Adaptive integration time for Neural Ordinary Differential Equations

Lucas Grasso Ramos
University of Buenos Aires
lucasgrassoramos@gmail.com

Abstract

This work introduces *Adaptive Integration Time* (AIT), a method that allows Neural Ordinary Differential Equations (NODEs) to dynamically learn the optimal per-input integration time. AIT augments the ODE state with a learned halting unit whose accumulated activation triggers a solver event that terminates integration, yielding a per-input depth of computation. The mechanism is deterministic, differentiable, applicable to any NODE, and trained with a regularization term that encourages less computate.

1 Introduction

In many tasks, the computational effort required to process different inputs varies significantly and unpredictably, and in many cases is unrelated to input size. In the Neural Ordinary differential Equations (NODEs) paradigm [Chen et al., 2019], the depth of compute is governed by the integration time of the state trajectory. However, standard NODEs require that the integration time be defined beforehand, which impedes the capacity to dynamically allocate computational resources based on the input difficulty.

In a discrete setting, algorithms such as Adaptive Computation Time (ACT) [Graves, 2017] and PonderNet [Banino et al., 2021] successfully allow RNNs to dynamically halt computation. Bridging this to continuous-time models requires a mechanism to implicitly terminate integration.

We define a continuous time analogue of ACT, leveraging the mathematical framework of the Neural Event ODE [Chen et al., 2021] and event handling for ODE solvers [Shampine and Thompson, 2000], allowing the network to implicitly determine its own halting time by continuously accumulating a halting probability, governed by a *halting unit* h whose activation represents the instantaneous rate of halting. Computation terminates dynamically via a threshold-based event function the exact moment this accumulated probability reaches 1. The resulting continuous halting distribution is then used to define a mean-field expected output for the system.

While solving for the minimal compute implies determining the incomputable Kolmogorov complexity of the data (8.2.1), we encourage the network to find solutions with minimal "compute" by introducing a regularization term to the loss function.

Previous work [Chen et al., 2021] allows for differentiation through event functions, so this allows for the differentiation of our approach (8.1).

2 Related Work

Recent literature has recognised the limitation of fixed integration times in Neural ODEs. For example, Anumasa and Srijith [2021] proposed the Adaptive Latent Time NODE (ALT-NODE), which addresses this by treating the end integration time as a latent variable and using variational

Preprint.

inference to predict an input dependent posterior distribution over stopping times prior to solving the ODE. While being useful for uncertainty estimation, their approach relies on a priori stochastic sampling. In contrast, our AIT-NODE evaluates halting dynamically during the ODE integration, and terminates deterministically without variational approximations. On the other hand, the Learnable Final-Time NODE (LFT-NODE) [Pang et al., 2024] reformulates the training of the NODE as a final time free optimal control problem. They make the final integration time an explicit, learnable parameter of the neural network. While it relaxes the fixed-time constraint, it does not learn a per-input specific integration time, and also does not dynamically halt based on the specific difficulty of the input currently passing through the network. Finally, the adaptive-depth Neural ODE [Massaroli et al., 2021] determines a per-input integration time with a trainable neural network, but the integration time depends only on the initial state, not the whole state trajectory.

3 Preliminaries

3.1 Neural ODEs (NODEs)

In models such as recurrent neural networks, the hidden state $z \in \mathbb{R}^D$ at a layer $t+1 \in \{1 \cdots T\}$ can be described as

$$h_{t+1} = h_t + f(h_t, \theta_t). \tag{1}$$

These iterative updates can be seen as an Euler discretization of the following differential equation (Haber and Ruthotto [2017], Ruthotto and Haber [2018], Chen et al. [2019], Lu et al. [2020]).

$$\frac{dz(t)}{dt} = f(z(t), t, \theta). \tag{2}$$

and starting from layer $z(t_0)$, we can define the output at time T, $z(T)$ as the solution to a black-box ODE solver [Chen et al., 2019]

$$\begin{aligned} z(T) &= ODESolve(z(t_0), f, t_0, T, \theta) \\ &= z(t_0) + \int_{t_0}^{T} f(z(\tau), \tau, \theta) d\tau. \end{aligned} \tag{3}$$

Reverse-mode automatic differentiation of the ODE solutions can be performed via the adjoint method [Chen et al., 2019].

3.2 Event Handling

Event handling in the context of ODE solutions [Shampine and Thompson, 2000], allows the specification of a termination criterion via an event function. Let $g(t, z(t), \phi)$ be an event function with ϕ denoting a set of parameters specific to that function. An ODE solver equipped with event handling terminates (halts) integration at the first occurrence of the event function crossing 0. An ODE solver with event handling capabilities outputs:

$$T^*, z(T^*) = ODESolveEvent(z(t_0), f, g, t_0, \theta, \phi). \tag{4}$$

T^* cannot be determined beforehand, as it depends on the trajectory $z(t)$ (It is a function of z_0, t_0, θ and ϕ). It is important to note that $ODESolveEvent$ generalises $ODESolve$ given that we can always define $g(t, z(t), \phi) = T - t$.

Chen et al. [2021] provides the necessary framework for reverse-mode automatic differentiation through an ODE solver equipped with event handling.

3.3 Adaptive Computation Time (ACT)

The Adaptive Computation Time [Graves, 2017] algorithm allows a recurrent network to learn how many computational steps to take between receiving an input and emitting an output, rather than fixing that number in advance. Given a sequence $(x_1, \cdots, x_K)$, the core idea is to augment the network with a sigmoidal *halting unit* that, at each input step k and each intermediate step n, produces

a halting probability $h_k^n \in (0,1)$. At input step k, computation stops at the first intermediate step $N(k)$ where the accumulated probability reaches a threshold:

$$N(k) = \min \left\{ n : \sum_{i=1}^{n} h_k^i \geq 1 - \epsilon \right\}. \tag{5}$$

The leftover probability mass is assigned to a *remainder* $R(k) = 1 - \sum_{i=1}^{N(k)-1} h_k^i$, so that the halting probabilities $\{p_k^n\}$ form a valid distribution over the $N(k)$ steps. The output and state at step k are obtained as the expected value (mean field) under this distribution:

$$\bar{s}_k = \sum_{n=1}^{N(k)} p_k^n \, s_k^n, \qquad \bar{y}_k = \sum_{n=1}^{N(k)} p_k^n \, y_k^n. \tag{6}$$

To prevent the network from *pondering* indefinitely, the computational cost is penalised. The *ponder* cost at step k is defined as $\rho_k = N(k) + R(k)$ and added to the loss through a hyperparameter $\tau \geq 0$:

$$\hat{\mathcal{L}} = \mathcal{L} + \tau \sum_{k=1}^{K} \rho_k. \tag{7}$$

ACT is deterministic and differentiable.

4 Adaptive Integration Time (AIT)

Consider a NODE, where $f : \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^p \to \mathbb{R}^d$ is a Neural Network parametrised by $\theta \in \mathbb{R}^p$. We wish to learn the per-input halting or integration time T^*. To do so, we build on the event-handling framework of the Neural Event ODE [Chen et al., 2021], which provides differentiable integration up to event locations defined by an event function g. Whereas the Neural Event ODE models a sequence of events, each triggering an instantaneous state update, we use this machinery in a restricted form: a single terminal event that halts integration, with no instantaneous jump.

If we treat the optimal halting time T^* as a random variable, we can define the continuous accumulator $A(x(t), t)$ as the Cumulative Distribution Function (CDF) of the halting time:

$$A(x(t), t) = P[t_0 \leq T^* \leq t]. \tag{8}$$

Given the monotony of A, either $A(x(t), t)$ is the CDF of a continuous random variable or it can be approximated by an integral. We semantically treat is as the former. We know nothing *a priori* about the optimal distribution of T^*, so we model its density as a neural network $h : \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^q \to \mathbb{R}_{>0}$ parametrized by $\psi \in \mathbb{R}^q$. We define $\Theta = [\theta, \psi]$. The accumulated probability up to time t is therefore the following integral:

$$A(x(t), t) = \int_{t_0}^{t} h(x(\tau), \tau, \psi) d\tau. \tag{9}$$

This definition is useful because we can include it in an augmented state and accumulate $A(x(t), t)$ easily, given that $\dfrac{dA(t)}{dt} = h(x(t), \tau, \psi)$.

Therefore, computation should terminate at the first instant when $A(x(t), t)$ accumulates to 1, so we define the integration time or halting time of x, T^* as:

$$T^* = \inf_{t \geq t_0} \{ A(x(t), t) = 1 \}. \tag{10}$$

Therefore, integration must be terminated in a root of the event function $g(x(t), t)$

$$g(x(t), t) = 1 - A(x(t), t). \tag{11}$$

This can be done via Event handling in ODE solvers. The parameters ϕ of g are notationally omitted given that they are empty.

While the unconstrained neural network h is not strictly a probability density function globally, our event function $g(x(t), t)$ explicitly terminates integration at the exact instant where the cumulative probability reaches 1. This truncation guarantees that h acts as a probability density function over the interval $[t_0, T^*]$, which we can use to define the expected or mean-field output $\bar{x} = \mathbb{E}_h[x] \in \mathbb{R}^d$ over the halting distribution, following the ACT paradigm. Defining the expected output over the halting distribution makes sense because it weighs more states were the model is confident to stop, i.e. closer to an answer.

We then extend the ODE state to a $\mathbb{R}^{2d+1}$ vector as follows, with the following initial conditions.

$$z(t) = \begin{bmatrix} x(t) \\ A(x(t), t) \\ \bar{x}(t) \end{bmatrix}, \quad \frac{dz(t)}{dt} = F(t) = \begin{bmatrix} f(x(t), t, \theta) \\ h(x, t, \psi) \\ h(x, t, \psi)x(t) \end{bmatrix}, \quad z(t_0) = \begin{bmatrix} x(t_0) \\ 0 \\ 0 \end{bmatrix}. \tag{12}$$

And we can obtain the output x, the accumulated halting probability and the mean-field output $\bar{x}$, at time T^* as:

$$T^*, z(T^*) = ODESolveEvent(z(t_0), F, g, t_0, \Theta, \emptyset). \tag{13}$$

One can now return the state of the trajectory at the end time, $x(T^*)$ or the mean-field or expected output over the halting probability distribution (as per [Graves, 2017]) $\bar{x}(T^*)$. We follow the latter approach in all experiments.

We define the *AIT-NODE* as an extended NODE with the AIT method. The proposed method is general so any NODE can be extended. It is deterministic and differentiable ([Chen et al., 2021], 8.1).

4.1 Limiting integration time

As before mentioned, our goal is to limit computation on NODEs, so we introduce a regularization term to the loss function that is related to our computation cost measure, T^*. Given inputs $X = (x_1, \cdots, x_K)$ and labels $Y = (y_1, \cdots, y_K)$ we define the ponder cost of X as the mean integration time:

$$\mathcal{T}(X, \Theta) = \frac{1}{K} \sum_{i=1}^{K} T^*_{x_i}. \tag{14}$$

We then introduce a *ponder penalty* hyper-parameter $\lambda \in \mathbb{R}_{\geq 0}$ and define the following cost function:

$$\hat{\mathcal{L}}(X, \Theta) = \mathcal{L}(X, \Theta) + \lambda \mathcal{T}(X, \theta). \tag{15}$$

Were $\mathcal{L}$ represents the task loss and both $\mathcal{L}$ and $\mathcal{T}$ are evaluated against $\bar{x}(T^*)$.

Given that h is strictly positive, if we integrate for long enough A will eventually reach 1. However, it can be desirable to bound the maximum integration time to a constant T_{max}. We have that:

$$h(x(\tau), \tau, \psi) \geq \frac{1}{T_{max} - t_0} \implies \exists t \in [t_0, T_{max}] : \int_{t_0}^{t} h(x(\tau), \tau, \psi) d\tau = 1. \tag{16}$$

With this in mind, h is redefined as:

$$\tilde{h} = h + \frac{1}{T_{max} - t_0}. \tag{17}$$

4.2 Relationship to ACT

Our method can be understood as the *continuous-time* analogue of ACT: the cumulative sum $\sum_n h_k^n$ is replaced by the integral $A(x, t) = \int_{t_0}^{t} h \, d\tau$, the min-based stopping criterion is replaced by a solver event, and the mean-field average $\bar{s}_k$ by the expectation $\bar{x}$ over the continuous halting distribution. The remainder term is not required here as we can always ensure that $A(x, T^*) = 1$.

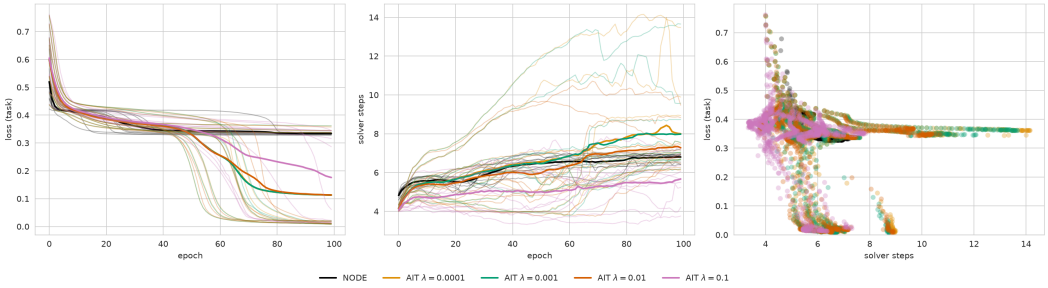

Figure 1: $g^{(1)}$ training curves. Bold lines show the mean over runs and faint ones represent each individual run. **Left:** task loss training curves. **Middle:** solver steps versus epochs. **Right:** solver steps versus task loss.

5 Experiments

The experiments were carried out in the `jax` [Bradbury et al., 2018] python framework. We used `diffrax` [Kidger, 2021], with the Tsit5 solver [Tsitouras, 2011], to solve ODEs. Neural Networks were defined using `equinox` [Kidger and Garcia, 2021]. All networks were trained using the ADAM [Kingma and Ba, 2017] optimizer with a learning rate of 0.001, and a batch size of 256.

5.1 Concentric Annuli

It was proven by Dupont et al. [2019] that Neural ODEs cannot represent the reflection map, or more generally, for $r_1 < r_2 < r_3$ and d $\in \mathbb{N}$, a NODE cannot represent $g^{(d)}_{r_1,r_2,r_3}$:

$$
\begin{aligned}
&g^{(d)}_{r_1,r_2,r_3} : \mathbb{R}^d \to \mathbb{R} \\
&g^{(d)}_{r_1,r_2,r_3}(x) = \begin{cases} 1 & \text{if } ||x|| \leq r_1 \\ -1 & \text{if } r_2 \leq ||x|| \leq r_3. \end{cases}
\end{aligned} \tag{18}
$$

It was discussed at Massaroli et al. [2021], that an Adaptive-Depth Neural ODE can learn g_1 without intersecting flows. We test that hypothesis with our AIT-NODE, given that is a more general approach to a Depth-Adaptive Neural ODE because integration time depends on the whole state trajectory in AIT. For the following experiments we used $r_1 = 0.5, r_2 = 1, r_3 = 1.5$ and defined f as a depth 2 MLP with a hidden dimension of 64. The tanh activation function was used. h was defined similarly but with depth 1. The time was concatenated to the state. All experiments were ran 10 times, for 100 epochs, with 1000 points in the inner annulus and 2000 on the outer. Softmax cross-entropy loss was used for training and models were scored using sign-accuracy. A initial bias of 1 was set for the halting MLP.

It can be seen in Fig. 1 that despite being initialisation-specific, *AIT-NODEs* can learn $g^{(1)}$, whilst NODEs do not. This shows the representation capabilities of the *AIT-NODE* and supports what has been discussed at Massaroli et al. [2021].

Regarding $g^{(2)}$, Vanilla NODEs can approximate the function but at doing so require more solver steps. (Fig. 2). Observe that the model chooses to integrate longer on the "trapped" inner class (Fig. 3).

6 Data Availability statement

The code to reproduce the results provided in the experiments is available publicly at the `https://github.com/LucasGrasso/AIT` repository.

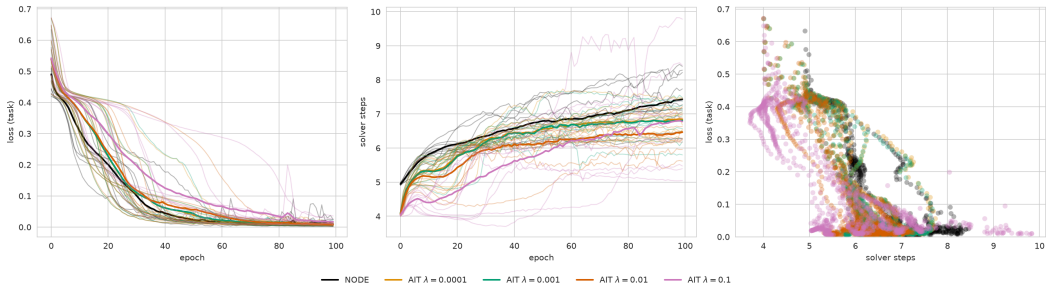

Figure 2: $g^{(2)}$ training curves. Bold lines show the mean over runs and faint ones represent each individual run. **Left:** task loss training curves. **Middle:** solver steps versus epochs. **Right:** solver steps versus task loss.

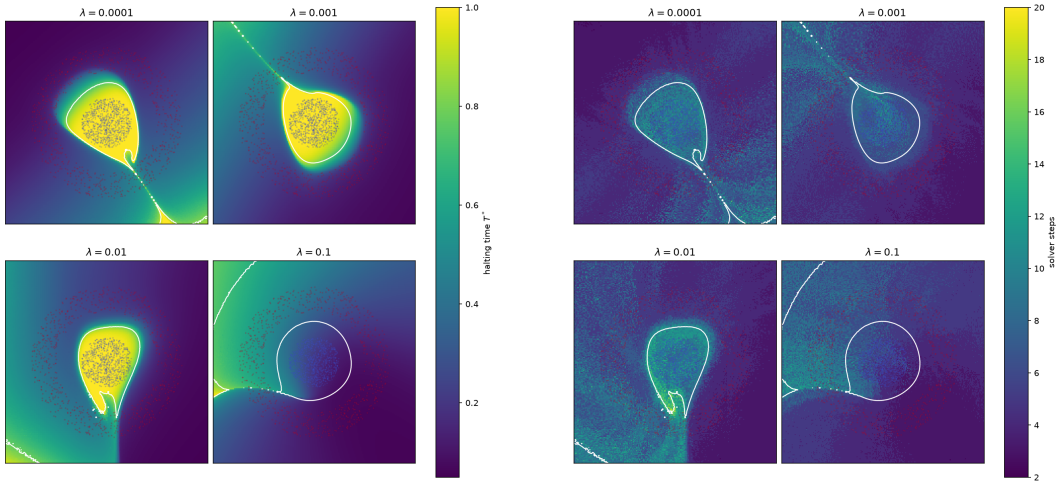

Figure 3: **Left:** $T*$ heatmap for the inputs of $g^{(2)}$. **Right:** Solver steps heatmap for the inputs of $g^{(2)}$. The best model (decided by score, tie-breaked by loss) of each λ over the 10 runs was used. The decision boundary is drawn in white.

7 Acknowledgements

The authors used Claude Opus 4.7 and Claude Opus 4.8 (Anthropic) for coding assistance and coding revisions. All code, results, equations, and conclusions were manually verified by the authors.

References

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019. URL `https://arxiv.org/abs/1806.07366`.

Alex Graves. Adaptive computation time for recurrent neural networks, 2017. URL `https://arxiv.org/abs/1603.08983`.

Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder, 2021. URL `https://arxiv.org/abs/2107.05407`.

Ricky T. Q. Chen, Brandon Amos, and Maximilian Nickel. Learning neural event functions for ordinary differential equations, 2021. URL `https://arxiv.org/abs/2011.03902`.

L.F. Shampine and S. Thompson. Event location for ordinary differential equations. *Computers & Mathematics with Applications*, 39(5):43–54, 2000. ISSN 0898-1221. doi: https://doi.org/10.1016/

S0898-1221(00)00045-6. URL https://www.sciencedirect.com/science/article/pii/S0898122100000456.

Srinivas Anumasa and P. K. Srijith. Latent time neural ordinary differential equations, 2021. URL https://arxiv.org/abs/2112.12728.

Dong Pang, Xinyi Le, Xinping Guan, and Jun Wang. Lft: Neural ordinary differential equations with learnable final-time. *IEEE Transactions on Neural Networks and Learning Systems*, 35(5): 6918–6927, 2024. doi: 10.1109/TNNLS.2022.3213308.

Stefano Massaroli, Michael Poli, Jinkyoo Park, Atsushi Yamashita, and Hajime Asama. Dissecting neural odes, 2021. URL https://arxiv.org/abs/2002.08071.

Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, December 2017. ISSN 1361-6420. doi: 10.1088/1361-6420/aa9a90. URL http://dx.doi.org/10.1088/1361-6420/aa9a90.

Lars Ruthotto and Eldad Haber. Deep neural networks motivated by partial differential equations, 2018. URL https://arxiv.org/abs/1804.04272.

Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations, 2020. URL https://arxiv.org/abs/1710.10121.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Patrick Kidger. *On Neural Differential Equations*. PhD thesis, University of Oxford, 2021.

Ch Tsitouras. Runge–kutta pairs of order 5 (4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications*, 62(2):770–775, 2011. doi: 10.1016/j.camwa.2011.06.002.

Patrick Kidger and Cristian Garcia. Equinox: neural networks in JAX via callable PyTrees and filtered transformations. *Differentiable Programming workshop at Neural Information Processing Systems 2021*, 2021.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes, 2019. URL https://arxiv.org/abs/1904.01681.

8 Appendix

8.1 Gradient Analysis

Following Eq. 12 of [Chen et al., 2019], we have that

$$\frac{d\hat{\mathcal{L}}}{d\Theta} = v^T \frac{\partial z(T^*)}{\partial \Theta} \tag{19}$$

$$v = \frac{d\hat{\mathcal{L}}}{dT^*}\left[\frac{-dg_{root}(T^*, z_0, t_0, \Theta)^{-1}}{dt}\frac{\partial g(T^*, z(T^*)}{\partial z}\right] + \frac{\partial \hat{\mathcal{L}}}{\partial z(T^*)}. \tag{20}$$

where $dg_{root}(t, z_0, t_0, \Theta) = g(t, ODESolve(z_0, F, t_0, t, \Theta))$.

We have that

$$\frac{d\hat{\mathcal{L}}}{dT^*} = \lambda + \left(\frac{\partial \mathcal{L}}{\partial \bar{x}}\right)^T \tilde{h}(x(T^*), T^*, \psi)x(T^*) \tag{21}$$

$$-\frac{dg_{root}(T^*, z_0, t_0, \Theta)^{-1}}{dt}\frac{\partial g(T^*, z(T^*)}{\partial z} = \begin{bmatrix} 0 \\ 1 \\ -\frac{1}{\tilde{h}(x(T^*), T^*, \psi)} \\ 0 \end{bmatrix} \tag{22}$$

$$\frac{\partial \hat{\mathcal{L}}}{\partial z(T^*)} = \begin{bmatrix} 0 \\ 0 \\ \frac{\partial \mathcal{L}}{\partial \bar{x}(T^*)} \end{bmatrix}. \tag{23}$$

So

$$v = \begin{bmatrix} 0 \\ -\frac{1}{\tilde{h}(x(T^*), T^*, \psi)}\frac{d\hat{\mathcal{L}}}{dT^*} \\ \frac{\partial \mathcal{L}}{\partial \bar{x}(T^*)} \end{bmatrix}. \tag{24}$$

The vector-Jacobian product (VJP) necessary to calculate $\frac{d\hat{\mathcal{L}}}{d\Theta}$ can be computed with a single $ODESolve$ [Chen et al., 2021]. This proves that our approach is differentiable.

8.2 Additional Proofs

8.2.1 Kolmogorov Complexity is incomputable

Let $\mathcal{U}$ be a prefix-free Universal Turing Machine.
Suppose $Kolmogorov(s) = \min\{\text{length}((P) : \mathcal{U}(P) = s\}$, the minimum program size of a program that outputs a string s, is computable. Then define the algorithm $P(N)$ as 1.

Because the number of programs of length $\leq N$ is finite ($2^{N+1} - 1$), but the number of strings is infinite, such a string s is guaranteed to exist. P will eventually find it, return it, and halt.

The loop, the string generation, and the return statement all take some constant number of bits, c_1, c_2, c_3. The Kolmogorov subroutine takes a constant number of bits, c_4 and the integer N can be encoded in $\log_2(N)$ bits. Therefore, the total length of the program P is roughly $\log_2(N) + c$, where $c = c_1 + c_2 + c_3 + c_4$.

By definition, the string s output by P has a Kolmogorov complexity of $K(s) > N$. However, P is literally a program that outputs s. Therefore:

$$N < K(s) \leq \text{length}(P) = \log_2(N) + c$$

Now choose N large enough such that $\log_2(N) + c \leq N$ Which results in a contradiction, therefore $Kolmogorov$ is not computable.

Algorithm 1 $P(N)$

Require: An integer $N \in \mathbb{N}$
Ensure: A binary string s such that $K(s) > N$

procedure P(N)
    $\mathcal{S} \leftarrow$ List of all binary strings sorted by length $(\epsilon, 0, 1, 00, \dots)$
    for each $s \in \mathcal{S}$ **do**
        $k \leftarrow Kolmogorov(s)$
        if $k > N$ **then**
            return s
        end if
    end for
end procedure