Recent Trends in Machine Learning Time Series

dsai.asia

Asian Data Science and Artificial Intelligence Master's Program

Co-funded by the Erasmus+ Programme of the European Union

Readings

Readings for these lecture notes:

- Goodfellow, I., Bengio, Y., and Courville, A. (2016), Deep Learning. MIT Press.

These notes contain material © Goodfellow et al. (2016).

Outline

Introduction

In this series, we discuss time series modeling.

We are face with a sequence of values $x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$.

If we use a fixed-size input, we can only consider a sliding window over time. We could use a 1D CNN for this.

A recurrent neural network is a network specialized for processing sequential data that can (usually) handle arbitrary-length sequences.

Like CNNs, RNNs use the principle of parameter sharing to allow flexible processing of information that could appear anywhere in the sequence.

Introduction

Goodfellow's example:

I went to Nepal in 2009.
In 2009, I went to Nepal.

Both sentences contain similar information at different positions.

Parameter sharing will help us form a compact model that applies the same rules or similar rules at different positions in the input.

Outline

Unfolding cyclic computations

A dynamical system has the form

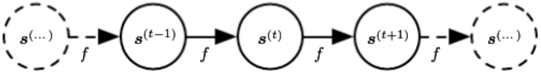
$$s^{(t)} = f(s^{(t-1)}; \boldsymbol{\theta}),$$

where $s^{(t)}$ is the state of the system at time t.

Considering the state after a particular number of steps τ, we observe

$$\begin{aligned} s^{(3)} &= f(s^{(2)}; \boldsymbol{\theta}) \\ &= f(f(s^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta}) \end{aligned}$$

This removal of recurrence is called unfolding the computation. The unfolded computatational graph looks like this:



Goodfellow, Bengio, and Courville (2016), Fig. 10.1

Unfolding cyclic computations

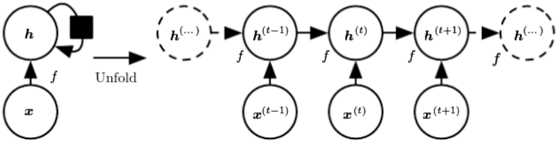
Now consider a dynamical system driven by an input $\mathrm{x}^{(t)}$:

$$\mathrm{s}^{(t)} = f(\mathrm{s}^{(t-1)}, \mathrm{x}^{(t)}; \boldsymbol{\theta}),$$

In the neural network community, we would use h rather than x as a hint that the state of the system is hidden and represented by hidden units in the model:

$$\mathrm{h}^{(t)} = f(\mathrm{h}^{(t-1)}, \mathrm{x}^{(t)}; \boldsymbol{\theta}),$$

The recurrent circuit (with black square indicating a time delay) on the left can be unfolded into the acyclic computational graph on the right:



Goodfellow, Bengio, and Courville (2016), Fig. 10.2

Unfolding cyclic computations

One of the most common tasks of a RNN is to predict the future from the past.

A model trained to do this will use $\mathrm{h}^{(t)}$ to form a lossy summary of the inputs $\mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(t)}$ up to time t.

Unfolding a circuit can be modeled mathematically by replacing the recurrent function $f(\cdot, \cdot; \cdot)$ with its unfolded version $g(\ldots)$:

$$\begin{aligned} \mathrm{h}^{(t)} &= g^{(t)}(\mathrm{x}^{(t)}, \mathrm{x}^{(t-1)}, \ldots, \mathrm{x}^{(1)}) \\ &= f(\mathrm{h}^{(t-1)}, \mathrm{x}^{(t)}; \boldsymbol{\theta}) \end{aligned}$$

Our goal, then, is to learn parameters $\boldsymbol{\theta}$ of the single model $f(\cdot, \cdot; \boldsymbol{\theta})$ using the unfolded computation $g^{(t)}(\ldots)$.

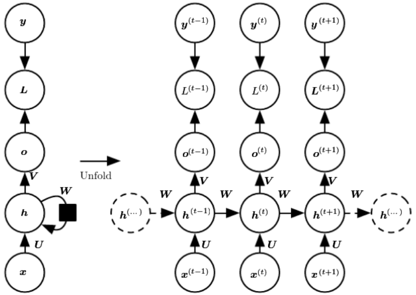
Outline

Recurrent neural networks

RNN types

Now we can consider different types of RNNs.

This "Elman" network produces an output at each time t and has recurrent connections between its hidden units:



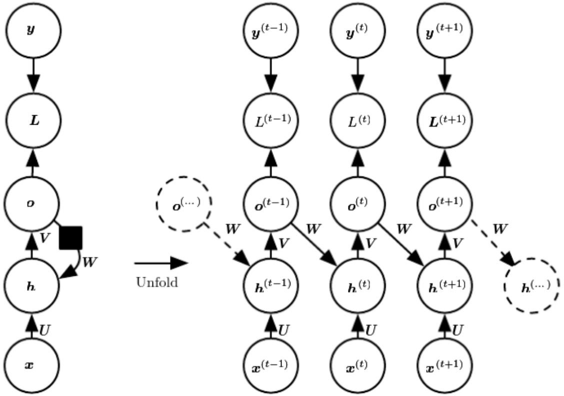
Goodfellow, Bengio, and Courville (2016), Fig. 10.3

L is a loss function that can be factored into individual comparisons $L^{(t)}(\mathrm{o}^{(t)}, \mathrm{y}^{(t)})$ of the actual output $\mathrm{o}^{(t)}$ with the desired output $\mathrm{y}^{(t)}$.

Recurrent neural networks

RNN types

This "Jordan" network produces an output at each time *t* and has recurrent connections from the output at one time step to the hidden units at the next step:

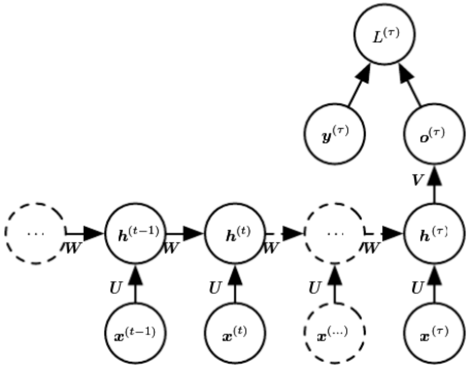


Goodfellow, Bengio, and Courville (2016), Fig. 10.4

Recurrent neural networks

RNN types

This network is similar to the Elman network but reads an entire sequence before producing an output.



Goodfellow, Bengio, and Courville (2016), Fig. 10.5

Recurrent neural networks

Assumptions

The Elman network (the first of the three types, Figure 10.3 in Goodfellow et al.) is typical and is Turing complete.

Let's make some assumptions:

- The hidden units use tanh activation functions
- The output $\text{o}^{(t)}$ is a vector of unnormalized log probabilities for a discrete multinomial ouptut
- The output vector is softmaxed to obtain $\hat{\text{y}}^{(t)}$.
- The loss function is negative log likelihood for the discrete multinomial output

Based on these assumptions, we can apply backpropagation to the unfolded model. This is called backpropagation through time.

Recurrent neural networks
BPTT

For the output at time t:[1]

$$(\nabla_{\mathbf{o}^{(t)}} L)_i = \hat{y}_i^{(t)} - \delta_{i,y^{(t)}}$$

This is true for any t, because we assume the loss is applied independently for each element in the sequence.

For the hidden layer, at the last $t = \tau$, if the hidden-to-output weights are $\mathtt{V}$, we have

$$\nabla_{\mathbf{h}^{(\tau)}} L = \mathtt{V}^\top \nabla_{\mathbf{o}^{(\tau)}} L$$

[1] $\delta_{i,j}$ is the Kronecker delta (1 if $i = j$, 0 otherwise).

Recurrent neural networks
BPTT

For other times $t < \tau$, we apply the chain rule, considering the effect of $\mathsf{h}^{(t)}$ on both $\mathsf{o}^{(t)}$ and $\mathsf{h}^{(t+1)}$:[2]

$$\begin{aligned}
\nabla_{\mathsf{h}^{(t)}} L &= \frac{\partial \mathsf{h}^{(t+1)}}{\partial \mathsf{h}^{(t)}} (\nabla_{\mathsf{h}^{(t+1)}} L) + \frac{\partial \mathsf{o}^{(t)}}{\partial \mathsf{h}^{(t)}} (\nabla_{\mathsf{o}^{(t)}} L) \\
&= \mathtt{W}^\top \operatorname{diag}\left(1 - \left(\mathsf{h}^{(t+1)}\right)^2\right) (\nabla_{\mathsf{h}^{(t+1)}} L) + \mathtt{V}^\top (\nabla_{\mathsf{o}^{(t)}} L)
\end{aligned}$$

Once these gradients are computed, we can compute the responsibility of each weight for the total loss.

[2] Recall that $\tanh(z) = (e^z - e^{-z})/(e^z + e^{-z})$ and $\tanh'(z) = (1 - \tanh^2(z))dz$.

Recurrent neural networks
BPTT

We introduce dummy variables $\mathtt{W}^{(t)}$ to indicate the value of the weights at time t, i.e., $w_{ij}^{(t)}$ connects $h_j^{(t-1)}$ to $h_i^{(t)}$.

Actually, during one iteration, the weights do not change ($\mathtt{W}^{(t)} = \mathtt{W}^{(t')}$).

However, the dummy variables will let us calculate the contribution of each weight to the loss $\nabla_{\mathtt{W}^{(t)}} L$ at each timestep separately.

For a particular weight $w_{ij}^{(t)}$ at time t and the hidden unit $h_i^{(t)}$ it's connected to, we have

$$\frac{\partial L}{\partial w_{ij}^{(t)}} = \frac{\partial L}{\partial h_i^{(t)}} \frac{\partial h_i^{(t)}}{\partial w_{ij}^{(t)}}.$$

Recurrent neural networks
BPTT

Summing over time, we obtain

$$\nabla_{\mathtt{W}} L = \sum_t \mathrm{diag}\left(1 - \left(\mathrm{h}^{(t)}\right)^2\right)(\nabla_{\mathrm{h}^{(t)}} L)\mathrm{h}^{(t-1)\top}.$$

For $\mathtt{U}$ connecting $\mathrm{x}^{(t)}$ to $\mathrm{h}^{(t)}$, we obtain the similar

$$\nabla_{\mathtt{U}} L = \sum_t \mathrm{diag}\left(1 - \left(\mathrm{h}^{(t)}\right)^2\right)(\nabla_{\mathrm{h}^{(t)}} L)\mathrm{x}^{(t)\top}.$$

For the hidden unit biases b, we obtain

$$\nabla_{\mathrm{b}} L = \sum_t \mathrm{diag}\left(1 - \left(\mathrm{h}^{(t)}\right)^2\right)\nabla_{\mathrm{h}^{(t)}} L.$$

Recurrent neural networks
BPTT

For $\mathtt{V}$ connecting $\mathsf{h}^{(t)}$ to $\mathsf{o}^{(t)}$, we obtain the simpler

$$\nabla_{\mathtt{U}} L = \sum_t (\nabla_{\mathsf{o}^{(t)}} L) \mathsf{h}^{(t)\top},$$

and for the output biases c it is easy to derive

$$\nabla_{\mathsf{c}} L = \sum_t \nabla_{\mathsf{o}^{(t)}} L.$$

Outline

RNNs as directed graphical models

Modeling a joint distribution

Thus far, our RNNs have modeled

$$p(\mathrm{y}^{(t)} \mid \mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(t)})$$

or

$$p(\mathrm{y}^{(t)} \mid \mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(t)}, \mathrm{y}^{(1)}, \ldots, \mathrm{y}^{(t-1)}),$$

the difference being whether $\mathrm{y}^{(t-1)}$ is an input to the model at time t or not.

To better understand RNNs, we'll see how they can model joint distributions over a scalar sequence $Y^{(1)} = y^{(1)}, \ldots, Y^{(\tau)} = y^{(\tau)}$, leaving out any external inputs x for now:

$$P(\mathbb{Y}) = P(Y^{(1)}, \ldots, Y^{(\tau)})$$

RNNs as directed graphical models

Factoring a joint distribution

We know that $P(\mathbb{Y})$ can always be factored as

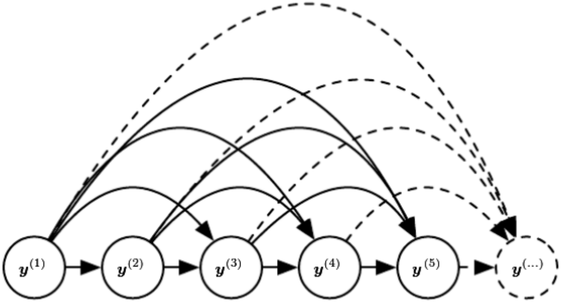
$$P(\mathbb{Y}) = \prod_{i=1}^{\tau} P(Y^{(t)} \mid Y^{(t-1)}, \ldots, Y^{(1)}).$$

If we wanted to estimate the parameters of $P(\mathbb{Y})$ using maximum likelihood, we would try to minimize

$$L = \sum_t L^{(t)} = -\sum_t \log P(Y^{(t)} = y^{(t)} \mid Y^{(t-1)} = y^{(t-1)}, \ldots, Y^{(1)} = y^{(1)}).$$

RNNs as directed graphical models
Fully connected graphical model

This corresponds to the “fully connected” graphical model



Goodfellow, Bengio, and Courville (2016), Fig. 10.7

The estimation procedure gets more complex each time we add an element to the sequence. If $Y^{(t)}$ is discrete with k values, the number of parameters is $O(k^{\tau})$!

RNNs as directed graphical models
Markov assumption

To avoid this increase in complexity in a graphical model, we usually make a Markov assumption that the distribution at time t only depends on the last k steps:

$$P(Y^{(t)} = y^{(t)} \mid Y^{(t-1)} = y^{(t-1)}, \ldots, y^{(1)}) =$$
$$P(Y^{(t)} = y^{(t)} \mid Y^{(t-1)} = y^{(t-1)}, \ldots, y^{(t-k)}).$$

RNNs as directed graphical models

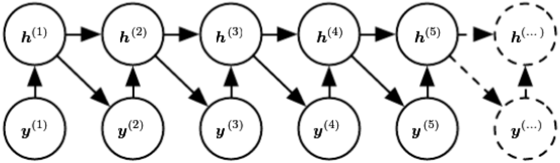
RNNs avoid Markov assumption

RNNs capture dependency more flexibly than the Markov assumption.

The model's state can remember values $y^{(i)}$ for any previous step i and capture a dependency of $Y^{(t)}$ on the fact that $Y^{(i)} = y^{(i)}$.

When we marginalize out the state $\mathrm{h}^{(t)}$ in a RNN, we get the fully connected model we previously saw (Figure 10.7).

But when we incorporate the RNN state $\mathrm{h}^{(t)}$, we decouple $Y^{(t)}$ from $Y^{(t-1)}, \ldots, Y^{(1)}$, and the number of parameters no longer depends on τ:



Goodfellow, Bengio, and Courville (2016), Fig. 10.8

RNNs as directed graphical models

Sampling

If we want to use a RNN as a generative model (sample from it), we just sample from the conditional distribution.

Determining the number of elements τ in the sequence is tricky but can be solved by using

- A special stop symbol or output variable indicating whether to stop.
- Sample from a distribution over τ.

With these techniques, our RNN can be used to synthesize sequential data.

RNNs as directed graphical models

Conditioning on input

Now, we would like to add the input $\mathrm{x}^{(t)}$.

We already have a model capable of representing $P(\mathrm{y};\boldsymbol{\theta})$.

To take x into account, we turn the model into a conditional one: $P(\mathrm{y} \mid \boldsymbol{\omega})$ with $\boldsymbol{\omega} = \boldsymbol{\theta}$ (we introduce a random variable $\boldsymbol{\omega}$ with value fixed to $\boldsymbol{\theta}$.

Next, we extend the model to $\boldsymbol{\omega}$ being a function of an input x.

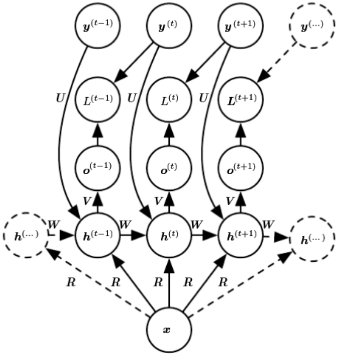
The input x could be

- An extra input at each time step t
- The initial value of the hidden state h
- Both

RNNs as directed graphical models

Conditioning on input

A common case is feeding the same fixed-size input x to the model at every time:



Goodfellow, Bengio, and Courville (2016), Fig. 10.9

An example is image captioning, in which the image is static but the output is a sequence of tokens (words).

RNNs as directed graphical models

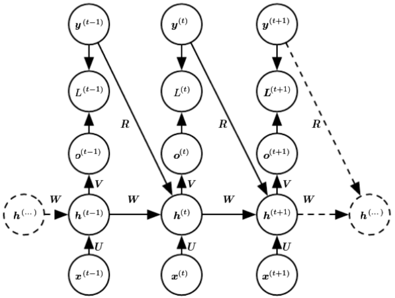
Conditioning on input

Another common scenario: input that varies over time.

This model is more powerful than the Elman model of Figure 10.3 which assumes

$$P(\mathrm{y}^{(1)}, \ldots, \mathrm{y}^{(\tau)} \mid \mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(\tau)}) = \prod_t P(\mathrm{y}^{(t)} \mid \mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(t)}).$$

By feeding $\mathrm{y}^{(t-1)}$ to $\mathrm{y}^{(t)}$, we can model arbitrary dependencies among the $\mathrm{y}^{(t)}$.



Goodfellow, et al. (2016), Fig. 10.10

Outline

Specialized RNN structures

Bidirectional RNNs

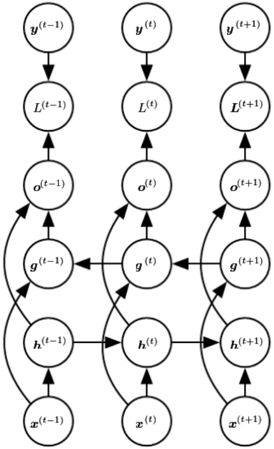
Thus far, the models we've seen have been causal.

The sequence is processed in one direction only, so future inputs cannot influence decisions we make at time t.

Alternative: bidirectional RNNs process in left-to-right and right-to-left concurrently.

Output $o^{(t)}$ is conditional not only on $h^{(t)}$ but also $g^{(t)}$, which summarizes all "future" inputs.

The approach can be generalized to 2D data (e.g., images) with four directions.



Goodfellow et al. (2016), Fig. 10.11

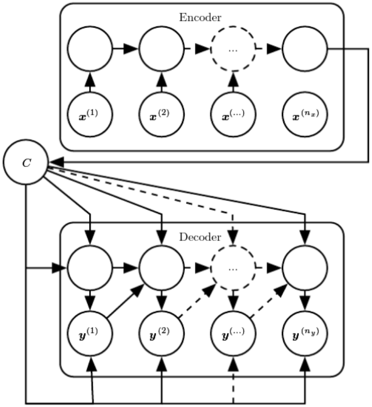
Specialized RNN structures

Encoder-decoder architectures

When we are modeling translations between variable-length sequences, a very powerful modern architecture is the sequence to sequence or encoder-decoder architecture.

Similar models were introduced in 2014 by Google (seq2seq) and Cho et al. (encoder-decoder).

We explore the ability of this model to translate between languages in lab.



Goodfellow et al. (2016), Fig. 10.12

Specialized RNN structures
Deep RNNs

Thus far the models we've seen have been relatively shallow.

We have blocks of parameters for

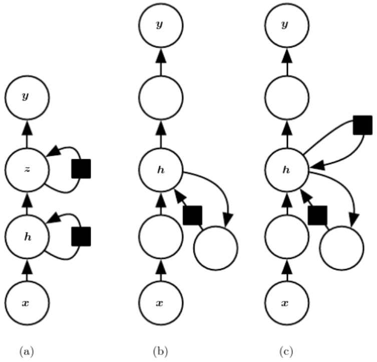
- Input to hidden
- Hidden to output
- Hidden to hidden

RNNs can be made deeper at multiple levels.

Empirically, this has been shown to improve performance on large complex problems.

Specialized RNN structures
Deep RNNs

Example: adding additional layers in the hidden-to-hidden transformation.



Goodfellow, Bengio, and Courville (2016), Fig. 10.13

Models like this are more difficult to optimize, but using skip connections (rightmost architecture) helps.

Specialized RNN structures
Recursive RNNs

Recursive RNNs use a tree structure to process the input rather than a chain.

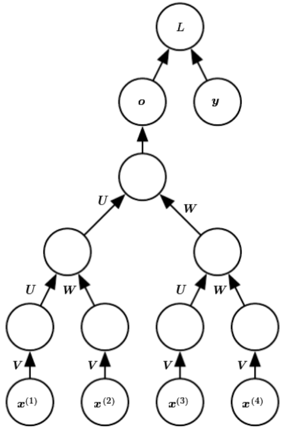
Variable-length input of length τ can be processed with $\log(\tau)$ parameters.

Processing can be parallelized.

Issues include how to structure the tree or how to learn an appropriate structure of the tree.

If the model is processing a rich data structure that has a tree structure already, such as a parse tree, the approach is very efficient.

There are many variations on the idea.



Goodfellow et al. (2016), Fig. 10.14

Outline

Dealing with long-term dependences

Vanishing and exploding gradients

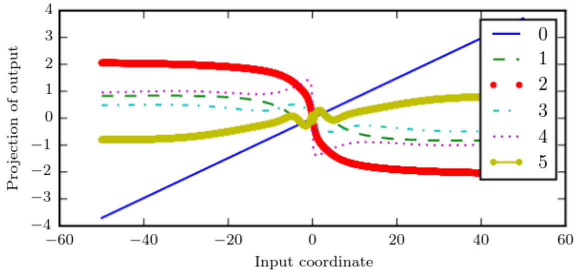
The big problem with RNNs is long-term dependencies.

Gradients propagated over many stages tend to vanish or explode.

Vanishing gradients are most often the problem. We'll do a brief analysis of the problem here.

Dealing with long-term dependences
Nonlinearity

Repeated computations over multiple stages introduce nonlinearity that becomes more extreme as the number of stages increases:



Goodfellow, Bengio, and Courville (2016), Fig. 10.15

(The legend shows the number of steps in the recurrent calculation, the x axis shows an input along a random linear direction in the high dimensional input space, and the y axis shows a projection of the resulting output.)

Dealing with long-term dependences

Multi-stage calculations

Without any nonlinearity in the hidden layer calculation, part of a RNN's computation will be something like

$$\mathtt{h}^{(t)} = \mathtt{W}^\top \mathtt{h}^{(t-1)}.$$

Unfolding, we obtain

$$\mathtt{h}^{(t)} = \left(\mathtt{W}^t\right)^\top \mathtt{h}^{(0)}.$$

If $\mathtt{W}$ is diagonalizable,[3] it can be factored using an eigendecomposition

$$\mathtt{W} = \mathtt{Q}\Lambda\mathtt{Q}^T$$

with $\mathtt{Q}$ containing (orthogonal) eigenvectors of $\mathtt{W}$ and Λ a diagonal matrix containing the eigenvectors of $\mathtt{W}$.

This means we have

$$\mathtt{h}^{(t)} = \mathtt{Q}^\top \Lambda^t \mathtt{Q}\mathtt{h}^{(0)}.$$

[3] A non-diagonalizable square matrix is called defective. A defective matrix is one that has less than n distinct eigenvalues.

Dealing with long-term dependences

Nonlinearity

Note if τ is long, any entry in Λ will explode if it greater than one or vanish if it is less than one.

Also, any element of $h^{(0)}$ not aligned with the largest eigenvector of W will eventually be eliminated.

In feedforward networks, the problem is solved by using different W at each step of the feedforward calculation.

In a recurrent network, however, the gradient of a long-term interaction will necessarily be exponentially smaller than the gradient of a short-term interaction.

This makes it impractical to learn long-term interactions beyond 10 or 20 elements.

Dealing with long-term dependences

Echo state networks

Echo state networks and their cousins attempt to use hidden-to-hidden weights that efficiently store the input sequence.

Then only the hidden-to-output weights (short term interactions) need to be learned.

Dealing with long-term dependences

Multiple time scales

A different approach is to have a model that operates at multiple time scales.

Adding a skip connection, of length d, for example, decreases the length of the path from time t to time $t - d$ from d to 1.

Hidden units with linear self connections can give paths with a product of derivatives close to 1, minimizing the vanishing or exploding of gradients.

Units with linear self-connections are called leaky units.

It is also possible to remove short-term connections and replace them with long-term ones.

Dealing with long term dependencies is one of the open areas of research. The most effective strategy we have up till now is gating.

Outline

Gated RNNs

The most effective RNNs known today are gated RNNs:

- Long short-term memory (LSTM)
- Gated recurrent units (GRUs)

Basic idea: create a path through time that has derivatives that neither vanish nor explode.

Gated units use the idea of weights that can change at each time step to avoid vanishing/exploding that occurs when repeating the same transformation repeatedly.

Besides accumulating information over time like leaky units, we want to forget information that is no longer useful.

Gated RNNs learn when to forget by resetting their hidden state to 0.

Gated RNNs
LSTMs

LSTM was introduced by Hochreiter and Schmidhuber in 1997.

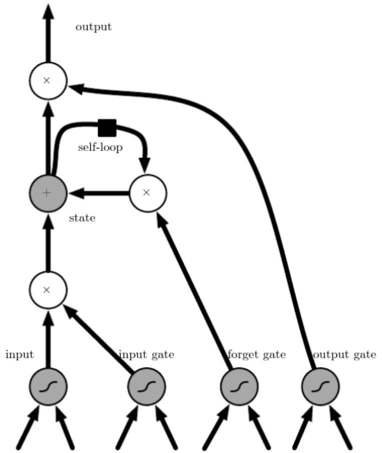
The model uses the idea of self loops to increase the practical length of interactions without vanishing gradients.

The self-loop is conditioned on the context rather than being fixed.

We can therefore think of the time constants for integration of information over long periods of time as being determined by the model once it sees the input.

Gated RNNs

LSTMs



Goodfellow, Bengio, and Courville (2016), Fig. 10.16

Gated RNNs
LSTMs

The forget gate computes forget outputs

$$\mathtt{f}^{(t)} = \sigma\left(\mathtt{b}^{f} + \mathtt{U}^{f}\mathtt{x}^{(t)} + \mathtt{W}^{f}\mathtt{h}^{(t-1)}\right)$$

The external input gate computes outputs

$$\mathtt{g}^{(t)} = \sigma\left(\mathtt{b}^{g} + \mathtt{U}^{g}\mathtt{x}^{(t)} + \mathtt{W}^{g}\mathtt{h}^{(t-1)}\right)$$

then the state units compute the state

$$\mathtt{s}^{(t)} = \mathtt{f}^{(t)} \odot \mathtt{s}^{(t-1)} + \mathtt{g}^{(t)} \odot \sigma\left(\mathtt{b} + \mathtt{U}\mathtt{x}^{(t)} + \mathtt{W}\mathtt{h}^{(t-1)}\right)$$

Gated RNNs
LSTMs

In the meantime, the output gate computes outputs

$$\mathrm{q}^{(t)} = \sigma\left(\mathrm{b}^{o} + \mathtt{U}^{o}\mathrm{x}^{(t)} + \mathtt{W}^{o}\mathrm{h}^{(t-1)}\right),$$

then the final hidden state output by the LSTM module is

$$\mathrm{h}^{(t)} = \tanh\left(\mathrm{s}^{(t)}\right) \odot \mathrm{q}^{(t)}.$$

Sometimes the internal state $\mathrm{s}^{(t-1)}$ is used as an additional input to the gates at time t.

LSTMs have been shown to learn long term dependencies more easily than the state units in ordinary RNNs.

They are the basis of seq2seq (Sutskever et al., 2014) and many other successful models.

Gated RNNs
GRUs

LSTM is extremely successful but a little complicated.

We would like to know what is essential and what is unnecessary in the LSTM architecture.

The Gated Recurrent Unit (GRU) is similar but slightly simpler:

$$\mathrm{h}^{(t)} = \mathrm{u}^{(t-1)} \odot \mathrm{h}^{(t-1)} + (1 - \mathrm{u}^{(t-1)})\sigma\left(\mathrm{b} + \mathtt{U}\mathrm{x}^{(t)} + \mathtt{W}\left(\mathrm{r}^{(t-1)} \odot \mathrm{h}^{(t-1)}\right)\right)$$

u stands for update gate, and r stands for reset gate:

$$\mathrm{u}^{(t)} = \sigma\left(\mathrm{b}^{u} + \mathtt{U}^{u}\mathrm{x}^{(t)} + \mathtt{W}^{u}\mathrm{h}^{(t)}\right)$$

$$\mathrm{r}^{(t)} = \sigma\left(\mathrm{b}^{r} + \mathtt{U}^{r}\mathrm{x}^{(t)} + \mathtt{W}^{r}\mathrm{h}^{(t)}\right)$$

Gated RNNs
GRUs

The update gate acts as a leaky integrator with amount of integration dependent on the input.

It can ignore the input (copying the old hidden state) or ignore the old hidden state (replacing it with the new target hidden state).

The reset and update gates can each selectively ignore parts of the state vector.

There have been many studies of variations. The upshot:

- The forget gates are critical (for LSTM and GRU).
- A bias of 1 for the LSTM forget gate is very useful.
- LSTM and GRU have similar performance across a wide variety of tasks, and no variations have proven definitely superior.

Outline

Optimization of RNNs
Second-order methods

Before the power of LSTM was realized, many attempts were made to deal with the vanishing gradient problem.

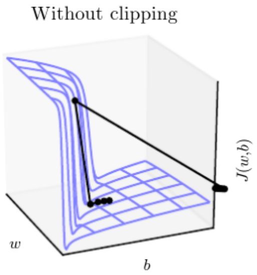
One approach was the use of second order optmization methods (Newton's method, essentially dividing first derivates by second derivatives, or more exactly, multiplying the gradient by the inverse Hessian).

These techniques do not work as well as ordinary SGD with LSTMs!

Take-home message: it is better to design a model that is easy to optimize than to use fancy optmization methods.

Optimization of RNNs
Gradient clipping

One simple optimization technique that helps prevent exploding gradients gradient clipping, which prevents overshooting when going “down a cliff:”



Goodfellow, Bengio, and Courville (2016), Fig. 10.17

Clipping can be done elementwise (clipping only the too-large elements) or by a single scalar (limiting the length of the gradient vector but maintaining direction).

Optimization of RNNs
Gradient regularization

Another group of techniques attempt to prevent vanishing gradients by trying to maintain a large-enough gradient at every step over time.

Such gradient regularization techniques help with traditional RNNs but are not as effective as LSTMs.

Outline

Explicit memory
Memory networks

We know neural networks are very good at learning and storing implicit knowledge.

They are not so good at directly storing explicit information such as

- A cat is a type of animal
- There is a meeting with the sales team at 3:00 PM

Humans have some kind of working memory in which we store information currently needed to achieve an immediate goal.

Memory networks store and allow access to information indexed by addresses.

Explicit memory
Neural Turing machines

Memory cells in Neural Turing machines (NTMs) are similar to LSTMs but generate an internal state specifying which cell to read from or write to.

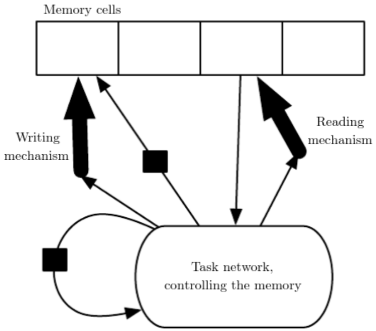
The memory access, rather than using integer address, outputs a set of weights used to compute a weighted average of many cells, for example via a softmax function.

The memory cells may contain an arbitrary-length vector, which is both efficient (one address indexes a larger memory array) and allows content-based addressing.

Explicit memory

Neural Turing machines

NTMs learn a task network that able to fetch and store information from explicit memory cells.



Goodfellow, Bengio, and Courville (2016), Fig. 10.18

Explicit memory

Types of memory access

Memory can be accessed in two ways:

- A deterministic method that makes soft decisions (weighted averages)
- A stochastic method that makes hard decisions by sampling.

Thus far, deterministic soft methods seem to perform better than hard stochastic methods.