Recurrent Neural Networks

STATS305B: Applied Statistics II

Scott Linderman

March 3, 2025

Last Time...

- Transformers
- Attention
- Some tricks

Today...

Outline:

- Recurrent Neural Networks
- Backpropagation Through Time
- Vanishing Gradients and Gated RNNs
- Other Variations and Uses of RNNs
- Revisiting HMMs
- Linear RNNs and Parallel Inference

Autoregressive Models

Consider a sequence of observations $\boldsymbol{x}_{1:T} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ with each $\boldsymbol{x}_t \in \mathbb{R}^D$. We can always factor a joint distribution over observation into a product of conditionals using the chain rule,

$$p(\boldsymbol{x}_{1:T}) = p(\boldsymbol{x}_1) \prod_{t=2}^{T} p(\boldsymbol{x}_t \mid \boldsymbol{x}_{1:t-1}).$$

This is called an **autoregressive model** since the conditional of $\boldsymbol{x}_t$ depends only on previous observations $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{t-1}$.

Autoregressive models are well-suited to sequential modeling since they make forward generation or *forecasting* easy. As long as we have access to the conditionals, we can sample forward indefinitely.

The question is, how should we parameterize these conditional distributions? It looks challenging since each one takes in a variable-length history of previous observations.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are autoregressive models in which the conditional distributions are functions of a finite-dimensional **hidden state**, $\boldsymbol{h}_t \in \mathbb{R}^K$,

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{1:t-1}) = p(\boldsymbol{x}_t \mid g(\boldsymbol{h}_t; \boldsymbol{\theta})).$$

The hidden state is updated with each new observation as,

$$\boldsymbol{h}_{t+1} = f(\boldsymbol{h}_t, \boldsymbol{x}_t; \boldsymbol{\theta}).$$

Defining Property of an RNN *The conditional distribution over the next observation is a function of hidden state. The hidden state is updated recursively, and its size is fixed regardless of the sequence length.*

Vanilla RNNs

The standard, “vanilla” RNN consists of a linear-nonlinear state update. For example,

$$f(\boldsymbol{h}_t, \boldsymbol{x}_t; \boldsymbol{\theta}) = \tanh\left(\boldsymbol{W}\boldsymbol{h}_t + \boldsymbol{B}\boldsymbol{x}_t\right),$$

where - $\boldsymbol{W} \in \mathbb{R}^{K \times K}$ are the dynamics weights, - $\boldsymbol{B} \in \mathbb{R}^{K \times D}$ are the input weights, - $\tanh(\cdot)$ is the hyperbolic tangent function.

Hyperbolic Tangent and the Logistic Function The hyperbolic tangent function equivalent is typically written as,

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}.$$

Vanilla RNNs

We can rewrite tanh as,

$$\begin{aligned}\tanh(a) &= 1 - \frac{2e^{-a}}{e^{a} + e^{-a}} \\ &= 1 - 2\sigma(-2a) \\ &= 1 - 2(1 - \sigma(2a)) \\ &= 2\sigma(2a) - 1.\end{aligned}$$

Thus, we see that the the hyperbolic tangent is simply a scaled and shifted logistic function.

The “read-out” of a vanilla RNN is typically a simple linear or generalized linear model, depending on the type of observations. For example,

$$g(\boldsymbol{h}_t, \boldsymbol{x}_t; \boldsymbol{\theta}) = \boldsymbol{C}\boldsymbol{h}_t + \boldsymbol{d},$$

where $\boldsymbol{C} \in \mathbb{R}^{D \times K}$ is the read-out weight matrix and $\boldsymbol{d} \in \mathbb{R}^{D}$ is the bias.

Let $\boldsymbol{\theta} = (\boldsymbol{W}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{d})$ denote the set of parameters.

Theoretical Neuroscience

In machine learning, RNNs are useful function approximators for sequential data. In neuroscience, they have a long history as theoretical models of biological computation.

The hidden state $\boldsymbol{h}_t \in \mathbb{R}^K$ corresponds to the relative **firing rates** of K neurons. With a tanh nonlinearity, $\boldsymbol{h}_t \in (-1, 1)^K$, and negative rates don't make sense. Instead, we think of $\boldsymbol{h}_t$ as an offset to a baseline firing rate.

The dynamics weights correspond to **synaptic connections**, with positive weights as excitatory synapses and negative weights as inhibitory. When a presynaptic neuron spikes, it induces an electrical current in postsynaptic neurons. The activation $\boldsymbol{W}\boldsymbol{h}_t$ is the **input current** from other neurons.

As a neuron receives input current, its voltage steadily increases until it reaches a **threshold**, at which point the voltage spikes and the neuron fires an **action potential**. These spikes induce currents on downstream neurons, as described above.

After a cell fires, there is a short **refractory period** before the neuron can spike again. Thus, there is an upper bound on firing rates, which the hyperbolic tangent is meant to capture.

Backpropagation Through Time

Artificial RNNs are trained using stochastic gradient descent (SGD) to minimize the negative log likelihood,

$$\begin{aligned}
\mathscr{L}(\boldsymbol{\theta}) &= -\sum_{t=1}^{T} \log p(\boldsymbol{x}_t \mid \boldsymbol{x}_{1:t-1}) \\
&= -\sum_{t=1}^{T} \log p(\boldsymbol{x}_t \mid g(\boldsymbol{h}_t; \boldsymbol{\theta})) \\
&= -\sum_{t=1}^{T} \log p(\boldsymbol{x}_t \mid g(f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_{t-1}; \boldsymbol{\theta}); \boldsymbol{\theta})) \\
&= -\sum_{t=1}^{T} \log p(\boldsymbol{x}_t \mid g(f(\cdots f(\boldsymbol{h}_1, \boldsymbol{x}_1; \boldsymbol{\theta}) \cdots, \boldsymbol{x}_{t-1}; \boldsymbol{\theta}); \boldsymbol{\theta})).
\end{aligned}$$

Backpropagation Through Time

Now, you would simply use automatic differentiation to compute the necessary gradients to minimize this loss, but we can gain some insight by working them out manually.

With some algebra, we can show that the Jacobian of the loss with respect to the dynamics weights (other parameters are similar) is,

$$\frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \boldsymbol{W}} = \sum_{t=1}^{T} \frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}}$$

We need the Jacobian of the loss with respect to the hidden states. These can be computed recursively by **backpropagation through time (BPTT)**,

$$\frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \boldsymbol{h}_t} = \frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t} - \frac{\partial \log p(\boldsymbol{x}_t \mid g(\boldsymbol{h}_t; \boldsymbol{\theta}))}{\partial \boldsymbol{h}_t}$$

Backpropagation Through Time

For a vanilla RNN, the Jacobian of the next state with respect to the current state is,

$$\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t} = \mathrm{diag}\left(1 - \boldsymbol{h}_{t+1}^2\right) \boldsymbol{W}$$

since $\frac{\mathrm{d}}{\mathrm{d}a} \tanh(a) = 1 - \tanh(a)^2$.

BPTT is a linear dynamical system

The "state" of the BPTT recursions is the Jacobian, or equivalently its transpose, the gradient $\boldsymbol{s}_t \triangleq \left(\frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \boldsymbol{h}_t}\right)^\top$. This state obeys a linear dynamical system,

$$\boldsymbol{s}_t = \boldsymbol{A}_t \boldsymbol{s}_{t+1} + \boldsymbol{b}_t$$

where $\boldsymbol{A}_t = \left(\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}\right)^\top$ and $\boldsymbol{b}_t = -\left(\frac{\partial \log p(\boldsymbol{x}_t \mid g(\boldsymbol{h}_t; \boldsymbol{\theta}))}{\partial \boldsymbol{h}_t}\right)^\top$.

Biological Plausibility

Backpropagation is the most effective way we know of to train artificial RNNs, so it's reasonable to think that the brain might be using a similar learning algorithm.

Unfortunately, it's not clear how the backpropagation through time algorithm could be implemented by a neural circuit.

The multiplication by $\boldsymbol{A}_t$ in the gradient recursions amounts to passing information backward across synapses, and canonical synaptic models don't have mechanisms to do this.

Recent years have seen a substantial amount of research into biologically plausible mechanisms of backpropagation.

Vanishing Gradients

When the Jacobians $\boldsymbol{A}_t$ have small eigenvalues ($\ll 1$), we run into problems of **vanishing gradients**.

Consider the case of a linear RNN in which the tanh is replaced with identity: then the Jacobians are $\boldsymbol{A}_t = \boldsymbol{W}^\top$ for all time steps.

If the eigenvalues of $\boldsymbol{W}$ are much less than one, the gradients will decay to zero exponentially quickly, absent strong inputs $\boldsymbol{b}_t$.

Vanishing gradients are especially problematic when $\boldsymbol{x}_t$ depends on much earlier observations, $\boldsymbol{x}_s$ for $s \ll t$. In that case, the hidden state must propagate information about $\boldsymbol{x}_s$ for many time steps during the forward pass, and likewise, the gradient must pass information backward many timesteps during the backward pass.

If the weights have small eigenvalues, those gradients will decay and the learning signal will fail to propagate.

Gated RNNs

One way to combat the vanishing gradient problem is by modifying the RNN architecture. Architectures like **long short-term memory (LSTM)** networks achieve this via gated units.

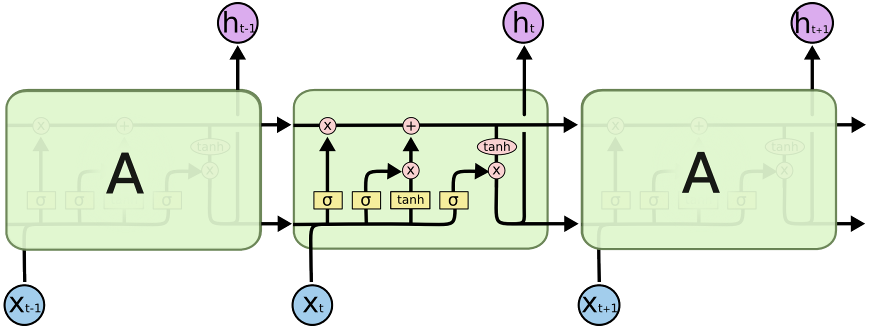

Figure: An LSTM cell. Figure from `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

Gated RNNs

An LSTM has internal (aka "cell") states $\boldsymbol{c}_t \in \mathbb{R}^K$ and hidden states $\boldsymbol{h}_t \in \mathbb{R}^K$. The internal states follow **conditionally linear** dynamics,

$$\boldsymbol{c}_t = \boldsymbol{F}_t \boldsymbol{c}_{t-1} + \boldsymbol{b}_t$$

where

$$\begin{aligned} \boldsymbol{F}_t &= \mathrm{diag}(f_{t,1}, \ldots, f_{t,K}) \\ f_{t,k} &= \sigma(\boldsymbol{W}^{(f)} \boldsymbol{h}_{t-1} + \boldsymbol{B}^{(f)} \boldsymbol{x}_{t-1}). \end{aligned}$$

The bounded entries $f_{t,k} \in [0,1]$ ensure stability. When $f_{t,k} \approx 1$, the state is propagated, and when $f_{t,k} \approx 0$, the state is forgotten.

Thus, $\boldsymbol{f}_t = (f_{t,1}, \ldots, f_{t,K})^\top \in [0,1]^K$ are called the **forget gates**, and they are parameterized by the matrices $\boldsymbol{W}^{(f)}$ and $\boldsymbol{B}^{(f)}$.

Gated RNNs

The affine term is determined by,

$$
\begin{aligned}
\boldsymbol{b}_t &= \boldsymbol{g}_t \odot \boldsymbol{i}_t \\
\boldsymbol{g}_t &= \sigma(\boldsymbol{W}^{(g)} \boldsymbol{h}_{t-1} + \boldsymbol{B}^{(g)} \boldsymbol{x}_{t-1}) \\
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}^{(i)} \boldsymbol{h}_{t-1} + \boldsymbol{B}^{(i)} \boldsymbol{x}_{t-1})
\end{aligned}
$$

The vector $\boldsymbol{g}_t \in [0,1]^K$ plays the role of an **input gate**, and the input s themselves are given by $\boldsymbol{i}_t \in [0,1]^K$.

Finally, the hidden states $\boldsymbol{h}_t$ are gated functions of the internal state passed through a nonlinearity,

$$
\begin{aligned}
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}^{(o)} \boldsymbol{h}_{t-1} + \boldsymbol{B}^{(o)} \boldsymbol{x}_{t-1})
\end{aligned}
$$

Gated RNNs

where $\boldsymbol{o}_t \in [0,1]^K$ are the **output gates**. As in a vanilla RNN, the final prediction depends on a (generalized) linear function of the hidden state, $g(\boldsymbol{h}_t; \boldsymbol{\theta})$.

We can think of an LSTM as an RNN that operates on an extended state $(\boldsymbol{c}_t, \boldsymbol{h}_t) \in \mathbb{R}_+^K \times [-1,1]^K$. The forget gates let the eigenvalues of $\boldsymbol{F}_t$ to be close to one, allowing cell states to be propagated for long periods of time on the forward pass, and gradients to be backpropagated without vanishing on the backward pass.

There are many variants of gated RNNs. Besides the LSTM, the most commonly used in the gated recurrent unit (GRU), which has a slightly simplified architecture. See Goodfellow et al. (2016), Ch. 10,for more detail.

Other Variations and Uses of RNNs

We motivated RNNs from an autoregressive modeling persepective, but they are useful in other sequential data settings as well.

For example, suppose we want to predict the sentiment of a review $y \in \mathbb{R}$ given a variable-length sequence of input words $\boldsymbol{x}_{1:T}$.

We can use an RNN to summarize the input sequence in terms of a hidden state for prediction,

$$p(y \mid \boldsymbol{x}_{1:T}) = p(y \mid g(\boldsymbol{h}_T; \boldsymbol{\theta})).$$

Sequence to Sequence Models

Sometimes we want to map one sequence $\boldsymbol{x}_{1:T}$ to another sequence $\boldsymbol{y}_{1:T'}$. The sequences may be of different length; e.g., when we want to translate a sentence from English to French.

Again, we can train an **encoder** RNN to produce a hidden state $\boldsymbol{h}_T$ that then becomes the initial condition for a **decoder** RNN that generates the output sequence.

Formally,

$$\begin{aligned} p(\boldsymbol{y}_{1:T'} \mid \boldsymbol{x}_{1:T}) &= \prod_{t=1}^{T'} p(\boldsymbol{y}_t \mid \boldsymbol{y}_{1:t-1}, \boldsymbol{x}_{1:T}) \\ &= \prod_{t=1}^{T'} p(\boldsymbol{y}_t \mid \boldsymbol{h}'_t, \boldsymbol{h}_T) \end{aligned}$$

where $\boldsymbol{h}_T$ is the output of an RNN that processed $\boldsymbol{x}_{1:T}$, and $\boldsymbol{h}'_t$ is the state of an RNN that runs over $\boldsymbol{y}_{1:T}$.

Bidirectional RNNs

In the example above, one challenge is that the hidden state $\boldsymbol{h}_T$ obtained by processing $\boldsymbol{x}_{1:T}$ may not adequately represent early inputs like x_1.

For these purposes, you can use a **bidirectional RNN** that runs one recursion forward $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ and another backward $\boldsymbol{x}_T, \ldots, \boldsymbol{x}_1$ to produce two hidden states at each time t.

These combined states can then be passed into the decoder.

Deep RNNs

As with deep neural networks that stack layer upon layer, we can stack RNN upon RNN to construct a deeper model.

In such models, the outputs of one layer, $g(\boldsymbol{h}_t^{(i)}; \boldsymbol{\theta}^{(i)})$ become the inputs to the next layer, $\boldsymbol{x}_t^{(i+1)}$.

Then we can backpropagate gradients through the entire stack to train the model.

HMMs Are RNNs Too!

We presented Hidden Markov Models (HMMs) as latent variable models with hidden states $\boldsymbol{z}_{1:T}$ and observations $\boldsymbol{x}_{1:T}$, but we can also view them as an autoregressive model,

$$\begin{aligned} p(\boldsymbol{x}_t \mid \boldsymbol{x}_{1:t-1}) &= \sum_{k=1}^{K} p(z_t = k \mid \boldsymbol{x}_{1:t-1})\, p(\boldsymbol{x}_t \mid z_t = k) \\ &= \sum_{z_t} \overline{\alpha}_{t,k}\, p(\boldsymbol{x}_t \mid z_t = k) \end{aligned}$$

where, $\overline{\boldsymbol{\alpha}}_t \in \Delta_{K-1}$ are the **normalized forward messages** from the forward-backward algorithm.

They followed a simple recursion,

$$\overline{\boldsymbol{\alpha}}_{t+1} = \boldsymbol{P}^\top \left(\frac{\overline{\boldsymbol{\alpha}}_t \odot \boldsymbol{l}_t}{\overline{\boldsymbol{\alpha}}_t^\top \boldsymbol{l}_t} \right)$$

with $\boldsymbol{l}_t \in \mathbb{R}^K$ is the vector of likelihoods with entries $l_{t,k} = p(\boldsymbol{x}_t \mid z_t = k)$ and $\boldsymbol{P} \in \mathbb{R}^{K \times K}$ is the transition matrix.

Categorical HMMs

Consider an HMM with categorical emissions,

$$p(\boldsymbol{x}_t \mid z_t) = \mathrm{Cat}(\boldsymbol{x}_t \mid \boldsymbol{c}_{z_t}),$$

where $\boldsymbol{x}_t$ is a one-hot encoding of a variable that takes values in $\{1, \ldots, V\}$, and $\boldsymbol{c}_k \in \Delta_{V-1}$ for $k = 1, \ldots, K$ are pmfs.

Define the matrix of likelihoods $\boldsymbol{C} \in \mathbb{R}^{V \times K}$ to have columns $\boldsymbol{c}_k$,

$$\boldsymbol{C} = \begin{bmatrix} | & & | \\ \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_K \\ | & & | \end{bmatrix}.$$

The HMM parameters are $\boldsymbol{\theta} = (\boldsymbol{P}, \boldsymbol{C})$. (Assume the initial distribution is fixed, for simplicity.)

Categorical HMMs

For a categorical HMM, we can write the likelihoods as $\boldsymbol{l}_t = \boldsymbol{C}^\top \boldsymbol{x}_t$ so that the the forward recursions simplify to,

$$\begin{aligned}\overline{\boldsymbol{\alpha}}_{t+1} &= \boldsymbol{P}^\top \left(\frac{\overline{\boldsymbol{\alpha}}_t \odot \boldsymbol{C}^\top \boldsymbol{x}_t}{\overline{\boldsymbol{\alpha}}_t^\top \boldsymbol{C}^\top \boldsymbol{x}_t} \right) \\ &= f(\overline{\boldsymbol{\alpha}}_t, \boldsymbol{x}_t; \boldsymbol{\theta})\end{aligned}$$

Likewise, the autoregressive distributions reduce to,

$$\begin{aligned}p(\boldsymbol{x}_t \mid \boldsymbol{x}_{1:t-1}) &= \mathrm{Cat}(\boldsymbol{x}_t \mid \boldsymbol{C}\overline{\boldsymbol{\alpha}}_t) \\ &= \mathrm{Cat}(\boldsymbol{x}_t \mid g(\overline{\boldsymbol{\alpha}}_t; \boldsymbol{\theta})).\end{aligned}$$

Framed this way, a categorical HMM can be seen as a simple recurrent neural network!

A Cool Trick for Computing the Gradients

This formulation suggests that we could estimate the parameters of an HMM by directly maximizing the log likelihood,

$$\mathcal{L}(\boldsymbol{\theta}) = \log p(\boldsymbol{x}_{1:T}; \boldsymbol{\theta}) = \sum_{t=1}^{T} \log p(\boldsymbol{x}_t \mid \boldsymbol{x}_{1:t-1}; \boldsymbol{\theta}).$$

With automatic differentiation at our disposal, this sounds like it might be a lot easier!

Let’s pursue this idea a little further. First, we’d prefer to do unconstrained optimization, so let’s parameterize the model in terms of $\log \boldsymbol{P}$ and $\log \boldsymbol{C}$.

Aside: The softmax function

When we need the constrained versions, we will just apply the softmax to obtain simplex vectors:

$$\operatorname{softmax}(\log \boldsymbol{c}_k) = \left(\frac{e^{\log c_{k,1}}}{\sum_{v=1}^{V} e^{\log c_{k,v}}}, \ldots, \frac{e^{\log c_{k,V}}}{\sum_{v=1}^{V} e^{\log c_{k,v}}} \right)^\top$$

Softmax is translation invariant Note that the softmax operation is translation invariant,

$$\operatorname{softmax}(\log \boldsymbol{c}_k) = \operatorname{softmax}(\log \boldsymbol{c}_k + a)$$

for any constant $a \in \mathbb{R}$.

Thus, we will call our optimization variables $\log \boldsymbol{C}$ and $\log \boldsymbol{P}$, but they are not actually the log of matrices with simplex columns or rows; they are unconstrained parameters that become properly normalized via the softmax.

A Cool Trick for Computing the Gradients

To maximize the likelihood with gradient ascent, we need the Jacobians, $\frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \log \boldsymbol{P}}$ and $\frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \log \boldsymbol{C}}$. For the RNNs above, we computed them using backpropagation through time, but here we can use an even cooler trick.

A Cool Trick for Computing the Gradients

First, note that the posterior distribution in an HMM can be written as an exponential family,

$$
\begin{aligned}
p(\boldsymbol{z}_{1:T} \mid \boldsymbol{x}_{1:T}; \boldsymbol{\theta}) &= \exp\left\{\log p(\boldsymbol{z}_{1:T}, \boldsymbol{x}_{1:T}; \boldsymbol{\theta}) - \log p(\boldsymbol{x}_{1:T}; \boldsymbol{\theta})\right\} \\
&= \exp\left\{\log p(z_1; \pi_0) + \sum_{t=2}^{T} \log p(z_t \mid z_{t-1}; \boldsymbol{P}) + \sum_{t=1}^{T} \log p(\boldsymbol{x}_t \mid z_t; \boldsymbol{C}) - \log p(\boldsymbol{x}_{1:T}; \boldsymbol{\theta})\right\} \\
&= \exp\left\{\sum_{k=1}^{K} \langle \mathbb{I}[z_1 = k], \log \pi_{0,k} \rangle \right. \\
&\qquad + \sum_{t=2}^{T} \sum_{i=1}^{K} \sum_{j=1}^{K} \langle \mathbb{I}[z_{t-1} = i \wedge z_t = j], \log P_{i,j} \rangle \\
&\qquad\qquad + \sum_{t=1}^{T} \sum_{v=1}^{V} \sum_{k=1}^{K} \langle \mathbb{I}[x_t = v \wedge z_t = k], \log C_{v,k} \rangle \\
&\qquad\qquad\qquad \left. - A(\boldsymbol{\theta}) \right\}
\end{aligned}
$$

where the log marginal likelihood $A(\boldsymbol{\theta}) = \log p(\boldsymbol{x}_{1:T}; \boldsymbol{\theta})$ is the log normalizer.

A Cool Trick for Computing the Gradients

Recall that for exponential family distributions, gradients of the log normalizer yield expected sufficient statistics. Thus,

$$\frac{\partial A(\boldsymbol{\theta})}{\partial \log C_{v,k}} = \sum_{t=1}^{T} \mathbb{I}[x_t = v] \cdot \mathbb{E}_{p(\boldsymbol{z}_{1:T} \mid \boldsymbol{x}_{1:T}; \boldsymbol{\theta})} \left[\mathbb{I}[z_t = k]\right]$$

and

$$\frac{\partial A(\boldsymbol{\theta})}{\partial \log P_{i,j}} = \sum_{t=2}^{T} \mathbb{E}_{p(\boldsymbol{z}_{1:T} \mid \boldsymbol{x}_{1:T}; \boldsymbol{\theta})} \left[\mathbb{I}[z_{t-1} = i \wedge z_t = j]\right]$$

The gradients are essentially the posterior marginals and pairwise marginals we computed in the EM algorithm!

In the M-step of EM, we solve for the parameters that satisfy the constrained optimality conditions, whereas in SGD we just take a step in the direction of the gradient. EM tends to converge must faster in practice for this reason.

Linear RNNs

Consider the special case of *linear* RNNs,

$$\boldsymbol{h}_t = \boldsymbol{A}\boldsymbol{h}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t.$$

with outputs $\boldsymbol{y}_t = \boldsymbol{C}\boldsymbol{h}_t + \boldsymbol{d}$.

Since a linear operators compose, the mapping from $\boldsymbol{x}_{1:T}$ to $\boldsymbol{y}_{1:T}$ is a linear function,

$$\begin{aligned}
\boldsymbol{y}_t &= \boldsymbol{C}\boldsymbol{h}_t + \boldsymbol{d} \\
&= \boldsymbol{C}(\boldsymbol{A}\boldsymbol{h}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t) + \boldsymbol{d} \\
&= \boldsymbol{C}(\boldsymbol{A}(\boldsymbol{A}\boldsymbol{h}_{t-2} + \boldsymbol{B}x_{t-1}) + \boldsymbol{B}\boldsymbol{x}_t) + \boldsymbol{d} \\
&= \boldsymbol{C}\left(\sum_{s=0}^{t-1} \boldsymbol{A}^s \boldsymbol{B}\boldsymbol{x}_{t-s}\right) + \boldsymbol{d}
\end{aligned}$$

with the convention that $\boldsymbol{h}_0 = 0$.

Linear RNNs

This is a **convolution**,

$$\boldsymbol{y}_t = [\boldsymbol{K} \circledast \boldsymbol{x}]_t$$

with **kernel**,

$$\boldsymbol{K} = \begin{bmatrix} \boldsymbol{CB} & \boldsymbol{CAB} & \boldsymbol{CA}^2\boldsymbol{B} \cdots & \boldsymbol{CA}^{T-1}\boldsymbol{B} \end{bmatrix}.$$

PyTorch and JAX have highly optimized, parallel convolution routines for evaluating the hidden states. Likewise, sampling the model is straightforward and efficient using the RNN formulation.

While these simple linear RNNs may seem too simple to capture complex sequential dependencies, it turns out that stacks of linear layers with simple nonlinearities in between — a *deep* linear state space model (Gu et al., 2021) — can be highly expressive!

Matrix Powers Note that the kernel involves **matrix powers $\boldsymbol{A}^t$**, which typically are cubic in the hidden state dimension. Gu et al. (2021) derived clever algorithms for efficiently computing the kernel and evaluating the convolution for certain structured classes of dynamics matrices.

Input-dependent dynamics with parallel scan

One limitation of the convolutional formulation above is that it requires the dynamics matrix $\boldsymbol{A}$ to be the same at all time-steps. Smith et al. (2023) showed an alternative way to evaluate the linear RNN that relaxes this constraint.

Consider two consecutive state updates, now with time-dependent dynamics matrices $\boldsymbol{A}_t$ and affine terms $\boldsymbol{b}_t = \boldsymbol{B}_t \boldsymbol{x}_t$,

$$\begin{aligned} h_t &= \boldsymbol{A}_t \boldsymbol{h}_{t-1} + \boldsymbol{b}_t \\ &= \boldsymbol{A}_t(\boldsymbol{A}_{t-1}\boldsymbol{h}_{t-2} + \boldsymbol{b}_{t-1}) + \boldsymbol{b}_t \\ &= \boldsymbol{A}_{t-2:t}\boldsymbol{h}_{t-2} + \boldsymbol{b}_{t-2:t} \end{aligned}$$

where $\boldsymbol{A}_{t-2:t} \triangleq \boldsymbol{A}_t \boldsymbol{A}_{t-1}$ and $\boldsymbol{b}_{t-2:t} \triangleq \boldsymbol{A}_t \boldsymbol{b}_{t-1} + \boldsymbol{b}_t$.

Input-dependent dynamics with parallel scan

This is just another affine map, but now it takes $\boldsymbol{h}_{t-2}$ to $\boldsymbol{h}_t$. In parallel, we can compute the maps from time t to $t+2$, from $t+2$ to $t+4$, and so on.

In the next iteration, we can combine these linear maps to obtain $(\boldsymbol{A}_{t-4:t}, \boldsymbol{b}_{t-4:t})$, and so on.

After $\log_2 T$ iterations, we obtain a map from $\boldsymbol{h}_0$ to $\boldsymbol{h}_T$. With $\mathcal{O}(\log T)$ more work, we can obtain maps from $\boldsymbol{h}_0$ to all intermediate times t as well.

This algorithm is called a **parallel scan** or **binary associative scan**, since it is based on a binary associative operator, $\circ$, of the form,

$$(\boldsymbol{A}_{i,j}, \boldsymbol{b}_{i,j}) \circ (\boldsymbol{A}_{j,k}, \boldsymbol{b}_{j,k}) = (\boldsymbol{A}_{i,k}, \boldsymbol{b}_{i,k})$$

With T parallel processors, it requires only $\mathcal{O}(\log T)$ time and $\mathcal{O}(T)$ memory.

Complex Diagonal Matrices Again, notice that each update requires matrix multiplication, which is typically cubic in the state dimension. Smith et al. (2023) proposed to work with *complex diagonal* matrices instead, which keeps the time and memory costs in check.

Parallelizing Nonlinear RNNs

The parallel scan relied on the linearity of the RNN dynamics. Can the same be applied to nonlinear RNNs?

It turns out that yes, in many cases we can speed up the evaluation of nonlinear RNNs using a similar trick!

Consider an RNN with nonlinear dynamics function $f(\boldsymbol{h}_t)$. Form a first-order Taylor approximation arond an initial guess, $\boldsymbol{h}_t^{(0)}$,

$$\widetilde{f}(\boldsymbol{h}_t) = f(\boldsymbol{h}_t^{(0)}) + J_f(\boldsymbol{h}_t^{(0)}) \left[\boldsymbol{h}_t - \boldsymbol{h}_t^{(0)} \right].$$

where $J_f(\boldsymbol{h}_t^{(0)}) = \frac{\partial f}{\partial \boldsymbol{h}_t}\Big|_{\boldsymbol{h}_t = \boldsymbol{h}_t^{(0)}}$ is the Jacobian of f evaluated at $\boldsymbol{h}_t^{(0)}$.

Parallelizing Nonlinear RNNs

The Taylor approximations yield a *linear* RNN with time-varying dynamics, which can be evaluated with a parallel scan to obtain a new sequence of latent states, $(\boldsymbol{h}_1^{(1)}, \ldots, \boldsymbol{h}_T^{(1)})$, which can be used as the guess for the next iteration.

Repeating this process until convergence is equivalent to the **Gauss-Newton method** for minimizing the sum of squares loss function,

$$\mathscr{L}(\boldsymbol{h}_{1:T}) = \sum_{t=1}^{T} \|\boldsymbol{h}_t - f(\boldsymbol{h}_{t-1})\|_2^2.$$

This idea was proposed by Lim et al (2023) and called DEER. Gonzalez et al (2024) explained the connection to the Gauss-Newton method and proposed an extension called ELK.

Conclusion

Recurrent neural networks are foundational models for sequential data.

They're useful machine learning models, and they're standard models in theoretical neuroscience.

While Transformers are the star of modern large language models, RNNs are making a comeback as efficient techniques for capturing long range dependences in sequential data.