dmturi

9/5/18

Derivatives of Turing Machines in Linear Logic

The aim of this talk is to explain how to differentiate a Turing machine. More precisely: in a remarkable paper in 2003 Ehrhard and Regnier defined the derivative of <u>any</u> algorithm with respect to one of its inputs in the setting of lambda calculus. The derivative of an algorithm is itself a kind of algorithm, but in an extended language called <u>differential lambda calculus</u>:

[1] T. Ehrhard and L. Regnier, "The differential lambda-calculus", Theoretical Computer Science (2003).

They prove some strong results about this system, but when I started looking into this with my then masters student James, we were struck on the one hand by how radical it seems to claim that <u>any algorithm has a derivative</u> and on the other hand by how inscrutable we found the answer. It wasn't clear to us what it was that the derivative of an algorithm <u>computes</u>.

Part of the problem, we decided, was that while Turing machines and lambda calculus are equivalent formalisations of the intuitive notion of an algorithm, in the sense that the same class of functions $\mathbb{N} \longrightarrow \mathbb{N}$ may be encoded in both (the computable functions), these two models of computation are in other ways very different. For example: it is more intuitive to program in Turing machines than in lambda calculus. So roughly speaking, we set out to see <u>what the Ehrhard-Regnier derivative meant for Turing machines</u>, in order that we might obtain a more conceptual understanding of the derivative of algorithms. Today I'd like to present what we found. Our joint work on this is spread across three papers, the last two of which we are putting the finishing touches on currently.

⌜<u>Aside</u>: if you enjoy math on the border of logic, categories and computation, check out our seminar at http://therisingsea.org/post/seminar-ch/ ⌟

[2] J. Clift and D. Murfet "Cofree coalgebras and differential linear logic" arXiv: 1701.01285.

[3] J. Clift and D. Murfet "Encodings of Turing machines in linear logic", in prep.

[4] J. Clift and D. Murfet "Derivatives of Turing machines in linear logic," in prep.

Outline of the talk

1. Introduction to Turing machines / why derivatives don't make sense
2. Naive Bayesian Turing machines
3. The derivative of a Turing machine
4. Application: gradient descent

Introduction A Turing machine M is a tuple (Σ, Q, δ) where

Σ – finite tape alphabet ($\square \in \Sigma$ is called blank)
Q – finite set of states
$\delta : \Sigma \times Q \longrightarrow \Sigma \times Q \times \{L, R\}$ – transition function

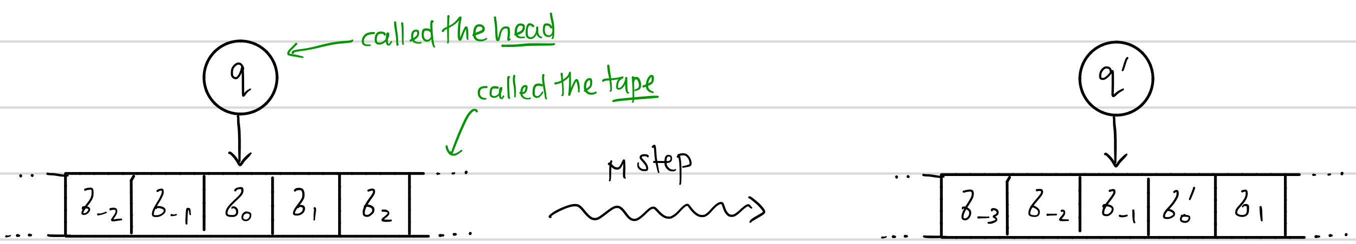
A configuration of M is an element of $\Sigma^{\mathbb{Z},\square} \times Q$ where

$$\Sigma^{\mathbb{Z},\square} = \{ f : \mathbb{Z} \to \Sigma \mid f(n) = \square \text{ for all but finitely many } n \}$$

The step function of M is the function

$$ {}_M\text{step} : \Sigma^{\mathbb{Z},\square} \times Q \longrightarrow \Sigma^{\mathbb{Z},\square} \times Q $$

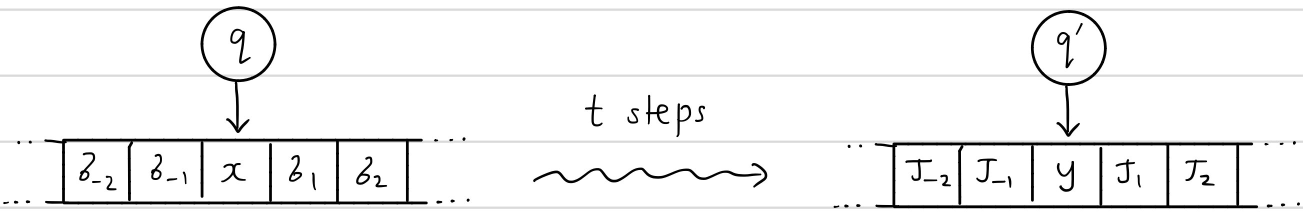
derived from δ according to the scheme



(here $f : \mathbb{Z} \to \Sigma$, $f(n) = z_n$) (here $f(0) = z_{-1}$, $f(1) = z_0'$, ...)

$$ \text{if} \quad \delta(z_0, q) = (z_0', q', L) \qquad (\text{similarly for } R) $$

Derivatives? Suppose we run M for t steps, and call the contents of the tape squares under the head at time 0 and time t by x, y respectively.



Clearly $y = y(x)$ depends on x (viewing the z_i as fixed), and the pair (x, y) determine a function $f : \Sigma \longrightarrow \Sigma$.

- Since Σ is finite and discrete, f has no meaningful derivative, but
- there is more information in the algorithm M than in the function f, and
- from this information we can extract a meaningful tangent map Tf.

But <u>why should anyone care</u> if TMs can be differentiated or not? Let me give two quick reasons:

1) Algorithms are fundamental objects of mathematics, if they admit an intrinsic and conceptually meaningful notion of derivative this is a big deal.

2) Differentiating TMs is the key to making sense of "spaces" of TMs, in a way that may have applications in e.g. Machine Learning. More on this at the end of the talk.

2 Naive Bayesian probability

Conceptually the meaning of the derivative of F is grounded in a version of Bayesian probability, as I will now explain. Ultimately, however, the derivative is justified on technical grounds because it is the Ehrhard-Regnier derivative of an encoding of F into linear logic (as I will explain in part 4).

<u>Bayesian probability</u> axiomitises a (partial) function

$$
\begin{aligned}
\mathbb{P} : \mathcal{A} \times \mathcal{A} &\longrightarrow [0,1] \\
(p, q) &\longmapsto \mathbb{P}(p \mid q)
\end{aligned}
$$

typically $X = x$, where X is a random variable

where $\mathcal{A}$ is a Boolean algebra $(\wedge, \vee, \neg, 0, 1)$ of propositions (Cox 1946). We read $\mathbb{P}(p \mid q)$ as the conditional probability of p given q, thought of as a <u>degree of belief</u> assigned by an observer. Frequentists take $\mathbb{P}(p \mid q) = \mathbb{P}(p \wedge q) / \mathbb{P}(q)$ whenever this make sense, but Bayesians axiomitise $\mathbb{P}(p \mid q)$ directly.

<u>Probabilistic step</u> Given a set Z we write

$$
\Delta Z = \Big\{ \sum_{z \in Z} \lambda_z z \in \mathbb{R} Z \;\Big|\; \lambda_z \geqslant 0 \text{ for all } z \text{ and } \sum_z \lambda_z = 1 \Big\} \subseteq \mathbb{R} Z .
$$

for the space of (finitely supported) probability distributions on Z. The step function of $\mathcal{M}$ has a <u>probabilistic extension</u> $\Delta^{std}_{\mathcal{M}}\mathrm{step}$ defined by

$$
\begin{tikzcd}
\Delta(\Sigma^{\mathbb{Z},\square} \times Q) \arrow[r, "\Delta^{std}_{\mathcal{M}}\mathrm{step}"] & \Delta(\Sigma^{\mathbb{Z},\square} \times Q) \\
\Sigma^{\mathbb{Z},\square} \times Q \arrow[u, hook] \arrow[r, "{}_{\mathcal{M}}\mathrm{step}"'] & \Sigma^{\mathbb{Z},\square} \times Q \arrow[u, hook]
\end{tikzcd}
\qquad
\begin{aligned}
&\Delta^{std}_{\mathcal{M}}\mathrm{step}(C) \\
&\quad \| \\
&\sum_a \Big(\sum_{\substack{a' \\ {}_{\mathcal{M}}\mathrm{step}(a') = a}} C_{a'} \Big) \cdot a
\end{aligned}
$$

We view $C \in \Delta(\Sigma^{\mathbb{Z},\square} \times Q)$ as describing the uncertainty of a Bayesian observer about the configuration of the machine, and $\Delta_M^{std}\text{step}(C)$ as the propagation of this uncertainty to the next time step. Part of the information in a distribution like C is captured by the distributions of the random variables

$Y_u(t)$ – contents of tape square in relative pos^{n} u at time t $\quad(\Sigma)$
$S(t)$ – state at time t $\quad(Q)$

But there is additional information in C (i.e. joint distributions). Using random variables

$Mv(t)$ – direction to move at time t $\quad(R, L)$
$Wr(t)$ – symbol to write at time t $\quad(\Sigma)$

and given some initial probability C at time $t=0$, $\quad\left(\delta : \Sigma \times Q \longrightarrow \Sigma \times Q \times \{L, R\}\right)$

ⓐ $$\mathbb{P}(Mv(t) = d \mid C) = \sum_{s,q} \delta_{\delta(s,q)_3 = d}\, \mathbb{P}(Y_0(t) = s \wedge S(t) = q \mid C)$$

ⓑ $$\mathbb{P}(Wr(t) = s \mid C) = \sum_{s',q} \delta_{\delta(s',q)_1 = s}\, \mathbb{P}(Y_0(t) = s' \wedge S(t) = q \mid C)$$

ⓒ $$\mathbb{P}(S(t+1) = q \mid C) = \sum_{s,q'} \delta_{\delta(s,q')_2 = q}\, \mathbb{P}(Y_0(t) = s \wedge S(t) = q' \mid C)$$

ⓓ $$\begin{aligned}\mathbb{P}(Y_u(t+1) = s \mid C) = \; & \delta_{u \neq -1}\, \mathbb{P}(Y_{u+1}(t) = s \wedge Mv(t) = R \mid C) \\ & + \delta_{u=-1}\, \mathbb{P}(Wr(t) = s \wedge Mv(t) = R \mid C) \\ & + \delta_{u \neq 1}\, \mathbb{P}(Y_{u-1}(t) = s \wedge Mv(t) = L \mid C) \\ & + \delta_{u=1}\, \mathbb{P}(Wr(t) = s \wedge Mv(t) = L \mid C)\end{aligned}$$

Notice how we cannot compute

$$\Big(\big(\mathbb{P}(Y_u(t+1) | C) \big)_{u \in \mathbb{Z}},\ \mathbb{P}(S(t+1) / C) \Big) \in (\Delta\Sigma)^{\mathbb{Z},\square} \times \Delta Q$$

purely from the distributions for the same random variables $\{Y_u\}_{u\in\mathbb{Z}}$, S at time t, because we need to also know various joint distributions. This need to maintain joint distributions at each time step explains why "probabilistic programming" is computationally expensive. The "cheapskate" approximation to Δ_M^{std} step is to assume first of all $C \in (\Delta\Sigma)^{\mathbb{Z},\square} \times \Delta Q$ has no correlations to begin with, and compute at each step assuming conditional independence (at equal times) of pairs

$$\{Y_0, S\},\ \{Y_u, Mv\}_{u\neq 0},\ \{Wr, Mv\}$$

That is, we define inductively

ⓐ $$\mathbb{P}_{nv}(Mv(t) = d \mid C) = \sum_{z,q} \delta_{\delta(z,q)_3 = d}\ \mathbb{P}_{nv}(Y_0(t) = z \mid C) \cdot \mathbb{P}_{nv}(S(t) = q \mid C)$$

ⓑ $$\mathbb{P}_{nv}(Wr(t) = z \mid C) = \sum_{z',q} \delta_{\delta(z',q)_1 = z}\ \mathbb{P}_{nv}(Y_0(t) = z' \mid C) \cdot \mathbb{P}_{nv}(S(t) = q \mid C).$$

ⓒ $$\mathbb{P}_{nv}(S(t+1) = q \mid C) = \sum_{z,q'} \delta_{\delta(z,q')_2 = q}\ \mathbb{P}_{nv}(Y_0(t) = z \mid C) \cdot \mathbb{P}_{nv}(S(t) = q' \mid C)$$

ⓓ $$\begin{aligned}\mathbb{P}_{nv}(Y_u(t+1) = z \mid C) = {} & \mathbb{P}_{nv}(Mv(t) = R)\big\{ \delta_{u\neq -1} \mathbb{P}_{nv}(Y_{u+1}(t) = z \mid C) \\ & \quad + \delta_{u=-1} \mathbb{P}_{nv}(Wr(t) = z \mid C) \big\} \\ & + \mathbb{P}_{nv}(Mv(t) = L)\big\{ \delta_{u\neq 1} \mathbb{P}_{nv}(Y_{u-1}(t) = z \mid C) \\ & \quad + \delta_{u=1} \mathbb{P}_{nv}(Wr(t) = z \mid C) \big\}\end{aligned}$$

We call this <u>naive probability</u> since the same kind of "cheapskate" conditional independence hypotheses are used in machine learning to define naive Bayesian classifiers. Note that

$$\Big(\big(\mathbb{P}_{nv}(Y_u(t+1) | C) \big)_{u \in \mathbb{Z}}, \ \mathbb{P}_{nv}(S(t+1) / C) \Big) \in (\Delta\Sigma)^{\mathbb{Z},\square} \times \Delta Q$$

<u>can</u> be computed from the naive probability distributions of $\{ Y_u(t) \}_u, S(t)$. This defines an update rule $\Delta_M$step as in the commutative diagram

$$\begin{tikzcd}
(\Delta\Sigma)^{\mathbb{Z},\square} \times \Delta Q \arrow[r, "\Delta_M\text{step}"] & (\Delta\Sigma)^{\mathbb{Z},\square} \times \Delta Q \\
\Sigma^{\mathbb{Z},\square} \times Q \arrow[u, hook] \arrow[r, "{}_M\text{step}"'] & \Sigma^{\mathbb{Z},\square} \times Q \arrow[u, hook]
\end{tikzcd}$$

which we think of as the time evolution of a <u>naive Bayesian observer</u> of M. We call $\Delta_M$step the <u>naive proabilistic extension</u> of ${}_M$step.

(3) <u>The derivative of a Turing machine</u>

This strange probability is justified by

<u>Theorem</u> (Clift-M) The Ehrhard-Regnier derivative of a Turing machine M computes the derivatives (tangent maps) of $\Delta_M$step.

By the Ehrhard-Regnier derivative of M we mean the derivative of an encoding of the t-step function of M as a proof in linear logic. The denotation of this derivative under a particular semantics of linear logic in vector spaces gives a linear map, which is the tangent map of (a restriction of) $\Delta_M\text{step}^t$.

language of closed symmetric monoidal categories with cofree coalgebras

$$\begin{tikzcd}
\{ \text{Turing machines} \} \arrow[r, "\text{encode}"] & \{ \text{linear logic proofs} \} \arrow[d, "\text{add dual numbers}"] \arrow[dr, "{[\![-]\!]}"] & \\
& \{ \text{differential linear logic proofs} \} \arrow[r, "{[\![-]\!]}"] \arrow[loop below, "\frac{\partial}{\partial x}"] & \mathrm{Vec}_k
\end{tikzcd}$$

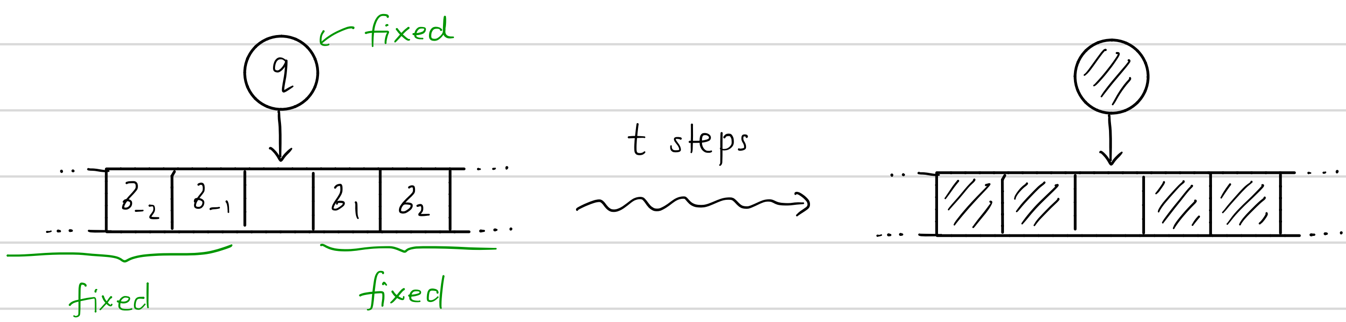
based on work of Girard

Ehrhard-Regnier derivative

Sweedler semantics (based on cofree coalgebras)

There is no time in this talk to explain linear logic and the theory of coalgebras, so for the moment the Ehrhard-Regnier derivative is just some complicated operator on terms in a formal language. I want to focus on the tangent maps of Δ_{Mstep}.

Consider running M for t steps, with a fixed initial state and fixing all the tape squares except for the one initially under the head. After t steps we read off the symbol under the head and ignore the rest of the tape and the state



This gives a function

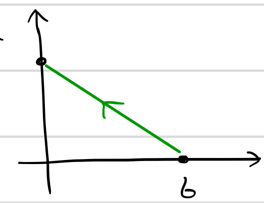
$$\Sigma \xrightarrow{\text{add } z, q} \Sigma^{\mathbb{Z},\square} \times Q \xrightarrow{\text{Mstep}^t} \Sigma^{\mathbb{Z},\square} \times Q \xrightarrow{\text{discard}} \Sigma$$

which is precisely $f : \Sigma \longrightarrow \Sigma$ from earlier. The derivative of this function doesn't make sense, because we can't infinitesimally vary an input $x \in \Sigma$. But we <u>can</u> vary such an input inside the simplex $x \in \Sigma \subseteq \Delta\Sigma$. If we allow some uncertainty about the symbol initially under the head, and propagate that uncertainty using the naive Bayesian approach, we have the bottom row of the following commuting diagram

$$\begin{tikzcd}
\Sigma \arrow[r] \arrow[d, hook] & \Sigma^{\mathbb{Z},\square} \times Q \arrow[r, "{}_M\mathrm{step}^t"] \arrow[d, hook] & \Sigma^{\mathbb{Z},\square} \times Q \arrow[r, two heads] \arrow[d, hook] & \Sigma \arrow[d, hook] \\
\Delta\Sigma \arrow[r] & (\Delta\Sigma)^{\mathbb{Z},\square} \times \Delta Q \arrow[r, "\Delta\, {}_M\mathrm{step}^t"'] & (\Delta\Sigma)^{\mathbb{Z},\square} \times \Delta Q \arrow[r, two heads] & \Delta\Sigma
\end{tikzcd}$$

Let us denote the bottom row by $\Delta f : \Delta\Sigma \longrightarrow \Delta\Sigma$, which is a smooth map of manifolds with corners. Suppose $\Sigma = \{ z, \hat{z} \}$ so that we can identify

$$\begin{aligned} \gamma : [0,1] &\xrightarrow{\ \cong\ } \Delta\Sigma \\ h &\longmapsto (1-h) z + h \hat{z} \end{aligned}$$



which gives an identification of tangent spaces

$$\begin{array}{ccccc} T_x([0,1]) & \xrightarrow{\ \cong\ } & T_{\gamma(x)}(\Delta\Sigma) & \subseteq & T_{\gamma(x)}(\mathbb{R}^2) \\ \cong & & \cong & & \\ \mathbb{R}\frac{\partial}{\partial h} & & \mathbb{R}\left(\frac{\partial}{\partial x_{\hat{z}}} - \frac{\partial}{\partial x_{z}} \right) & & \end{array}$$

and so we can consider

$$T_z(\Delta f) : T_z(\Delta\Sigma) \longrightarrow T_{f(z)}(\Delta\Sigma)$$
$$\frac{\partial}{\partial x_{\hat{z}}} - \frac{\partial}{\partial x_z} \longmapsto \lambda\left(\frac{\partial}{\partial x_{\hat{z}}} - \frac{\partial}{\partial x_z}\right)$$

Then an infinitesimal change in the input from z, viewed as an infinitesimal revision of the naive Bayesian observer's degree of belief from certainty about z to a state of uncertainty $(1-\Delta h)z + \Delta h\,\hat{z}$, propagates to a state of uncertainty about the output (assuming $f(\hat{z}) \neq f(z)$)

$$(1-\lambda\Delta h)f(z) + \lambda\Delta h\, f(\hat{z}).$$

Upshot $T_z(\Delta f)$ encodes a rate of change of belief λ, and the Ehrhard-Regnier derivative of M computes this λ.

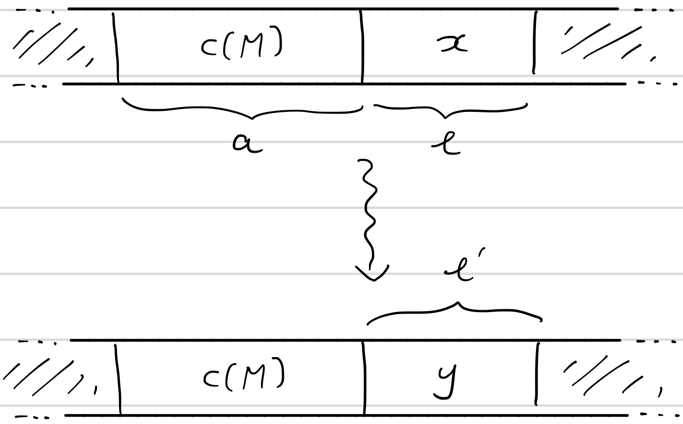
Summary Semantics of linear logic gives a natural way to propagate uncertainty through TMs, s.t. the rate of change of this uncertainty is computed by the Ehrhard-Regnier derivative of the TM.

[Note propagating uncertainty with Δ^{std} instead gives $\lambda = 1$]

(4) Application: gradient descent

In light of the above we can propagate uncertainty through arbitrary algorithms (Turing machines) in such a way that the rate of change of output uncertainty with respect to input uncertainty is computed by the Ehrhard-Regnier derivative of the algorithm. If we apply this to a Universal Turing Machine (UTM) $\mathcal{U}$ we get the following picture.

To simulate M on $\mathcal{U}$ we input the code $c(M) \in \{0,1\}^*$ on some designated part of $\mathcal{U}$'s tape, and the input $x \in \{0,1\}^*$ on another part of the tape. If M halts on input x with output y (written $M(x)=y$) then $\mathcal{U}$ halts on input $(c(M), x)$ with output y



<u>Problem</u> Given (x,y) find M s.t. $M(x)=y$ within t steps.

Restricting the step function to a fixed start state and reading off only a region of the final tape gives $\mathcal{U}\,\mathrm{step}^t : \{0,1\}^a \times \{0,1\}^\ell \longrightarrow \{0,1\}^{\ell'}$ which has a naive probabilistic extension

$$\Delta \mathcal{U}\,\mathrm{step}^t : \Delta\{0,1\}^a \times \Delta\{0,1\}^\ell \longrightarrow \Delta\{0,1\}^{\ell'}$$

One adds a regularisation term to ensure sol^{N}s are actually in $\{0,1\}^a$ i.e. are TMs

Taking the KL-divergence against (x,y) gives a smooth map

$$L := D_{KL}\big(y \,\|\, \Delta \mathcal{U}\,\mathrm{step}^t(-,x)\big) : (\Delta\{0,1\})^a \longrightarrow \mathbb{R}.$$

Gradient descent with respect to L is a way of searching the "space" of probabilistic algorithms indexed by $(\Delta\{0,1\})^a$, in which actual TMs sit as the vertices $\{0,1\}^a$, using derivatives of L and thus of $\Delta \mathcal{U}\,\mathrm{step}^t$ and thus the Ehrhard-Regnier derivatives of $\mathcal{U}$ itself.

Remark To actually perform this gradient descent on $\Delta\{0,1\}^a \cong [0,1]^a$ we have to compute at each step the distribution

$$\Delta_{\mathcal{U}} \mathrm{step}^t(\underline{h}, x) \qquad (*)$$

for some point $\underline{h} \in [0,1]^a$. For Δ^{std} this calculation has time complexity 2^a because in this case the gradient descent is essentially just trying every Turing machine with code length $\leq a$ to find one that works. Thus gradient descent using Δ^{std} is pointless. However the time complexity of computing the naive probability $(*)$ is polynomial, so in principle it is a feasible way to search for programs.

Example Consider the special case $a = 2$ and $\ell' = 1$, $y = 1 \in \Sigma$

$$L : [0,1]^2 \longrightarrow \mathbb{R}$$

stands for $(1-h)\cdot 0 + h\cdot 1 \in \Delta\{0,1\}$.

$$L(h,k) = -\ln\left(\Delta_{\mathcal{U}} \mathrm{step}^t(h,k,x)_1\right)$$

Now recall $\Delta_{\mathcal{U}} \mathrm{step}^t(-,-,x)_1 : [0,1]^2 \longrightarrow \mathbb{R}$

$$T_{(h,k)}\left(\Delta_{\mathcal{U}} \mathrm{step}^t(-,-,x)\right) : \mathbb{R}\tfrac{\partial}{\partial h} \oplus \mathbb{R}\tfrac{\partial}{\partial k} \longrightarrow \mathbb{R} \quad \text{is} \quad (\lambda_1, \lambda_2)$$

$$\therefore \quad -\frac{\partial}{\partial h} L(h,k) = \frac{\lambda_1}{\Delta_{\mathcal{U}} \mathrm{step}^t(h,k,x)_1}$$

$$-\frac{\partial}{\partial k} L(h,k) = \frac{\lambda_2}{\Delta_{\mathcal{U}} \mathrm{step}^t(h,k,x)_1}$$