Reviving and Improving Recurrent Back Propagation

Renjie Liao\textsuperscript{1,2,3}, Yuwen Xiong\textsuperscript{1}, Ethan Fetaya\textsuperscript{1,3}, Lisa Zhang\textsuperscript{1,3}, KiJung Yoon\textsuperscript{4,5}, Xaq Pitkow\textsuperscript{4,5}, Raquel Urtasun\textsuperscript{1,2,3}, Richard Zemel\textsuperscript{1,3,6}

\textsuperscript{1}University of Toronto, \textsuperscript{2}Uber ATG Toronto, \textsuperscript{3}Vector Institute, \textsuperscript{4}Rice University, \textsuperscript{5}Baylor College of Medicine, \textsuperscript{6}Canadian Institute for Advanced Research

rjliao@cs.toronto.edu

July 12, 2018
Overview

1 Motivations & Backgrounds

2 Algorithms

3 Experiments
Motivations

Recurrent Back-Propagation (RBP), a.k.a., Almeida-Pineda algorithm, is independently proposed by following papers:


Recurrent Back-Propagation (RBP), a.k.a., Almeida-Pineda algorithm, is independently proposed by following papers:


**Property**

- It exploits implicit function theorem to compute the gradient without back-propagating through time
Motivations

Recurrent Back-Propagation (RBP), a.k.a., Almeida-Pineda algorithm, is independently proposed by following papers:


Property

- It exploits implicit function theorem to compute the gradient without back-propagating through time
- It is efficient with memory and computation
Recurrent Back-Propagation (RBP), a.k.a., Almeida-Pineda algorithm, is independently proposed by following papers:


**Property**

- It exploits implicit function theorem to compute the gradient without back-propagating through time
- It is efficient with memory and computation
- It is successful for limited cases, e.g., Hopfield Networks
Recurrent Back-Propagation (RBP), a.k.a., Almeida-Pineda algorithm, is independently proposed by following papers:


### Property

- It exploits implicit function theorem to compute the gradient without back-propagating through time
- It is efficient with memory and computation
- It is successful for limited cases, e.g., Hopfield Networks

Similar technique (implicit differentiation) was rediscovered later in PGMs!
**Convergent Recurrent Neural Networks**

- **Dynamics:**
  \[ h^{t+1} = F(x, w, h^t) \]

  where \( x, w \) and \( h^t \) are data, weight and hidden state.

- **Steady/Stationary/Equilibrium State:**
  \[ h^* = F(x, w, h^*) \]

**Special Instances**

- Jacobi method
- Gauss–Seidel method
- Fixed-point iteration method
Forward Pass:

\[
\begin{align*}
&h^0 \rightarrow F \rightarrow h^1 \\
&h^1 \rightarrow F \rightarrow h^2 \\
&h^2 \rightarrow F \rightarrow h^{T-1} \\
&h^{T-1} \rightarrow F \rightarrow h^T \\
&h^T \rightarrow L
\end{align*}
\]
Back-Propagation Through Time

Forward Pass:

\[ h^0 \xrightarrow{F} h^1 \xrightarrow{F} h^2 \xrightarrow{F} h^{T-1} \xrightarrow{F} h^T \xrightarrow{L} \]

\[ x \quad x \quad x \quad x \]

\[ w \quad w \quad w \quad w \]

\[ F \quad F \quad F \quad F \]

\[ h^0 \quad h^1 \quad h^2 \quad h^{T-1} \quad h^T \]

Renjie Liao (UofT)
Back-Propagation Through Time

Forward Pass:

\[ h^0 \xrightarrow{F} h^1 \xrightarrow{F} h^2 \xrightarrow{F} h^{T-1} \xrightarrow{F} h^T \xrightarrow{L} \]

\[ x \xrightarrow{w} W \]

\[ x \xrightarrow{w} W \]

\[ x \xrightarrow{w} W \]

\[ x \xrightarrow{w} W \]
Back-Propagation Through Time

Forward Pass:

\[
\begin{align*}
  h^0 & \xrightarrow{F} h^1 \\
  x & \xrightarrow{w} h^0 \\
  h^1 & \xrightarrow{F} h^2 \\
  x & \xrightarrow{w} h^1 \\
  h^2 & \xrightarrow{F} h^{T-1} \\
  x & \xrightarrow{w} h^2 \\
  h^{T-1} & \xrightarrow{F} h^T \\
  x & \xrightarrow{w} h^{T-1} \\
  h^T & \xrightarrow{F} L \\
\end{align*}
\]
Backward Pass:

\[
\begin{align*}
F_{h^0} & \rightarrow h^1 \\
F_{h^1} & \rightarrow h^2 \\
F_{h^2} & \rightarrow h^{T-1} \\
F_{h^{T-1}} & \rightarrow h^T \\
F_{h^T} & \rightarrow L
\end{align*}
\]
Back-Propagation Through Time

Backward Pass:
Back-Propagation Through Time

Backward Pass:
Back-Propagation Through Time

Backward Pass:

\[
\begin{align*}
&h_0 \xrightarrow{F} h^1 \xrightarrow{F} h^2 \xrightarrow{F} \ldots \xrightarrow{F} h^T-1 \xrightarrow{F} h^T \xrightarrow{F} L \\
&w \quad w \quad w \quad w \quad w \quad w
\end{align*}
\]
Back-Propagation Through Time

Backward Pass:

\[
\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h^T} \left( \frac{\partial h^T}{\partial w} + \frac{\partial h^T}{\partial h^{T-1}} \frac{\partial h^{T-1}}{\partial w} + \ldots \right)
\]

\[
= \frac{\partial L}{\partial h^T} \sum_{k=1}^{T} \left( \prod_{i=T-k+1}^{T-1} J_{F,h^i} \right) \frac{\partial F(x, w, h^{T-k})}{\partial w}
\]
Truncated Back-Propagation Through Time

Backward Pass:

Truncated at $K$-th step:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h^T} \left( \frac{\partial h^T}{\partial w} + \frac{\partial h^T}{\partial h^{T-1}} \frac{\partial h^{T-1}}{\partial w} + \ldots \right)$$

$$= \frac{\partial L}{\partial h^T} \sum_{k=1}^{K} \left( \prod_{i=T-k+1}^{T-1} J_{F,h^i} \right) \frac{\partial F(x, w, h^{T-k})}{\partial w}$$
Implicit Function Theorem

Let \( \Psi(x, w, h) = h - F(x, w, h) \), at steady state \( h^* \), we have
\[
\Psi(x, w, h^*) = 0
\]

Implicit Function Theorem is applicable if two conditions hold:

- \( \Psi \) is continuously differentiable
- \( I - J_{F,h^*} \) is invertible
Recurrent Back-Propagation

Implicit Function Theorem

Let \( \Psi(x, w, h) = h - F(x, w, h) \), at steady state \( h^* \), we have
\( \Psi(x, w, h^*) = 0 \)

Implicit Function Theorem is applicable if two conditions hold:
- \( \Psi \) is continuously differentiable (LSTM, GRU)
- \( I - J_{F,h^*} \) is invertible
Contraction Mapping on Banach Space

$F$ is a contraction mapping on Banach (completed and normed) space $B$, iff there exists some $0 \leq \mu < 1$ such that $\forall x, y \in B$

$$
\|F(x) - F(y)\| \leq \mu \|x - y\|
$$

One Sufficient Condition

- Contraction Mapping $\implies \sigma_{\text{max}}(J_F, h^*) \leq \mu < 1$
- We then have,

$$
|\det(I - J_F, h^*)| = \prod_{i} |\sigma_i(I - J_F, h^*)| \\
\geq [1 - \sigma_{\text{max}}(J_F, h^*)]^d > 0
$$

- $I - J_F, h^*$ is invertible
### Implicit Function Theorem

\[
\frac{\partial \psi(x, w, h^*)}{\partial w} = \frac{\partial h^*}{\partial w} - \nabla F(x, w, h^*) \nabla w = (I - J_{F,h^*}) \frac{\partial h^*}{\partial w} - \frac{\partial F(x, w, h^*)}{\partial w} = 0 \tag{1}
\]

The desired gradient is:

\[
\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h^*} (I - J_{F,h^*})^{-1} \frac{\partial F(x, w, h^*)}{\partial w}
\]
Derivation of Original RBP

- Gradient:

\[ \frac{\partial L}{\partial w} = \frac{\partial L}{\partial h^*} (I - J_{F,h^*})^{-1} \frac{\partial F(x, w, h^*)}{\partial w} \]

- Introduce \( z^\top = \frac{\partial L}{\partial h^*} (I - J_{F,h^*})^{-1} \) which defines an adjoint linear system,

\[ (I - J_{F,h^*}^\top) z = \left( \frac{\partial L}{\partial h^*} \right)^\top \]  \hspace{1cm} (2)

- Original RBP uses fixed-point iteration method,

\[ z = J_{F,h^*}^\top z + \left( \frac{\partial L}{\partial h^*} \right)^\top = f(z) \]  \hspace{1cm} (3)
### Algorithm 1: Original RBP

1. **Initialization**: initial guess $z_0$, e.g., draw uniformly from $[0, 1]$, $i = 0$, threshold $\epsilon$
2. repeat
3. $i = i + 1$
4. $z_i = J^T_{F,h^*} z_{i-1} + \left( \frac{\partial L}{\partial h^*} \right)^T$
5. until $\|z_i - z_{i-1}\| < \epsilon$
6. Return $\frac{\partial L}{\partial w} = z_i^T \frac{\partial F(x,w,h^*)}{\partial w}$

### Pros & Cons

- Memory cost scales constantly w.r.t. \# time steps whereas BPTT scales linearly
- It is often faster than BPTT for many-step RNNs
- It may converge slowly and sometimes numerically unstable
Our core problem is \((I - J_F^T, h^*) z = \left( \frac{\partial L}{\partial h^*} \right)^\top\), simplified as \(Az = b\)

- CG is better than fixed-point iteration if \(A\) is PSD
  
- \(A\) is often asymmetric for RNNs

**Conjugate Gradient on the Normal Equations (CGNE)**

- Multiple \((I - J_F, h^*)\) on both sides,

\[
(I - J_F, h^*) \left( I - J_F^T, h^* \right) z = (I - J_F, h^*) \left( \frac{\partial L}{\partial h^*} \right)^\top
\]

- Apply CG to solve \(z\)

**Caveat:** the condition number of the new system is squared!
Neumann Series based RBP

Neumann Series
- It's a mathematical series of the form $\sum_{t=0}^{\infty} A^t$ where $A$ is an operator, a.k.a., matrix geometric series in matrix terminology.
- A convergent Neumann series has the property:
  \[
  (I - A)^{-1} = \sum_{k=0}^{\infty} A^k
  \]

Neumann-RBP
- Recall auxiliary variable $z$ in RBP:
  \[
  z = \left( I - J_{F,h^*}^\top \right)^{-1} \left( \frac{\partial L}{\partial h^*} \right)^\top
  \]
- Replace $A$ with $J_{F,h^*}^\top$ and truncate it at $K$-th power.
Algorithm 2: Neumann-RBP

1: Initialization: \( v_0 = g_0 = \left( \frac{\partial L}{\partial h^*} \right)^\top \)
2: for \( t = 1, 2, \ldots, K \) do
3: \( v_t = J_{F,h^*}^\top v_{t-1} \)
4: \( g_t = g_{t-1} + v_t \)
5: end for
6: Return \( \frac{\partial L}{\partial w} = (g_K)^\top \frac{\partial F(x,w,h^*)}{\partial w} \)

We show Neumann-RBP is related to BPTT and TBPTT:

**Proposition 1**

Assume that we have a convergent RNN which satisfies the implicit function theorem conditions. If the Neumann series \( \sum_{t=0}^{\infty} J_{F,h^*}^t \) converges, then the full Neumann-RBP is equivalent to BPTT.

**Proposition 2**

For the above RNN, let us denote its convergent sequence of hidden states as \( h^0, h^1, \ldots, h^T \) where \( h^* = h^T \) is the steady state. If we further assume that there exists some step \( K \) where \( 0 < K \leq T \) such that \( h^* = h^T = h^{T-1} = \cdots = h^{T-K} \), then \( K \)-step Neumann-RBP is equivalent to \( K \)-step TBPTT.
Proposition 3

If the Neumann series $\sum_{t=0}^{\infty} J_{F,h^*}^t$ converges, then the error between $K$-step and full Neumann series is as follows,

$$\left\| \sum_{t=0}^{K} J_{F,h^*}^t - \sum_{t=0}^{\infty} J_{F,h^*}^t \right\| \leq \| (I - J_{F,h^*})^{-1} \| \| J_{F,h^*} \|^K + 1$$

Pros & Cons

- CG-RBP requires fewer # updates but may be slower in run time and is sometimes problematic due to the squared condition number
- Neumann-RBP is stable and has same time & memory complexity
Continuous Hopfield Networks

Model

- **Inference:**

\[
\frac{d}{dt} h_i(t) = - \frac{h_i(t)}{a} + \sum_{j=1}^{N} w_{ij} \phi(b \cdot h_j(t)) + l_i,
\]

- **Learning:**

\[
\min_w \frac{1}{|I|} \sum_{i \in I} \| \phi(b \cdot h_i) - l_i \|_1
\]
Continuous Hopfield Networks

Figure: Visualization of associative memory. (a) Corrupted input image; (b)-(f) are retrieved images by BPTT, TBPTT, RBP, CG-RBP, Neumann-RBP respectively.

Table: Success (final loss \(\leq 50\%\) initial loss) rate.

<table>
<thead>
<tr>
<th>Truncation Step</th>
<th>10</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBPTT</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>RBP</td>
<td>1%</td>
<td>4%</td>
<td>99%</td>
</tr>
<tr>
<td>CG-RBP</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Neumann-RBP</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Gated Graph Neural Networks

Input → Hidden Layer → GRU → Hidden Layer → GRU → Output

Renjie Liao (UofT)
Gated Graph Neural Networks

- Semi-supervised document classification in citation networks

Figure: (a) Training loss; (b) Validation accuracy. (c) Difference between consecutive hidden states.
Hyperparameter Optimization

- lr, wd, etc.

MLP + SGD

$w_1$

$w_2$

$w_T$

$W_0$

$W_1$

$W_2$

$W_T$

$L$

data

MLP + SGD

Renjie Liao (UofT)
Optimize 100 steps, truncate 50 steps:

Figure: (a) Training loss at last meta step; (b) Meta training loss.
Optimize 1500 steps, truncate 50 steps:

![Graph showing training loss at last meta step](a)

**Figure:** (a) Training loss at last meta step; (b) Meta training loss.
Hyperparameter Optimization

(a) Meta step 20
(b) Meta step 40.

<table>
<thead>
<tr>
<th>Truncation Step</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>Run Time</th>
<th>10</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\times 3.02)</td>
<td>(\times 2.87)</td>
<td>(\times 2.68)</td>
<td>Memory</td>
<td>(\times 4.35)</td>
<td>(\times 4.25)</td>
<td>(\times 4.11)</td>
</tr>
</tbody>
</table>

**Table:** Run time and memory comparison. We show the ratio of BPTT’s cost divided by Neumann-RBP’s.
def neumann_rbp(weight, hidden_state, loss, rbp_step):
    # get the gradient of last hidden state
    grad_h = autograd.grad(loss, hidden_state[-1], retain_graph=True)

    # set v, g to grad_h
    neumann_v = grad_h.clone()
    neumann_g = grad_h.clone()

    for i in range(rbp_step):
        # set last hidden_state’s gradient to neumann_v[prev]
        # and get the gradient of last second hidden state
        neumann_v = autograd.grad(
            hidden_state[-1], hidden_state[-2],
            grad_outputs=neumann_v,
            retain_graph=True)

        neumann_g += neumann_v

    # set last hidden_state’s gradient to neumann_g
    # and return the gradient of weight
    return autograd.grad(hidden_state[-1], weight, grad_outputs=neumann_g)
RBP is very efficient for learning convergent RNNs
Neumann-RBP is stable, often faster than BPTT and takes constant memory
Neumann-RBP is simple to implement

Welcome to our poster #178 tonight!
Thank You