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 [8] 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 [15]. 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 relevant to this kind of task, then we discuss implementation details, and finally we go through the results obtained during the project.

1 Time-Continuous Neural Networks

In the following section, we presented the main ideas behind Time-Continuous Neural Networks (TCNN).

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) models, as those are more relevant for control, but most of those approaches generalize well to time-dependent models as well. In other words, we will not consider systems which behave differently depending on the time at which we evaluate them. 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.

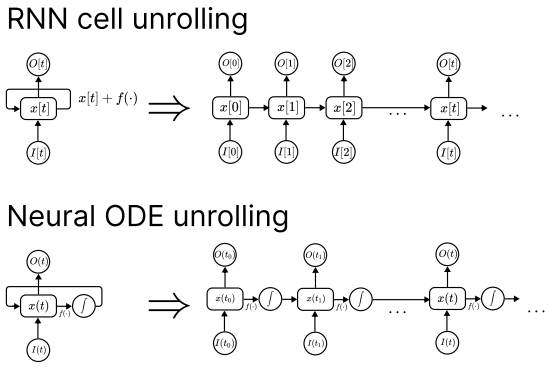

Fig. 1: An illustration of the difference between unrolling a Time-Continuous neural network and unrolling a RNN.

The most straight forward approach to implementing a continuous-time neural network is to directly use a neural network to model the flow D. This is the approach chosen by [8] (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 [2], 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 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 while avoiding unstable neuron dynamics. In their original paper, Funahashi and Nakamura propose to use $f = \tanh\left(\sum_{j=1}^{m} w_{i,j} \cdot \sigma\left(x_{i,t}\right) + I_i(t)\right)$, where the weights wi, j are elements of the vector θ.

More recently, one suggested implementation that seems to give better performance is suggested by Hasani and Lechner under the

name "Liquid Time-Constant networks (LTCs)" [16]. This model introduces further non-linearity by having f affect the time-constant (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 controls 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 [15].

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 less obvious because of the integrator step introduced in the computation of the inner state $\boldsymbol{x}$ (we provide a graphical representation of the difference between a RNN and a TCNN in figure 1). In the following section we discuss two approaches to the computation of such gradients together with their pros and cons: *backpropagation through time* (BPTT, which is recommended by Hasani et al. [16]) and the *adjoint sensitivity method* (which is recommended by Chen et al [8]).

Backpropagation through time (BPTT) [16] 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). BPTT requires saving the inner state of neurons for every step of the ODE solver, this comes at a high memory cost.

The *adjoint sensitivity method* [8], required saving a so-called adjoint state $\boldsymbol{a}(t) = \frac{\partial L}{\partial x(t)}$ throughout the unrolling of the neural network.

Given some Loss function:

$$L(x(t)) = L\left(x(t_0) + \int_{t_0}^{t} D(x(t), \theta)dt\right),$$

we define our adjoint sensitivity state as $a(t) = \frac{dL}{dx(t)}$ where x is the inner state of our differential equation. The dynamics of $a(t)$ are given by the equation: $\frac{dL(z(t))}{dx(t)}$. The idea here is to use the stored $a(t)$ to compute a gradient:

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} a(t)^T \frac{dD(x(t), \boldsymbol{I}_t, \theta)}{d\theta} dt$$

Note that we compute the loss backwards through time (from t_1 to t_0) which is why the negative sign appears in front of the integral. Which we can use to perform gradient descent on our model. This computation is performed using the same numerical ODE solver as for the forward pass.

Although Hasani et al. suggest that BPTT reduces numerical error during training [16], it's implementation is quite complex and we decide to stick with the adjoint method for our experiments, as we will discuss in the results section, this does not seem to stop us from learning policies.

1.3 Continuous Time Neural Network Cells

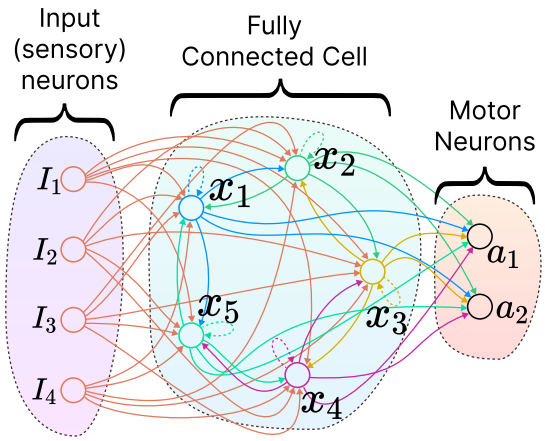

Fig. 2: Fully connected Time-Continuous Neural Network Cell. With 4 inputs ($o_{1,\dots,4}$), 5 inner-neurons (with inner states $x_{1,\dots,5}$) and 2 motor neurons (outputs $a_{1,2}$).

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 (refer to table 1 for a reference for the difference between the different models).

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:

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

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, a 16 neuron LTC cell, 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) [7], the algorithm that we will use to train our model.

2.1 Markov Decision Processes

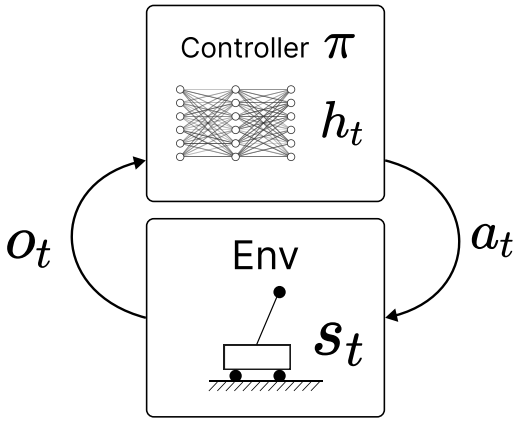

Fig. 3: A visual representation of the different components required to define an RL problem. An environment, which keeps state and defines the transition probability function, and a policy function (which may have a hidden state h) which takes observations o_t of s_t as inputs and returns actions a_t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

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:

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

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.:

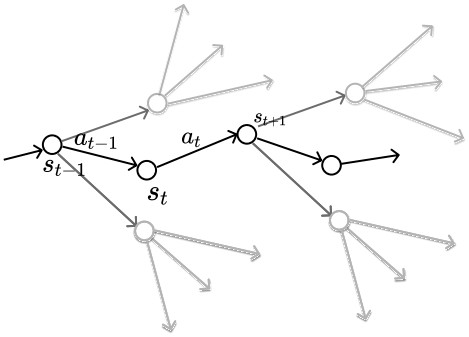

Fig. 4: One can think of a Markov Decision Process as a tree of possible states $s_t, s_{t+1}, ...$ connected by branches associated with the transition probability function $P(s_{t+1}|s_t, a_t)$.

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

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 [3]. 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 an actor-critic algorithm such as A2C we will 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.

$$\begin{aligned}
\hat{g} &= \mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t\right] \\
&= \mathop{\mathbb{E}}_{s_0, a_0, \ldots}\Bigg[\Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t\Big] \qquad \text{Observe that:} \\
&\qquad + \mathop{\mathbb{E}}_{s_{t+1}, r_{t+1}, \ldots, s_T, r_T}[G_t]\Bigg] \quad \mathbb{E}[G_t] = Q(s_t, a_t) \\
&= \mathop{\mathbb{E}}_{s_0, a_0, \ldots}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) Q(s_t, a_t)\right]
\end{aligned}$$

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:

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

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:

$$\begin{aligned}
\hat{g} &= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_{t+1}, s_t, a_t) \\
&= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))
\end{aligned}$$

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.

Algorithm 1: PPO-Clip

Initialize neural networks π_θ and V_ϕ
Set counter $t \leftarrow 0$, observe s_0 from the env
repeat
  forall *workers* k **do**
    Take action $a_t^{(k)} \leftarrow \pi_\theta(s_t^{(k)})$, **observe reward** $r_t^{(k)}$ and new state $s_{t+1}^{(k)}$
    Compute $R_t^{(k)} = r_t^{(k)} + \gamma V_\phi(s_{t+1})$ and the advantage $A_t^{(k)} = R_t^{(k)} - V_\phi(s_t)$
  Update the policy : optimize the surrogate loss $\hat{L}^{\text{CLIP}}(\theta') = \sum_k \gamma^t \text{clip}(r_{\theta'}(s_t, a_t), 1-\epsilon, 1+\epsilon) A_\theta(s_t, a_t)$ with gradient descent for M epochs
  Update the advantage estimator : ϕ by the gradient of $\sum_K (R_t^{(k)} - V_\phi(s_t^(k)))^2$
until *convergence*;

2.4 Proximal Policy Optimization

Starting from A2C one can get further improvements in the stability of the convergence by constraining gradient descent to a small region in parameter space for each mini-batch. This allows for faster learning and for less sensitivity to the learning rate. Algorithms making use of this idea (Trust Region Policy Optimization and Proxy Policy Optimization [6]) are giving state of the art results for continuous control tasks. A really nice walkthrough of the main ideas behind them can be found in Johanni Brea's lecture notes [19] for the course Artificial Neural Networks at EPFL.

A rough intuition of why these methods work can be gathered from writing out the expected policy improvements $J(\theta') - J(\Theta)$ of A2C.

$$\begin{aligned}
J(\theta') - J(\theta) &= J(\theta') - \mathop{\mathbb{E}}_{s_0 \sim p(s_0)}[V_\theta(s_0)] && \text{By def. of } J(\theta) \\
&= J(\theta') - \mathop{\mathbb{E}}_{s_t, a_t \sim \theta'}[V_\theta(s_0)] && \text{Take the expectation over all state action pairs (for policy } \pi(\theta')) \\
&= J(\theta') - \mathop{\mathbb{E}}_{s_t, a_t \sim \theta'}\Bigg[\underbrace{\sum_{t=0}^{\infty} \gamma^t V_\theta(s_t) - \sum_{t=1}^{\infty} \gamma^t V_\theta(s_t)}_{\sum_{t=0}^{\infty} \gamma^t [V_\theta(s_t) - \gamma V_\theta(s_t)]}\Bigg] && \text{Write out the expectation as a collapsing sum} \\
&= \mathop{\mathbb{E}}_{s_t, a_t \sim \theta'}\Bigg[\sum_{t=0}^{\infty} \gamma^t \underbrace{[r^a_{s_t \to s_{t+1}} - \gamma V_\theta(s_{t+1}) - V_\theta(s_t)]}_{A_\theta(s_t, a_t)}\Bigg] && \text{Plugging in the definiton of } J(\theta') \\
&= \mathop{\mathbb{E}}_{s_t, a_t \sim \theta'}\Bigg[\sum_{t=0}^{\infty} \gamma^t A_\theta(s_t, a_t)\Bigg] && \text{By definiton of the advantage function} \\
&= \mathop{\mathbb{E}}_{s_t, a_t \sim \theta}\Bigg[\sum_{t=0}^{\infty} \gamma^t \frac{\pi_{\theta'}(s_t, a_t)}{\pi_\theta(s_t, a_t)} A_\theta(s_t, a_t)\Bigg] && \text{Taking expectation over } \pi(\theta) \text{ instead. of } \pi(\theta')
\end{aligned}$$

We observe that the expected improvement is maximized in expectation when the term $\gamma^t \frac{\pi_{\theta'}(s_t, a_t)}{\pi_\theta(s_t, a_t)} A_\theta(s_t, a_t)$ is maximized. Assuming that the updated policy is close to the previous parametrization (i.e. $\frac{\pi_{\theta'}(s_t, a_t)}{\pi_\theta(s_t, a_t)} \approx 1$). We can directly optimize (maximize) an estimator of the improvement that we call $\hat{L}(\theta')$:

$$\hat{L}(\theta') = \sum_{t=0}^{\infty} \gamma^t \overbrace{\frac{\pi_{\theta'}(s_t, a_t)}{\pi_\theta(s_t, a_t)}}^{= r_{\theta'}(s_t, a_t) \approx 1} A_\theta(s_t, a_t).$$

There are several approaches to keep the updated policy "close" to the previous installment ($r_{\theta'}(s_t, a_t) \approx 1$), one of the is them compute an estimator of the KL divergence between both policies (which is what is done in TRPO) another is simply clip the loss at some maximum according to the following clipping function:

$$\text{clip}(x, \mu, \nu) = \begin{cases} \nu & \text{if } x < \nu \\ x & \text{if } \nu < x < \mu \\ \mu & \text{if } \mu < x \end{cases}.$$

We called the clipped loss function the "*surrogate loss*". This gives rise to the *PPO-Clip* algorithm for which we give a pseudocode implementation in algorithm 1.

Algorithm 2: Batched PPO-Clip

Initialize neural networks π_θ and V_ϕ
Set counter $t \leftarrow 0$, observe s_0 from the env
repeat
  Initialize a trajectory batch $\tau \leftarrow (\tau^{(1)}, \tau^{(2)}, ..., \tau^{(k)})$
  for $t \in [0, \textit{episode lenght}]$ **do**
    forall *workers* k **do**
      Take action $a_t^{(k)} \leftarrow \pi_\theta(s_t^{(k)})$, **observe reward** $r_t^{(k)}$ and new state $s_{t+1}^{(k)}$
      Compute $R_t^{(k)} = r_t^{(k)} + \gamma V_\phi(s_{t+1})$ and the advantage $A_t^{(k)} = R_t^{(k)} - V_\phi(s_t)$
      Append $s_t^{(k)}, a_t^{(k)}, s_{t+1}^{(k)}, x_t^{(k)}, R_t^{(k)}, A_t^{(k)}$ to the trajectory $\tau^{(t)}$ associated with the worker
  while *trajectory batch* τ *is not empty* **do**
    sample mini-batch from τ (and remove sampled elements)
    Update the policy : optimize the surrogate loss over sampled mini-batch $\hat{L}^{\text{CLIP}}(\theta') = \sum_k \gamma^t \text{clip}(r_{\theta'}(s_t, a_t), 1-\epsilon, 1+\epsilon) A_\theta(s_t, a_t)$ with gradient descent for M epochs
    Update the advantage estimator : ϕ by the gradient of $\sum_K (R_t^{(k)} - V_\phi(s_t^{(}k)))^2$ over sampled mini-batch
until *convergence*;

This reference implementation assumes that the policy is updated as the policy goes through environment steps. In practice this causes trouble when implementing the algorithm as it forces the batch size k to be directly fixed by the number of parallel workers. An alternative approach consists in sampling the environment for several steps (we call this sequence of steps an "*episode*") and then to reconstitute batches after the fact for training. This is the approach that we will use for our experiments. Algorithm 2 gives pseudocode for such an implementation.

2.5 An architecture for training TCNNs using PPO

Dealing with the continuous time nature of the TCNN cells doesn't cause much trouble when implementing a reinforcement learning algorithm as it mostly just changes the back-propagation algorithm but not the loss or the variables that need to be stored in the process. On the other hand the fact that our TCNN cells are recurrent (have an inner state) changes the process as we somehow need to implement the *unrolling* of the inner states.

The way we do it is by saving the inner states $x_i^{(t)}$ of each neuron at each time step in every trace and then to consider them to be observations when training the policy, this allows for a minimal amount of changes to the PPO architecture at the cost of a relatively high memory use.

3 Implementation of CTNN in a Reinforcement Learning framework

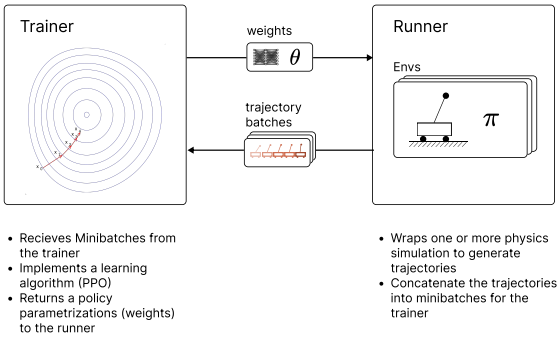

Fig. 5: An illustration of the main components of a reinforcement learning framework. The two main components are left: an implementation of the RL algorithm, and right a wrapper for physics simulations (or environment as they are often called in the RL literature). One of the key functions of such a 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.

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 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* [9], *stable baselines 3* [17], *Ray RLLib* [10] and *ACME* [14]. The requirements of our project lead us to seriously consider two main frameworks: *Ray RLLib* [10] and the less known *DERL* [11], which we will now compare in further details.

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 for large-scale distributed computing, its development started at UC Berkley's RISE Lab. *Ray RLLib* [10] 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* [5] or *pytorch* [13].

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 or are shipped with bugs for recurrent policies [20]. 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, especially difficult. 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. The size of *Ray*'s codebase further participates in slowing the process.

3.2 DERL

DERL [11] is a lightweight reinforcement learning framework written by Mikhail Konobeev, an EPFL Phd student, providing implementations of *PPO*, *SAC*, *A2C* and *DQN* as well as bindings with continuous control environments such as *Mujoco* and *Pybullet*. Unlike *RLLib*, *DERL* isn't intended for distributed computation and it's computation are for the most part run as part of a single python process, this makes debugging much more straightforward. Furthermore the light codebase (as a quick comparison the *Ray* git repository contains 643288 lines of python whereas the *DERL* repository only contains 4314) makes modifications much easier to carry-out. On the other hand the non-distributed nature of it's implementation makes it much slower than *RLLib* and greatly reduces the potential performance benefits that could be hoped for when running it on a cluster. Because of the exploratory nature of the project we deem this trade-off interesting.

The version of *DERL* I used (`06fcd44`) only supports *tensorflow* (and not *pytorch*) so it imposes the choice of the Deep Learning Library. One big argument for the use of *DERL* was an example implementation of non-recurrent neural ODEs for control [11], this provides a starting point for building of continuous time neural network cells. *DERL* doesn't provide support for recurrent policies, so a large part of my work resided in the implementation of training with recurrent policies, which mostly consists in handling the saving of the inner state of recurrent model as part of the trajectories.

3.3 Mujoco

The two most used physics simulator for reinforcement learning are *Pybullet* [21] and *Mujoco* [4]. The quadruped simulations used in *BIOROB* are most often built within the *Pybullet* environment which until recently was the best open-source choice for the task. This changed as *Google Deep Mind* bought and open-sourced the - until then proprietary - *Mujoco* physics engine in may 2022. *Mujoco* provides a slightly faster implementation of a physics simulation and as *DERL* provides bindings to the *Mujoco* gym environments developed at *OpenAI* it made sense to make use of the *Mujoco* environment in the context of the project.

4 Results and Discussion

4.1 Experimental Setup

In order to investigate the efficiency of TCNN cells we setup a simple (and rather classical in the RL literature [1]) motor control experiment: the cart-pole problem. We use the *Mujoco* gym

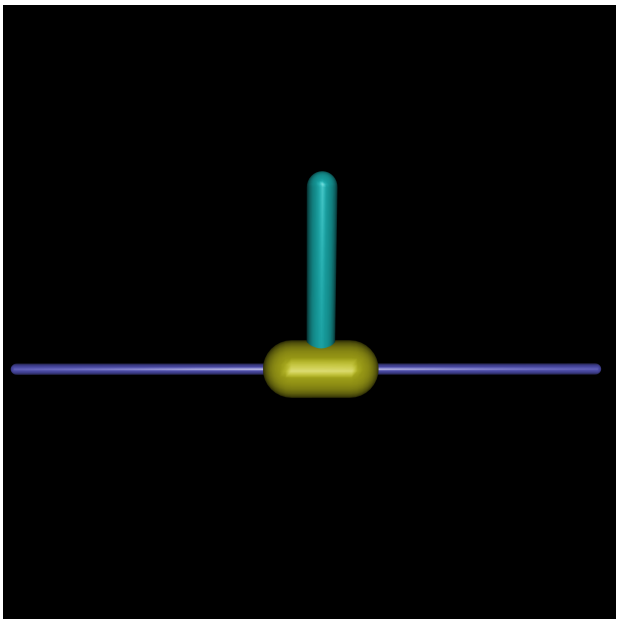

Fig. 6: The cart-pole system, as rendered inside of *Mujoco*'s built-in viewer.

implementation of cart-pole together with our *DERL* implementation of *PPO* training for TCNN policies to perform our experiments. We use the adjoint method for the computation of gradients and focus our experiments on *LTC cells*. The code used for the experiments is publicly available at https://github.com/RenardDesNeiges/CTNN_Policies_DERL.

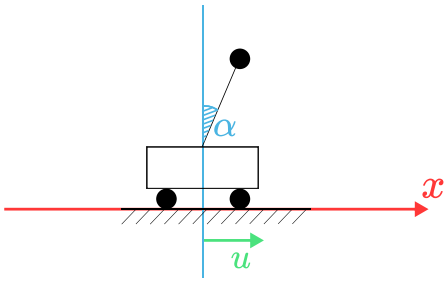

Fig. 7: The cart-pole system, with both coordinates x and α, as well as the control variable u (a linear force on the cart) clearly labeled.

The cart-pole problem is characterized by a 4-dimensional continuous observation space given by $o(t) = [\alpha, x, \dot{\alpha}, \dot{x}]$ and a single dimensional continuous action space which we denote as $a(t) = u$ (see figure 7). For all of our cart-pole experiments we use the following reward:

$$r_t = \begin{cases} -2 & \text{if } |\alpha| > \alpha_{\max} \text{ (failure)} \\ 1 - qu^2 & \text{otherwise} \end{cases}$$

Where $q = 0.2$ is a penalty on control (where u is a force measured in newtons) and where the angle $\alpha_{\max} = 0.209$ is used to define failure (with α the pole-angle measured in radiant). The cart-pole implementation we use has a limited range of motion in x (when the cart reaches the end of it's range of motion it cannot go further which means a well tuned policy should learn to avoid the

edges). Together with the observation space, the action space and the stochastic transition probability function defined by the physics simulation this defines a POMDP.

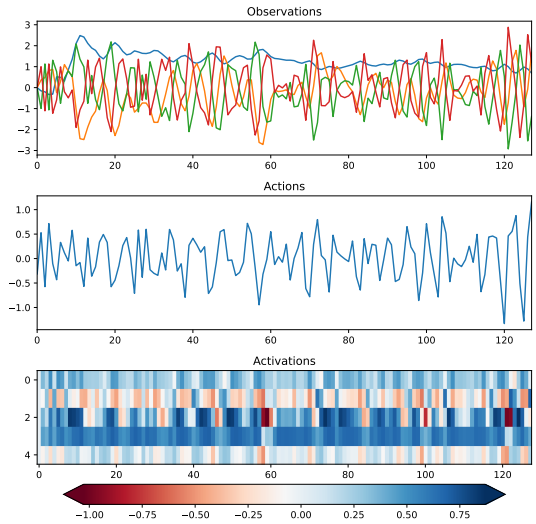

Fig. 8: Visualization of (from the top down) observation, action and neuron activations together for a 5 neuron LTC cell trained with PPO on a cart-pole *Mujoco* environment. The activation plot is structured as follows, each row correspond to a single neuron is colored along the t horizontal axis according to the activation x_i of that neuron at time t.

We find that the LTC cell trained with PPO can find efficient solutions to the cart-pole problem, but that it does so with a particularly low sample efficiency (compared to a reference *Multi-Layer Perceptron* (MLP) model trained with a perceptron). On the other hand the amount of neurons required to achieve a good solution is extremely low compared to the amount of MLP neurons that would be required for solving the problem with a MLP model. This ability of the model to find low-neuron count solutions isn't of interest for computational reasons (recall the the number of parameters per neuron is much higher than in the case of an MLP) but may prove interesting as that low neuron count may make the system easier to interpret.

4.2 The 5 neuron solution, an attempt at interpreting a learned policy

The lowest number of neuron that actually could solve the cart-pole problem is 5. The solution obtained with 5 neurons actually leads to a relatively unstable control (the cart-pole oscillates a lot under that policy, as can be observed in figure 8) but it is interesting because the policy can be investigated relatively easily, which we try to do in this section. We plot out connections of the trained cell in figure 9.

The cart-pole trained policies take a 4-dimensional vector as an input, so plotting a visual representation of their behavior is non trivial. One approach that we can to generate two two-dimensional plots where we compute responses of the policy and value nets to observations in either $\alpha, \dot{\alpha}$-space or $x, \dot{x}$-space. In both case we fix both the other two observation variables (respectively $x, \dot{x}$ or $\alpha, \dot{\alpha}$) to 0. For now we also choose to fix the inner states of all the neurons to 0 (we disregard self-connections). This enables us to plot out a *portrait* of how our LTC cell responds to various observations from it's environment (we call these plots *portraits* as we are trying to interpret these plots in a similar fashion to how phase-portraits are used in the dynamical system theory literature). We plot out both the output of the policy network and of the value network. We use the policy network output to plot a phase-portrait as a vector field where for each point $(\alpha, \dot{\alpha})$ we plot a vector (α, u) and for each point $(x, \dot{x})$

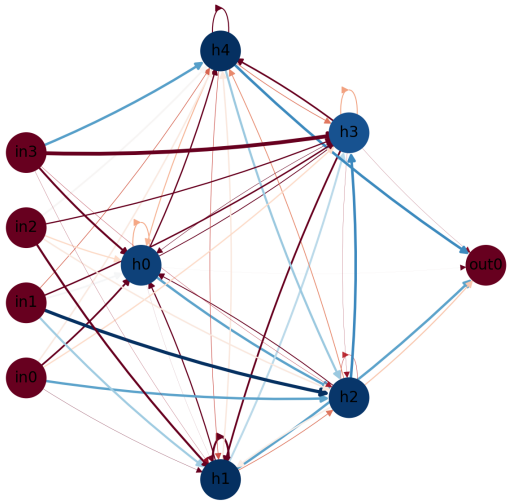

Fig. 9: Representation of the 5 neuron LTC cell. Connections are colored and scaled with respect to their intensities, the inner-neurons are colored with respect to their time-constants. The connections are colored by weight, blue connections are positive-weighted and red ones are negative-weighted.

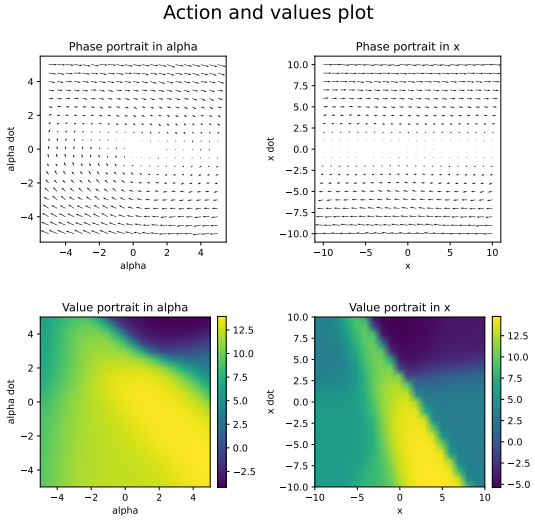

Fig. 10: Characterization of a 5-neuron LTC-cell policy on cart-pole. Phase portraits (top) and value portraits (down) in the pole angle phase space $\alpha, \dot{\alpha}$ (left) and cart position phase space (right).

we plot a vector (x, u) and the output of the value network (we construct this plot in hope of identifying sinks and sources in the flow of the closed loop dynamical system). In the case of the value network we simply plot out the estimated value for each point in the space we investigate, this gives us an idea of "*how good does the policy thinks being in a given state is*", we call these plots "*value portraits*".

The *portraits* we computed for the 5-neuron LTC cell (figure 10), allow us to identify a sink at the point $(0, 0)$ in the pole-angle phase space (see the top left plot in figure 10), which tends to indicate that our policy is tends to stabilize the cart-pole system around that point (which is what is to be expected). On the other hand the phase portrait in the cart-position phase-space doesn't show any clearly defined sink, this confirms what we observe when running simulations: the policy doesn't center the cart-pole's movements around the origin. This is suboptimal and shows that the 5-neuron policy isn't fully converged.

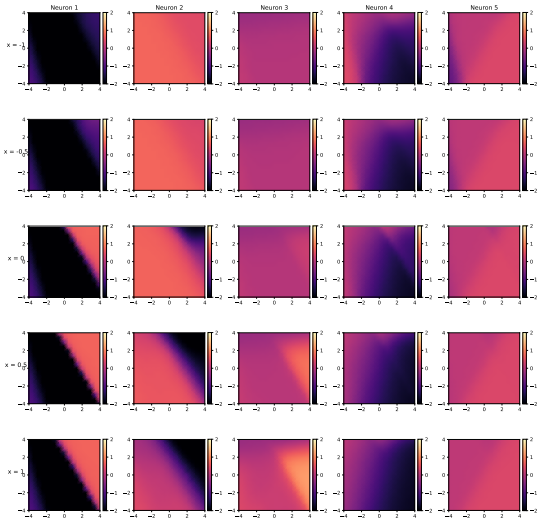

Fig. 11: Pole angle phase-space response of each of the 5 neurons of the cells with various inner-states activations x_i. From left-to-right: responses of neurons 1-to-5. From top-to-bottom: responses of the neurons with inner states -1-to-1.

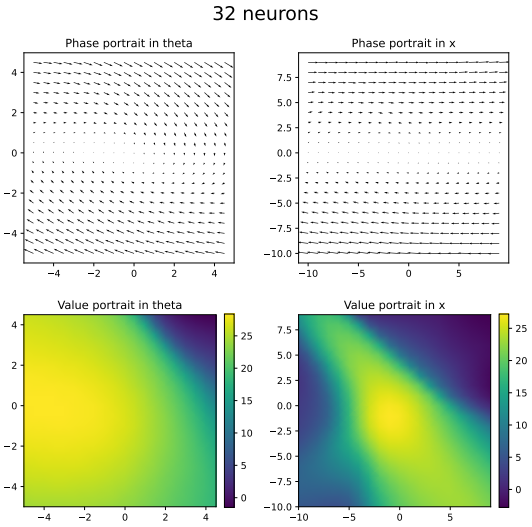

Fig. 12: Characterization of a 32-neuron LTC-cell policy on cart-pole. Phase portraits (top) and value portraits (down) in the pole angle phase space $\alpha, \dot{\alpha}$ (left) and cart position phase space (right).

Furthermore, the value portraits from figure 10 show asymmetric value estimations which further indicates that the policy shows hasn't completely converged (the control problem is symmetric so one should expect a converged value net to predict a symmetric performance).

This approach isn't sufficient to characterize the response of the 5-neuron cell as it completely disregards the influence of the inner-neuron states, it also fails to interpret the behavior of different neurons. In order to build a deeper understanding of the cell we investigate the response of individual neurons to different inputs and to various activations. We decide to focus on the policy network and on the pole-angle phase space, we construct a plot for each of the 5-neuron and for various inner-states activations. The resulting plot is presented in figure 11.

The resulting plot shows a clear influence of the inner states of the neurons influence (although the influence of the angle seems to dominate for every neuron, this is also a tendency which we observe when comparing the influence of angle compared to position). This indicates that the neuron make some use of the recurrent connections.

4.3 Neuron Count Experiments

We run experiments with various neuron counts ($5, 6, 7, 8, 16$, and 32 neurons). The main questions we aim to answer with such a protocol are "*Is there a minimum neuron count to solve such a problem?*" (and how different is it from the minimum neuron count with a perceptron) and "*how does the neuron count affect the quality of solution?*".

The answer to the first question, the minimum neuron count required to solve cart-pole seems to be somewhere around 5 (the cell that we investigated in section 4.2 is the lowest count for which we could observe convergence). As for the second question, it is worth mentioning that as we discussed although the 5-neuron policy "solves" the cart-pole problem, it displays a less stable solution than higher neuron-count policies this is obviously seen when comparing the value portrait of the 5-neuron policy from figure 10 to the one of a 32-neuron policy (figure 12). The value portraits are similar the value ranges are very different : the value of the 32 neuron cell is close to being strictly positive whereas the 5 neuron value is negative for large areas of the portraits. A negative value estimation implies that the cell "expects" it will fail if it ends up in a given area of the phase-space.

We produce a more rigorous characterization of that effect by comparing the failure rate of various policies as follows: for each policy we compute 20 independent simulation of 128 steps each with different random seeds. For each simulation we count the failures (as defined in section 4.1), for each failure we reset the environment and continue (so there can be more than 1 failure in 1 simulation). We then compare the total number of failures across policies, results are presented in figure 13.

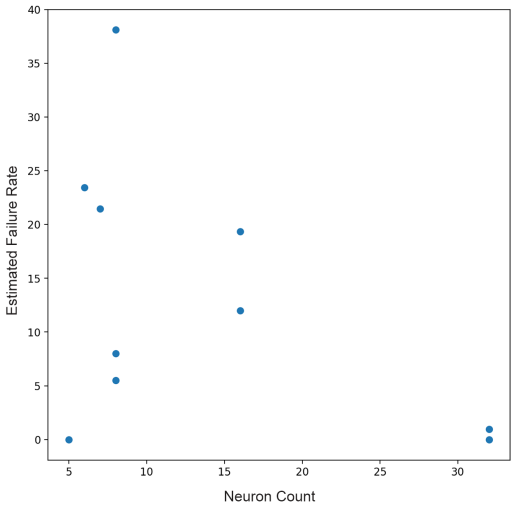

Fig. 13: Comparing the performance of policies with varying neuron counts. Each point represents one trained policy, the x-axis gives the number of neurons of that policy and the y-axis the number of failures during the experiments.

The failure rate experiment that we ran shows a general trend towards a lower failure rate for the policies with the bigger amount of neurons (see figure 13). Nonetheless it is interesting to observe that we have a clear outlier : the 5-neuron policy that we presented in section 4.2 shows 0 failures across the experiments. This seems to indicate that what limits the quality of the trained policies for low

neuron counts is perhaps not the expressivity of the TCNN cells but rather the learning process. This observation, together with the convergence speed results displayed in figure 14 may hint at an influence of overparametrization on the convergence speed [18].

4.4 Sample efficiency

One key aspect of the training of TCNN policies for control task with PPO is the limited sample efficiency compared to more straight forward models. The fastest convergence we manage to reach is around $30'000$ steps (with a relatively high neuron count, see figure 14) to convergence, this is much slower than what we could hope for with an equivalent MLP model (around 3 times slower for large TCNN cells, and even less for smaller ones). This is a problem as it makes more complex tasks, such as solving cart-pole with a masked environment, or higher degree-of-freedom control tasks such as legged locomotion, manipulation tasks etc, out of reach, because of computational limitations.

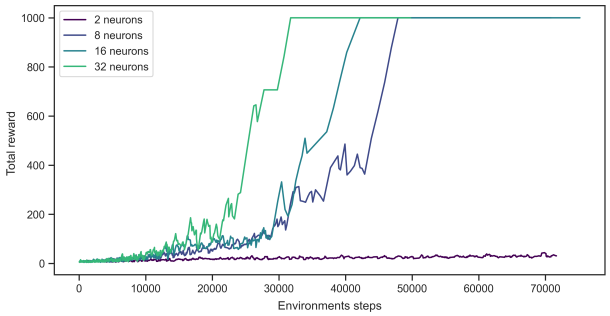

Fig. 14: Learning curves (reward over environment steps) of 4 example LTC policies trained on cart-pole with PPO. This plot showcases a clear inverse correlation between the per-step speed of learning and the number of neurons. The less neurons, the slower the learning.

4.5 Discussion of the results

The insights that we gain from this project, the difficulty in training (mostly the heavy computational cost), lead us to believe that the use of TCNN cells for simple control problems is probably neither necessary nor beneficial. As is often the case in engineering, simpler solutions are better solutions and the greater complexity of TCNN cells is most certainly not useful for simple problems such as the cart-pole problem.

This doesn't make TCNN cells uninteresting research objects. Their use might prove more interesting as a replacement for LSTMs, for instance as a last layer before control for a vision task requiring some form of memory (this is suggested by Lechner et. al. in [15]), but we did not manage to get successful results on a task requiring memory. We did see improvements on some (for instance on the *Mujoco reacher* benchmark), but we were not able to get them to converge in a sufficiently short time (we limited our training runs to $10h$).

4.6 Further work

The main bottleneck in this project has been the implementation of a training environment for a TCNN cell, and the software engineering work associated with that task. This was achieved starting from the *DERL* framework but the limitations of said framework (the absence of well implemented parallelization in the version that I am using) limit the range of problems for which experiments can be run. TCNN cells are theoretically able to implement some notion of memory but the simple cart-pole problem we considered doesn't allow us to investigate this question (the cart-pole problem doesn't require memory to be solved). To characterize the ability of TCNN cells to solve that problem, we would need to be able to train them much faster and parallelization would greatly help in that regard. Attempts at implementing this were made during the project were abandoned because of time-limitations.

Furthermore the similarity to biological neurons is certainly worth more investigation. Building LTC cells from known connectomes (copying the connection diagrams from known neural circuits) is perhaps a way of gaining insights on the inner-workings of biological neural circuits as well as a source of inspiration for the design of artificial neural networks. A first investigation into that question has been performed by Lechner et al. in [12] but not in a systematic way (only a single neural circuit has been experimented with by the authors).

Finally given the excellent performance of time-continuous neural networks for the modeling of dynamical system, an investigation into the opportunities to use them in the context of model-based reinforcement learning may certainly be interesting.

References

[1] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (1983), pp. 834–846. DOI: 10.1109/TSMC.1983.6313077.

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

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

[4] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control." In: *IROS*. IEEE, 2012, pp. 5026–5033. ISBN: 978-1-4673-1737-5. URL: http://dblp.uni-trier.de/db/conf/iros/iros2012.html#TodorovET12.

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

[6] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *ArXiv* abs/1707.06347 (2017).

[7] 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.

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

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

[10] 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.

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

[12] Mathias Lechner et al. "Designing Worm-inspired Neural Networks for Interpretable Robotic Control". In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 87–94. DOI: 10.1109/ICRA.2019.8793840.

[13] 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.

[14]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.

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

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

[17]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.

[18]Berfin Simsek et al. "Geometry of the Loss Landscape in Overparameterized Neural Networks: Symmetries and Invariances". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 9722–9732. URL: https://proceedings.mlr.press/v139/simsek21a.html.

[19]Johanni Brea. *Lecture notes for the course Artificial Neural Networks*. 2022. URL: https://moodle.epfl.ch/pluginfile.php/2779866/mod_resource/content/3/deeprl.pdf.

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

[21]Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. http://pybullet.org. 2016–2021.