# 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 [3] 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 [4]. 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:

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

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, a fixed parameter vector $A$ and a learnable parameter vector $\theta$. 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 $\Delta t$ to compute $\boldsymbol{x}_{t+\Delta t} = \int_t^{t+\Delta t} \frac{\partial \boldsymbol{x}_t}{\partial t} + \boldsymbol{x}_t$.

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

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:

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

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.

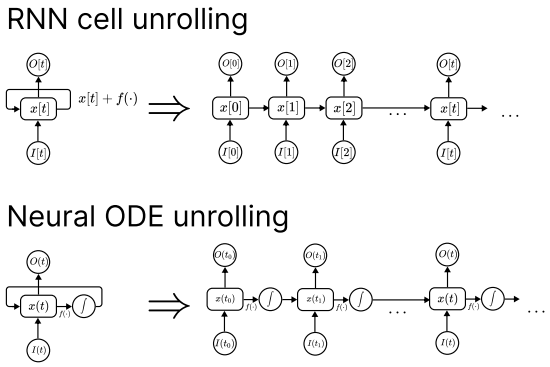


**Fig. 1**: An illustration of the difference between Neural ODE unrolling v.s. RNN unrolling.

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 [3] (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:

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

where $\tau$ 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 $\theta$.

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)" [5]. 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:

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

In practice, Hasani and Lechner propose to use the following activation function: $f(\boldsymbol{x}_t, \boldsymbol{I}_t, \theta, A) = \tanh(w_r \boldsymbol{x} + w\boldsymbol{I} + \mu)$, which is loosely connected to the non-linearity observed in synaptic dynamics between biological neurons [4].

**Table 1** Time-Continuous Neural Network Classes

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

### *1.2 Training*

**Discuss BPTT and the adjoint method here**

## 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) [2], the algorithm that we will use to train our model.

### *2.1 Markov Decision Processes*

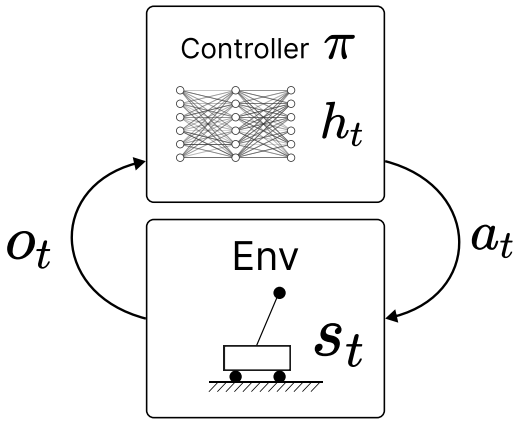


**Fig. 2**: A visual representation of the different components required to define an RL problem. An environment, which keeps state and defines the transition probability function, and a policy function (which may have a hidden state $h$) which takes observations $o_t$ of $s_t$ as inputs and returns 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 $R : (s_{t+1}, s_t, a_t) \to \mathbb{R}$.

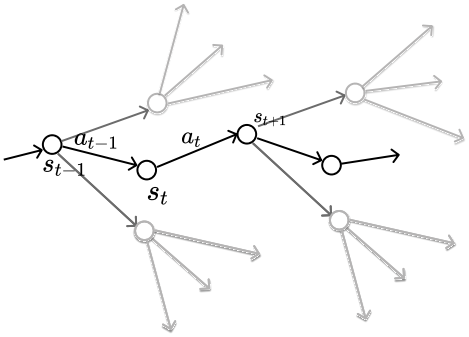


**Fig. 3**: One can think of a Markov Decision Process as a tree of possible states $s_t, s_{t+1}, ...$ connected by branches associated with the transition probability function $P(s_{t+1}|s_t, a_t)$.

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 $\pi$ (where the parameters are denoted $\theta$) 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 $\theta$ of the policy $\pi$. 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 $\tau$ denotes the set of all trajectories $s_0, a_0, s_1, a_1, ...$ in the state space under policy $\pi_\theta$. 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. 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 that in order to use an algorithm such as A2C we will have require two separate networks : a policy net (to implement a policy function) and a a critic net (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.

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

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 $R$:

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

### 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

## 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] 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.

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

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

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