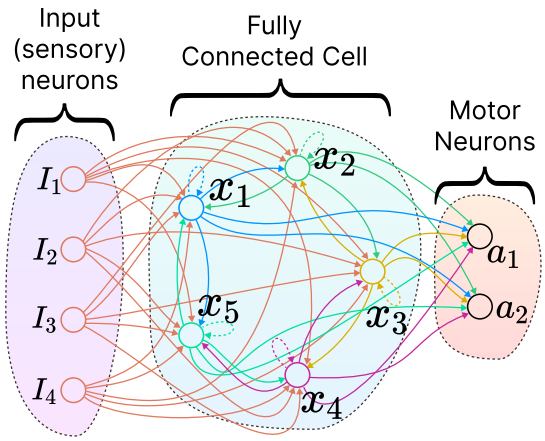

Fig. 2: Fully connected Time-Continuous Neural Network Cell. With 4 inputs ($o_{1,\dots,4}$), 5 inner-neurons (with inner states $x_{1,\dots,5}$) and 2 motor neurons (outputs $I_{1,2}$).

$$o_i = \sum_{j=1}^{k} w_{ij}^{output} x_j$$

Note that this implies that each neuron's flow is influenced by it's input, every single other neuron on the same layer and also itself (it has a recurrent connection). This gives $k \cdot (k + n + d)$ connections in a single LTC cell, which is much denser than a typical multi-layer perceptron. To take a concrete example, in the implementation section when we discuss a 16 neuron LTC cell for solving the cart-pole problem, it has 352 connections for 16 neurons, where a 16 neuron perceptron with a single hidden-layer perceptron would only have 96.

2 Policy Gradient Methods and Reinforcement Learning

In order to optimize the continuous time models we choose to use reinforcement learning methods, the following section covers the Markov Decision Process formalism and gives an introduction to the main ideas behind Proximal Policy Optimization (PPO) [4], the algorithm that we will use to train our model.

2.1 Markov Decision Processes

Formulating our control problem as a reinforcement learning problem requires us to write out a control task as a partially observable Markov decision process (POMDP). We define a POMDP through:

1. a (partially observable) state space $\mathcal{S}$ with states $s_t \in \mathbb{R}^n$,
2. an observation space $\mathcal{O}$ with states $o_t \in \mathbb{R}^m$, where observations o_t give some information about the true states s_t,
3. an action space $\mathcal{A}$ with action $a_t \in \mathcal{A} \subseteq \mathbb{R}^d$, which gives the possible actions that the agent can take in a given state,
4. an unknown transition probability function $P(s_{t+1}|s_t, a_t)$, which gives the probability that the system transitions to state s_{t+1} if it is in state s_t and action a_t is taken,
5. a reward function $R : (s_{t+1}, s_t, a_t) \to \mathbb{R}$.

Given a POMDP, a discount factor $\gamma \in [0, 1)$ and a policy function $\pi : \mathcal{S} \to \mathcal{A}$ we can compute the *expected discounted reward* using the Bellman equation:

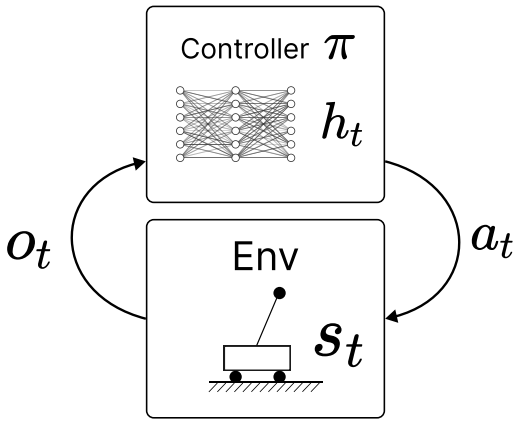

Fig. 3: A visual representation of the different components required to define an RL problem. An environment, which keeps state and defines the transition probability function, and a policy function (which may have a hidden state h) which takes observations o_t of s_t as inputs and returns actions a_t as outputs.

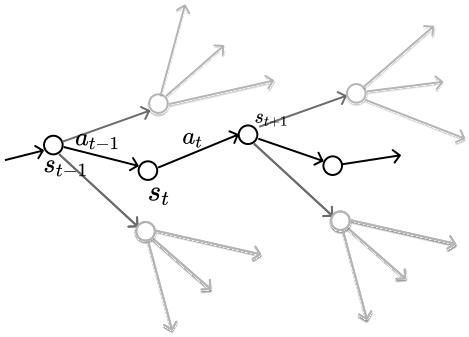

Fig. 4: One can think of a Markov Decision Process as a tree of possible states $s_t, s_{t+1}, ...$ connected by branches associated with the transition probability function $P(s_{t+1}|s_t, a_t)$.

$$J\pi_\theta = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_{t+1}, s_t, \pi_\theta(o_t))\right].$$

The problem of reinforcement learning is formulated as the optimization of some parametrizable policy function π (where the parameters are denoted θ) over the POMDP process that ensures the the J value is maximized in expectation across all states, i.e.:

$$\pi^* = \text{argmax}_{\pi_\theta} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_{t+1}, s_t, \pi_\theta(o_t))\right].$$

2.2 Policy Gradient Methods

We will consider a *policy gradient* reinforcement learning method, such an approach is the most natural for a continuous control task, these method work by directly updating a policy function rather than by computing an estimator of the discounted reward for all possible actions (which is what is done in Q-learning methods). Policy gradient RL methods use update rules of the form:

$$\theta_{k+1} = \theta_k + \alpha \nabla_{\theta_k} J(\pi_{\theta_k})|_{\theta_k}.$$

Here the tricky part of the method resides in deriving an expression for the gradient $\nabla_{\theta_k} J(\pi_{\theta_k})|_{\theta_k}$ of the expected reward J with respect to the parameters θ of the policy π. These methods are often presented in MDP instead of POMDP form but they generalize well to POMDPs. The derivation of policy gradient is performed as follows:

$$\begin{aligned}
\nabla_{\theta_k} J(\pi_{\theta_k})|_{\theta_k} &= \nabla_{\theta_k} \int_\tau P(\tau|\theta) R(\tau) && \text{Expand expectation} \\
&= \int_\tau \nabla_{\theta_k} P(\tau|\theta) R(\tau) && \text{Use linearity} \\
&= \int_\tau P(\tau|\theta) \nabla_{\theta_k} \log P(\tau|\theta) R(\tau) && \text{Log trick} \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\nabla_{\theta_k} \log P(\tau|\theta) R(\tau)\right] && \text{Take the expectations} \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\nabla_{\theta_k} \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) R(\tau)\right] && \text{Expand over trajectories}
\end{aligned}$$

Where τ denotes the set of all trajectories $s_0, a_0, s_1, a_1, ...$ in the state space under policy π_θ. In practice one can compute an estimate $\hat{g}$ of $\nabla_{\theta_k} J(\pi_{\theta_k})$ by sampling the MDP a sufficiently large number of time. Such an estimator is written out as follows:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t,$$

where G_t is the expected discounted reward over the remaining steps in the episode.

2.3 Advantage Actor Critic

Policy gradient methods derived as we described above tend to lead to unstable learning. This is largely attributable to the fact that the gradient norms are subject to a high variance [2]. The gradient in the variance of $\nabla_{\theta_k} J(\pi_{\theta_k})$ is proportional to the absolute value of the expected discounted reward over the trajectory $R(\tau)$, we can thus reduce the variance of our gradient estimator by subtracting a baseline to it's reward (this doesn't affect the gradients in expectation and thus the algorithm still converges to the same policy), the most common baseline used in that context is the *on-policy value function* $V_{\pi_\theta}(s)$. This means a policy algorithm such as A2C we will have require two separate networks, an actor network (to implement a policy function) and a a critic network (to implement a value estimator) which both need to train simultaneously. To derive the formulation of advantage actor critic we first observe policy gradient implicitely makes use of Q values.

$$\hat{g} = \mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t\right]$$

$$= \mathop{\mathbb{E}}_{s_0,a_0,\dots}\left[\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t\right]\right. \qquad \text{Observe that:}$$

$$\left. + \mathop{\mathbb{E}}_{s_{t+1},r_{t+1},\dots,s_T,r_T}[G_t]\right] \qquad \mathbb{E}[G_t] = Q(s_t, a_t)$$

$$= \mathop{\mathbb{E}}_{s_0,a_0,\dots}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) Q(s_t, a_t)\right]$$

Then we subtract the baseline V as follows:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - V(s))$$

Using the Bellman Optimality equation $Q(s_t, a_t) = \mathbb{E}[r_{t+1}] + \gamma V(s_{t+1})$ we have that can write out the advantage function as:

$$A(s_{t+1}, s_t, a_t) = Q(s_t, a_t) - V(s_t, a_t)$$
$$\sim r_{t+1} + \gamma V(s_{t+1}) - V(s_t, a_t)$$

This approach leads us the the *Advantage Actor Critic Method* (A2C) which computes it's gradients from an advantage function A instead of a direct reward R:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_{t+1}, s_t, a_t)$$

$$= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$$

In that framework we train two networks side by side, one of them is updated by the approximative gradient $\hat{g}$, and the other is updated by some loss function so that it converges to correct V values.

2.4 Proximal Policy Optimization

Describe PPO

2.5 An architecture for training CTNNs using PPO

Describe what I did

3 Implementation of CTNN in a Reinforcement Learning framework

The reinforcement learning-based training of a time-continuous neuron cell requires the setup of a learning environment. Such an environment must be able to:

1. simulate the POMDP (since we investigate a continuous control task, this is the physics of our controlled system),
2. implement the policy model (in our case our time-continuous neuron cells) which we discussed in section 1.3,
3. train it using our RL algorithm (PPO) which we discussed discussed in section 2.4.

Because of the complexity of such an environment we make use of a reinforcement learning framework that structures the data collection, the learning and the evaluation of RL policies. The reinforcement learning research community has come up with several such frameworks, notable examples are *stable baselines* [6], *stable baselines 3* [13], *Ray RLLib* [7] and *ACME* [10]. The requirements of our project lead us to seriously consider two main frameworks: *Ray RLLib* [7] and the less known *DERL* [8], which we will now compare in further details.

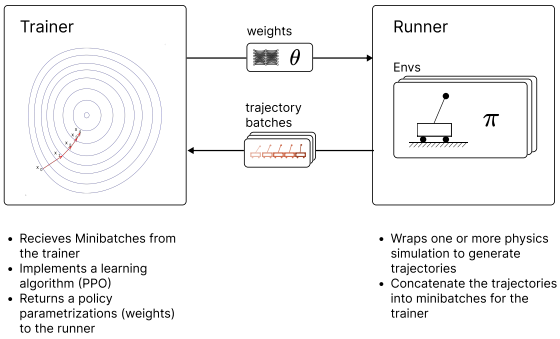

Fig. 5: An illustration of the main components of a reinforcement learning framework. The two main components are left: an implementation of the RL algorithm, and right a wrapper for physics simulations (or environment as they are often called in the RL literature). One of the key functions of such a system is handling the trajectories generated by the environment and converting them into mini-batches, depending on the framework this can be done as part of the runner or trainer components.

3.1 Ray RLLib

Our first investigation into the problem we stated were performed within the *Ray RLLib* framework. *Ray* is an open source machine learning framework intended form large-scale distributed computing, its development started at UC Berkley's RISE Lab. *Ray RLLib* [7] is a reinforcement learning framework built on top of *Ray* with the intent of allowing for fast training of RL policies on distributed hardware (for instance on a cluster). Ray features implementation of most state of the art reinforcement learning algorithms (for instance *PPO*, it's asynchronous variant *APPO*, *IMPALA*, various Q-learning derivatives as well as imitation learning algorithms) and bindings with most environments used for continuous control (most notably *PyBullet* and *Mujoco*). *RLLib* allows for implementing policies with either *tensorflow* [3] or *pytorch* [9].

The main argument for the use of *RLLib* is the performance it provides when deployed on a cluster, furthermore much of my direct supervisor Dr. Bellegarda's work on reinforcement learning for the control of complex quadruped robots was performed within this framework, which may open a door to experiments on such systems with less work involved.

Nonetheless I ended up abandoning *RLLib*, this is because of several drawbacks, first most *RLLib*'s implementation either do not support recurrent policies and are often shipped with bugs for recurrent policies [14]. This lack of support of recurrent policies makes the implementation of recurrent continuous time cells such as the ones we discuss in section 1.3, and the fact that *RLLib*'s documentation doesn't clearly specifies which model classes are unsupported further complicates the work. The second big argument for abandoning RLLib is the high complexity of implementing relatively simple changes to algorithms within the framework, this is partly a product of the fact that RLLib applications work as independent processes networked together. This makes the debugging of RLLib applications especially difficult. Furthermore the size of *Ray*'s codebase further participates in slowing the process.

3.2 DERL

DERL [8] is a lightweight reinforcement learning framework providing implementations of *PPO*, *SAC*, *A2C* and *DQN*.

4 Results and Discussion

References

[1] Ken-ichi Funahashi and Yuichi Nakamura. "Approximation of dynamical systems by continuous time recurrent neural networks". In: *Neural Networks* 6 (1993), pp. 801–806.

[2] Vijay Konda and John Tsitsiklis. "Actor-Critic Algorithms". In: *SIAM Journal on Control and Optimization*. MIT Press, 2000, pp. 1008–1014.

[3] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[4] John Schulman et al. "Proximal Policy Optimization Algorithms." In: *CoRR* abs/1707.06347 (2017). URL: http://dblp.uni-trier.de/db/journals/corr/corr1707.html#SchulmanWDRK17.

[5] Tian Qi Chen et al. "Neural Ordinary Differential Equations". In: *NeurIPS*. 2018.

[6] Ashley Hill et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2018.

[7] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 3053–3062. URL: https://proceedings.mlr.press/v80/liang18b.html.

[8] Mikhail Konobeev. *Neural Ordinary Differential Equations for Continuous Control*. https://github.com/MichaelKonobeev/neuralode-rl. 2019.

[9] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[10] Matt Hoffman et al. "Acme: A Research Framework for Distributed Reinforcement Learning". In: *arXiv preprint arXiv:2006.00979* (2020). URL: https://arxiv.org/abs/2006.00979.

[11] Mathias Lechner et al. "Neural circuit policies enabling auditable autonomy". In: *Nature Machine Intelligence* 2 (2020), pp. 642–652.

[12] Ramin M. Hasani et al. "Liquid Time-constant Networks". In: *AAAI*. 2021.

[13] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[14] *Github Issue [Bug] [rllib] RNN sequencing is incorrect*. https://github.com/ray-project/ray/issues/19976. Accessed: 2022-06-09.