Paper Review of Antisymmetric RNN

Bo Chang, Minmin Chen, Eldad Haber, Ed H. Chi

University of British Columbia, Google Brain

January 3, 2020

Table of Contents

Recurrent Neural Networks

- Recurrent Neural Networks(RNN) have found widespread use across a variety of domains from language modeling, machine translation to speech recognition, recommendation systems and time series prediction.
- A common misunderstanding is that RNN has been completely replaced by transformer-based models. This is correct for most language modeling tasks, but for many other tasks it's still SOTA.

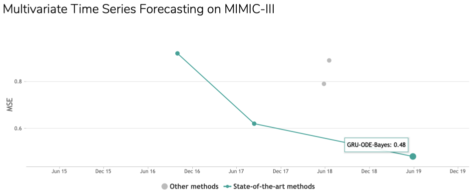


Figure: SOTA for Time Series Forecasting

Trainability of RNN

RNN faces two main drawbacks:

- Hard to parallelize
- Vanishing/Exploding Gradient Problem(this paper)

A lot of Variants are proposed to solve the problem:

- LSTM
- GRU
- ...

But lack math theory.

Vanishing/Exploding Gradient Problem

- RNN model: Input x_t from $t = 0$ to $t = t_0$.

$$h_t = \tanh(W_1 h_{t-1} + W_2 x_{t-1} + b)$$

Training: minimize $\mathcal{E} := \sum_{t=0}^{t_0} L_t = \sum_{t=0}^{t_0} L(h_t)$ to get W_1, W_2 and b.

- $$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{t=0}^{t_0} \frac{\partial L_t}{\partial \theta}, \frac{\partial L_t}{\partial \theta} = \sum_{k=0}^{t} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial \theta}$$

,

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = W_1^T diag(\tanh'(h_{i-1}))$$

Table of Contents

Dynamical System View of RNN

They add a residual connection to change original formula $h_t = \tanh(W_1 h_{t-1} + W_2 x_{t-1} + b)$ to

$$h_t = h_{t-1} + \epsilon \tanh(W_1 h_{t-1} + W_2 x_{t-1} + b) \tag{1}$$

It can be seen this is the forward Euler discretization of

$$h'(t) = \tanh(W_1 h(t) + W_2 x(t) + b) \tag{2}$$

(2) is the continuous analogue of (1). To study the stability of (1), it is good to first study the stability of (2).

Stability of ODE

We give the definition and criterion for the stability of $h'(t) = f(h(t))$, which is a general form of (2).

Definition

A solution $h(t)$ of the ODE $h'(t) = f(h(t))$ with initial condition $h(0)$ is stable if for any $\epsilon > 0$, there exists a $\delta > 0$ such that any other solution $\tilde{h}(t)$ of the ODE with initial condition $\tilde{h}(0)$ satisfying $|h(0) - \tilde{h}(0)| \leq \delta$ also satisfies $|h(t) - \tilde{h}(t)| \leq \epsilon$ for all $t \geq 0$.

Theorem

The solution of an ODE is stable if

$$\max_{i=1,2,\ldots,n} Re(\lambda_i(J(t))) \leq 0, \forall t \geq 0, \tag{3}$$

where $J(t)$ is the Jacobian matrix of f.

Stability and Trainability

Theorem

$$\frac{d}{dt}\left(\frac{\partial h(t)}{\partial h(0)}\right) = J(t)\frac{\partial h(t)}{\partial h(0)} \tag{4}$$

For notational simplicity, define $A(t) = \frac{\partial h(t)}{\partial h(0)}$, then we have

$$\frac{dA(t)}{dt} = J(t)A(t), \quad A(0) = I \tag{5}$$

This is a linear ODE with solution $A(t) = e^{J \cdot t} = Pe^{\Lambda(J)t}P^{-1}$, assuming the Jacobian J does not vary or vary slowly over time.
When $Re(\Lambda(J)) \approx 0$, the magnitude of $A(t)$ is approximately constant in time, thus no exploding or vanishing gradient problems.

Table of Contents

RNN Revisited

- Back to the RNN model, where $f(t) = \tanh(W_1 h(t) + W_2 x(t) + b)$, then $J(t) = diag[\tanh'(W_1 h(t) + W_2 x(t) + b)]W_1$.
- If the eigenvalues of W_1 are all imaginary, then the eigenvalue of J are all imaginary, which is what we want.
- Antisymmetric matrices have imaginary eigenvalues!
- Solution: Let $W_1 = W - W^T$
- Proposed Scheme:

$$h_t = h_{t-1} + \epsilon \tanh((W - W^T)h_{t-1} + W_2 x_t + b) \tag{6}$$

Diffusion is All You Need

However, a problem is encountered:

Theorem

The forward propagation in Equation (6) is stable if

$$\max_{i=1,2,\ldots,n} |1 + \epsilon \lambda_i(J_t)| \leq 1 \tag{7}$$

Since $\lambda_i(J_t)$ is imaginary, the scheme we proposed is always unstable. A diffusion term is added to rescue, and this gives the final form of Antisymmetric RNN:

$$h_t = h_{t-1} + \epsilon \tanh((W - W^T - \gamma I)h_{t-1} + W_2 x_{t-1} + b), \tag{8}$$

where $\gamma > 0$ is a hyperparameter that controls the strength of diffusion.

Gating Mechanism

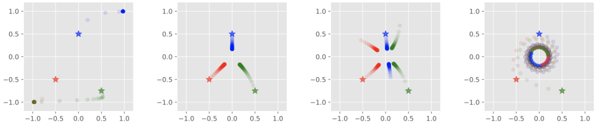
A variation of above scheme is also proposed:

$$
\begin{aligned}
z_t &= \sigma((W - W^T - \gamma I)h_{t-1} + W_z x_t + b_z), \\
h_t &= h_{t-1} + \epsilon z_t \circ \tanh((W - W^T - \gamma I)h_{t-1} + W_h x_t + b_h)
\end{aligned}
\tag{9}
$$

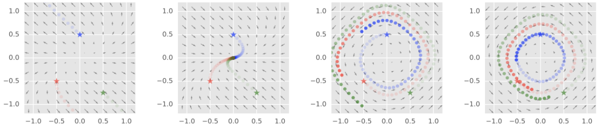
Gating is commonly employed in RNNs. Each gate is often modeled as a single layer network taking the previous hidden state h_{t-1} and data x_t as inputs, followed by a sigmoid activation.

Table of Contents

4 SIMULATION



(a) Vanilla RNN with a random weight matrix. (b) Vanilla RNN with an identity weight matrix. (c) Vanilla RNN with a random orthogonal weight matrix (seed = 0). (d) Vanilla RNN with a random orthogonal weight matrix (seed = 1).



(e) RNN with feedback with positive eigenvalues. (f) RNN with feedback with negative eigenvalues. (g) RNN with feedback with imaginary eigenvalues. (h) RNN with feedback with imaginary eigenvalues and diffusion.

Figure 1: Visualization of the dynamics of RNNs and RNNs with feedback using different weight matrices.

Figure: Dynamics of a Toy 2D System

Pixel by Pixel MNIST

MNIST images are grayscale with 28×28 pixels. The 784 pixels are presented sequentially to the recurrent net, one pixel at a time in scanline order (starting at the top left corner of the image and ending at the bottom right corner). In other words, the input dimension $m = 1$ and number of time steps $T = 784$. The pixel-by-pixel MNIST task is to predict the digit of the MNIST image after seeing all 784 pixels.

method	MNIST	pMNIST	# units	# params
LSTM (Arjovsky et al., 2016)[1]	97.3%	92.6%	128	68k
FC uRNN (Wisdom et al., 2016)	92.8%	92.1%	116	16k
FC uRNN (Wisdom et al., 2016)	96.9%	94.1%	512	270k
Soft orthogonal (Vorontsov et al., 2017)	94.1%	91.4%	128	18k
KRU (Jose et al., 2017)	96.4%	94.5%	512	11k
AntisymmetricRNN	98.0%	**95.8%**	128	10k
AntisymmetricRNN w/ gating	**98.8%**	93.1%	128	10k

Table 1: Evaluation accuracy on pixel-by-pixel MNIST and permuted MNIST.

Figure: Prediction Accuracy on Pixel by Pixel MNIST

Pixel by Pixel CIFAR-10

method	pixel-by-pixel	noise padded	# units	# params
LSTM	59.7%	11.6%	128	69k
Ablation model	54.6%	46.2%	196	42k
AntisymmetricRNN	58.7%	48.3%	256	36k
AntisymmetricRNN w/ gating	**62.2%**	**54.7%**	256	37k

Table 2: Evaluation accuracy on pixel-by-pixel CIFAR-10 and noise padded CIFAR-10.

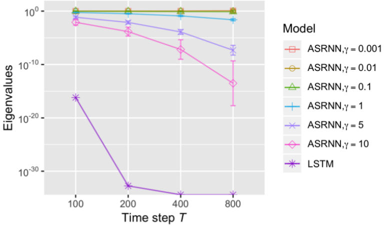


Figure: Eigenvalues of the Jacobian matrix in different models, trained on the noise padded CIFAR10

Experimental Details

Let m be the input dimension and n be the number of hidden units. The input to hidden matrices are initialized to $\mathcal{N}(0, 1/m)$. The hidden to hidden matrices are initialized to $\mathcal{N}(0, \sigma_w^2/n)$, where σ_w is chosen from $\sigma_w \in \{0, 1, 2, 4, 8, 16\}$. The bias terms are initialized to zero, except the forget gate bias of LSTM is initialized to 1, as suggested by Jozefowicz et al. (2015). For AntisymmetricRNNs, the step size $\epsilon \in \{0.01, 0.1, 1\}$ and diffusion $\gamma \in \{0.001, 0.01, 0.1, 1.0\}$. We use SGD with momentum and Adagrad (Duchi et al., 2011) as optimizers, with batch size of 128 and learning rate chosen from $\{0.1, 0.2, 0.3, 0.4, 0.5, 0.75, 1\}$. On MNIST and pixel-by-pixel CIFAR-10, all the models are trained for 50,000 iterations. On noise padded CIFAR-10, models are trained for 10,000 iterations. We use the standard train/test split of MNIST and CIFAR-10. The performance measure is the classification accuracy evaluated on the test set.

Table of Contents

Why Pytorch

- RNNs could be really slow if we use standard Tensorflow/PyTorch operators, because overhead is created: most Tensorflow/PyTorch operations launch at least one kernel on the GPU and RNNs generally run many operations due to their recurrent nature
- Both Tensorflow and Pytorch support CUDNNLSTM layers, which uses a fused kernel. It increases the speed of computation a lot, but it is difficult modify the base implementation(change the architecture).
- We can apply TorchScript in Pytorch to fuse operations and optimize our code automatically, launching fewer, more optimized kernels on the GPU.

Custom RNN

```
class ASNNCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size, sigma):
        super(ASNNCell, self).__init__()
        self.weight_ih = nn.Parameter(torch.randn(hidden_size,
                                                  input_size)/input_size)
        self.weight_hh = nn.Parameter(torch.randn(hidden_size, hidden_size)\
                                      *sigma*sigma/hidden_size)
        self.bias = nn.Parameter(torch.zeros(hidden_size))

    @jit.script_method
    def forward(self, inputs, hx, gammai):

        hy = hx + 0.01 * torch.tanh(torch.mm(inputs, self.weight_ih.t()) +
                 torch.mm(hx, (self.weight_hh.t()-self.weight_hh - gammai))
                 + self.bias)

        return hy

class ASNNLayer(jit.ScriptModule):
    def __init__(self, cell, *cell_args):
        super(ASNNLayer, self).__init__()
        self.cell = cell(*cell_args)

    @jit.script_method
    def forward(self, inputs, state, gammai):
        inputs = inputs.unbind(0)
        outputs = torch.jit.annotate(List[Tensor], [])
        for i in range(len(inputs)):
            state = self.cell(inputs[i], state, gammai)
            outputs += [state]
        return torch.stack(outputs),[state]
```

Figure: Code for Antisymmetric RNN

Speed Comparison

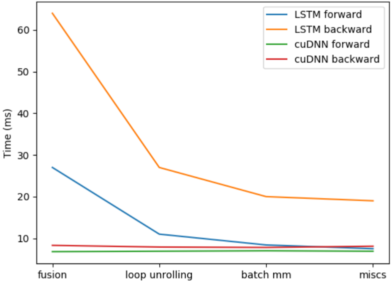


Figure: Comparison of LSTM with and without CuDNN Acceleration

Table: Time to train a single epoch on MNIST (second)

PyTorch(+)	PyTorch	TF(+)	TF
23.10	71.86	49.10	260.33

Table of Contents

- A new perspective on the trainability of RNNs from dynamical system point of view is given.
- Antisymmetric RNN is proposed based on discretization of ODEs that satisfy the critical criterion.
- The models proposed have demonstrated competitive performance over strong recurrent baselines on a set of benchmark tasks.

The End