Machine learning under physical constraints
Introduction to RNN

Sixin Zhang
(sixin.zhang@toulouse-inp.fr)

Outline

Recurrent Neural Networks (RNN)

Training strategies of RNN

Outline

Recurrent Neural Networks (RNN)

Training strategies of RNN

RNN for sequence processing

- Many data such as time-series, language, speech, genomics can be represented in a form of sequence: $x_1, x_2, \cdots$.
- How to process such data of various type and length?
- Two basic ideas: recurrent and convolutional.
- Goal of this lecture: define what is RNN, and to construct, train and use it.
- Reference: Ian Goodfellow, Yoshua Bengio, Aaron Courville. Deep Learning. MIT Press, 2016

What is Recurrent?

- View 1: Output of a dynamical system is fed back to some of its inputs, e.g. time-delayed system.

$$\frac{d}{dt}x(t) = f(t, x(t), x(t-\tau)), \quad \tau > 0$$

- View 2: Finite-state automata (or Turing machine)
  - Change from one state to another in response to some inputs
  - Computers are recurrent !

RNN: dynamical system view

- Assume state of a dynamical system at time t is h_t
- Classical form (θ is a parameter)

$$h_t = f(h_{t-1}; \theta)$$

- Input (external-signal) dependent form

$$h_t = f(h_{t-1}, x_t; \theta)$$

- h_t can be interpreted as hidden states in NN.

RNN: Automata view

- Automata: an abstract machine that can be in exactly one of a finite number of states at any given time

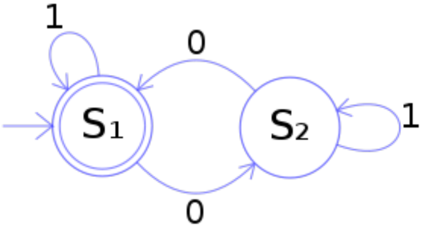

Figure: Representation of an acceptor; this example shows one that determines whether a binary number has an even number of 0s, where S1 is an accepting state and S2 is a non accepting state. See https://en.wikipedia.org/wiki/Finite-state_machine

Unrolling a dynamical system

- h_t depends on $x_t, x_{t-1}, \cdots, x_1$ and h_0.
- To see this, unroll recursively the hidden states:

$$h_t = f(h_{t-1}, x_t; \theta) = f(f(h_{t-2}, x_{t-1}; \theta), x_t; \theta)$$

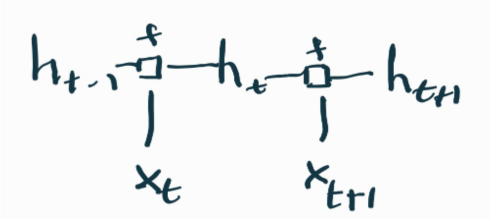

Figure: Unrolled computational graph

Definition of RNN

- RNN: a family of NN constructed from the idea of unrolling.
- Several examples:

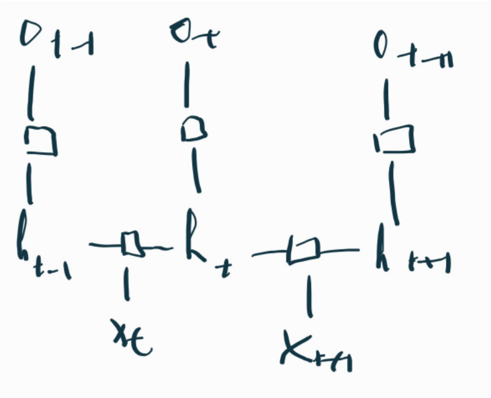

Figure: One hidden-layer RNN: allow the network's hidden units to see its (own) previous output

Example 1

- A concrete case of one hidden-layer RNN:

$$
\begin{aligned}
a_t &= Ux_t + b + Wh_{t-1} \\
h_t &= \tanh(a_t) \\
o_t &= c + Vh_t \\
\hat{y}_t &= \mathrm{softmax}(o_t)
\end{aligned}
$$

- Input-to-hidden parameters: U, b
- Hidden-to-hidden parameters: W, b
- Hidden-to-output parameters: V, c
- A loss L_t is then computed based on y_t and $\hat{y}_t$.

Example 2

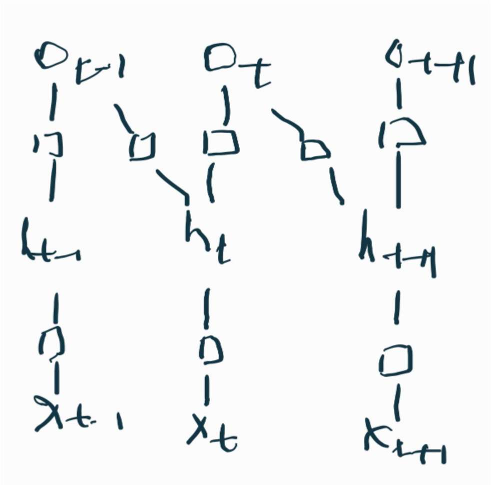

Figure: Allow the network's hidden units to see the previous output

Example 3

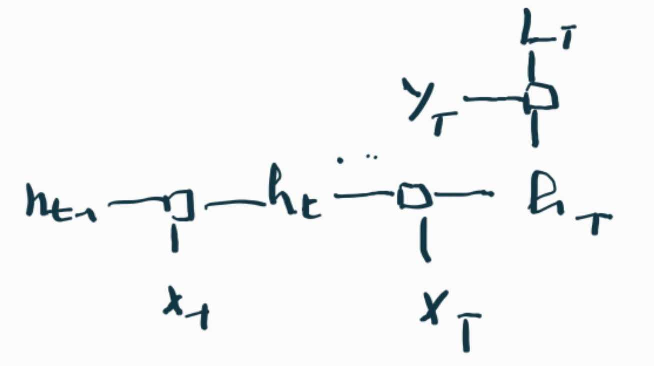

Figure: Feedback only come from the final output

Example 3

- A concrete case of TP
  - Goal: learn an optimal output o_T close to the target y_T.
  - Initial input $x_0 \sim p = \mathcal{N}(\mu, \sigma^2)$
  - $h_0 = x_0$
  - $h_t = h_{t-1} + \theta$
  - $o_T = h_T$
  - $y_T = \mu$
  - $L_T = (y_T - o_T)^2$
  - Optimization problem: $\min_{\theta \in \mathbb{R}} \mathbb{E}_{x_0 \sim p}(L_T)$

Elman RNN

- Finding structure in time (Elman 1990)
- Represent **time** by the effect it has on processing.
- Applications: sequential prediction, language understanding.

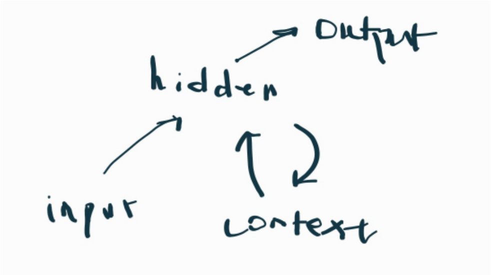

Figure: Save effects of time in a context state, aka memory

Example of Elman RNN: Data assimilation networks

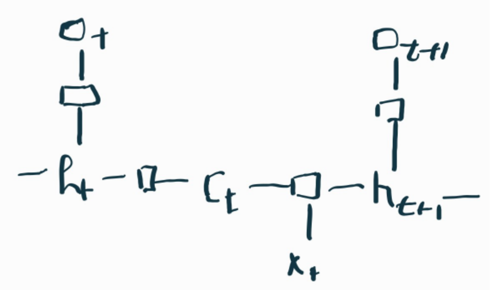

Figure: Unroll Elman RNN over time

- Relation with Elman RNN and Data assimilation networks
  - Hidden h_t: analysis posterior distribution
  - Context c_t: prediction posterior distribution
  - Input x_t: observed state in ODS

Advanced RNN

- Next-character generation (basis of Chat-GPT): **demo**
- Multiple hidden-layer (deep) RNN: audio source separation (demo)
- Bi-directional RNN: language model (read a sequence in 2 directions)
- Auto-encoder RNN: sequence to sequence language translation
- GRU, LSTM: long-dependency in signal (RNNnoise: demo)

Outline

Recurrent Neural Networks (RNN)

Training strategies of RNN

Back-propagation over time (BPTT)

- To train a RNN, one often uses the gradient of the parameters for efficient optimization. BPTT is a way to compute such gradients.
- It is based on the same idea of back-propagation in neural networks, but it is subtle due to the shared parameters across time.
- Nowadays, one can use automatic differentiation to do this. But one still needs to understand it to go further.

Example 1

- The concrete case

$$
\begin{aligned}
a_t &= Ux_t + b + Wh_{t-1} \\
h_t &= \tanh(a_t) \\
o_t &= c + Vh_t \\
\hat{y}_t &= \mathrm{softmax}(o_t)
\end{aligned}
$$

- What is the gradient of the softmax layer, $\hat{y}_t = \mathrm{softmax}(o_t)$?
- A loss L_t is computed based on y_t and $\hat{y}_t$.
- How to compute the total loss $L = \sum_t L_t$ with respect to all the parameters?

Example 1: BPTT by chain rule

- Total loss $L = \sum_t L_t$

$$\frac{\partial L}{\partial L_t} = 1$$

- From L_t to o_t

$$\nabla_{o_t} L = \left(\frac{\partial L}{\partial o_t}\right)^T = \left(\frac{\partial L}{\partial L_t}\frac{\partial L_t}{\partial o_t}\right)^T = \left(\frac{\partial L_t}{\partial o_t}\right)^T$$

- From o_t to h_t: output $o_t = c + Vh_t$

$$\nabla_{h_t} L = V^{\intercal} \nabla_{o_t} L$$

Example 1: BPTT by chain rule

- Compute gradient of h_t from h_{t+1}.
- Hidden states:

$$h_{t+1} = \tanh(Ux_{t+1} + b + Wh_t)$$

- Recursive relation

$$\nabla_{h_t} L = \left(\frac{\partial h_{t+1}}{\partial h_t}\right)^T \nabla_{h_{t+1}} L + \left(\frac{\partial o_t}{\partial h_t}\right)^T \nabla_{o_t} L$$

Example 1: BPTT by chain rule

- How about the parameters, e.g. (U, b, W)?

$$h_t = \tanh(Ux_t + b + Wh_{t-1})$$

- Idea: accumulate all the gradients over t.
- Write b as b_t to be clear of the partial derivatives:

$$\nabla_b L = \sum_t \left(\frac{\partial h_t}{\partial b_t}\right)^T \nabla_{h_t} L$$

where $h_t = \tanh(Ux_t + b_t + Wh_{t-1})$.

Deterministic (BPTT) and online (truncated BPTT)

- Initialize $\theta = \theta^{(0)}$
- Deterministic optimizer (BPTT) at each iteration k:

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla_\theta L(\theta^{(k)})$$

- Online optimizer (truncated BPTT) at iteration k:

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k \tilde{\nabla}_\theta L_{k+1}(\theta^{(k)})$$

Example 1: Truncated BPTT

- The cost (both CPU and memory) to compute $\nabla_b L$ is $O(T)$ due to the summation over $t \leq T$. This is prohibitive when T is very large.
- Truncated BPTT reduces this cost by focusing on the impact of the "current" parameter b_t on the current loss L_t.
- The truncated gradient of b at time t is

$$\tilde{\nabla}_b L_t = \left(\frac{\partial h_t}{\partial b_t}\right)^T \nabla_{h_t} L_t, \quad \nabla_{h_t} L_t = \left(\frac{\partial o_t}{\partial h_t}\right)^T \nabla_{o_t} L_t$$

- The cost to compute $\tilde{\nabla}_b L_t$ is $O(1)$.
- Discussion of p-truncated BPTT ($p = 1, 2, \cdots$) in the paper: Corentin Tallec, Yann Ollivier. Unbiasing Truncated Backpropagation Through Time.

Updates of truncated BPTT

- Principle: Step k is represented by $(\theta^{(k)}, \tilde{h}_k)$. It is updated to $(\theta^{(k+1)}, \tilde{h}_{k+1})$ using the loss at time $k+1$,

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k \tilde{\nabla}_\theta L_{k+1}(\theta^{(k)}),$$
$$\tilde{h}_{k+1} = f(\tilde{h}_k, x_{k+1}; \theta^{(k)}),$$

  where the truncated gradient

$$\tilde{\nabla}_\theta L_{k+1}(\theta^{(k)}) = \nabla_\theta \ell(f(\tilde{h}_k, x_{k+1}; \theta), y_{k+1})|_{\theta=\theta_k}.$$

- Difference to BPTT: In BPTT, the step k is represented by $\theta^{(k)}$ and updated to $\theta^{(k+1)}$ using the loss over time $t \leq T$.

Example 4: BPTT vs. Truncated BPTT

- Consider the following problem:
  - $h_0 = x_0 \sim \mathcal{N}(\mu, \sigma^2)$
  - $h_t = f(h_{t-1}; \theta) = h_{t-1} + \theta$
  - $L_t = \ell(h_t) = \mathbb{E}_{x_0 \sim p}(h_t - \mu)^2$
  - $\min_{\theta \in \mathbb{R}} L = \sum_{t=1}^{T} L_t$
- What is the optimal solution of L?
- What is the gradient $\nabla_\theta L$ and the truncated gradient $\tilde{\nabla}_\theta L_t$?
- (Exercise) What is the BPTT and Truncated BPTT algorithm for this problem.
- (Exercise) How to choose the learning rate η_k to find an optimal θ?