Recurrent neural networks

- Motivation
- Network unrollwing
- Backpropagation through time
- Vanishing and exploding gradients
- LSTMs
- GRUs

Motivation for recurrent neural networks

A feedforward neural network, like a CNN, nonlinearly maps an input to an output. If there is no noise in the system, the same input will always produce the same output. (If there is noise, this introduces stochasticity to the mapping.)

But what if the inputs have structure over time? Say we wanted to classify a video, or do future character prediction. E.g., the inputs $\mathbf{x}_t$ can be a video frame at time t, or a character in a word at iteration t. In these scenarios, past inputs, $\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \ldots$ provide additional information that can be critical for the output, $\mathbf{z}_t$.

Motivation for recurrent neural networks (cont.)

One way to handle this is to allow a feedforward neural network to accept inputs over some time span going back p time bins. One could implement

$$\mathbf{z}_t = f(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_{t-p})$$

where $f(\cdot)$ is a CNN. While this is certainly possible, it in a sense is a brute-force approach to the problem. The number of input parameters will scale up linearly with the temporal extent of the input, resulting in inputs that are potentially very high-dimensional. Further, it can never learn temporal dependencies longer than p.

Motivation for recurrent neural networks (cont.)

Another way to handle this problem is to introduce the notion of a dynamical system. In the dynamical system, there is a *state* which captures information about past inputs. Typically, we make what is called a *Markov* assumption. Intuitively, this means that all the information about the past and current inputs that I need to compute a future state is succinctly held in my current state.

Unpacking this statement, this means that a state variable $\mathbf{s}_{t-1}$ contains all the information I need to know about $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{t-1}$ to compute $\mathbf{s}_t$. Subsequently, $\mathbf{s}_t$ contains all the information about $\mathbf{x}_1, \ldots, \mathbf{x}_t$ that I need to know to calculate $\mathbf{s}_{t+1}$. This suggests the following recurrence:

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}, \mathbf{x}_t)$$

This defines a *dynamical system*, where the state variable $\mathbf{s}_t$ evolves through time in lawful ways, so that $\mathbf{s}_t$ is informative of $\mathbf{s}_{t+1}$. Colloquially, this is like stating "where I am informs where I'm going to go."

Motivation for recurrent neural networks (cont.)

With this more compact representation of the state, we can infer the output at time t, given by $\mathbf{z}_t$, as a function of the state at time t, which contains all the useful information about the inputs up to time t for our task at hand.

$$\begin{aligned} \mathbf{z}_t &= g(\mathbf{s}_t) \\ &= g(f(\mathbf{s}_{t-1}, \mathbf{x}_t)) \end{aligned}$$

This bears a resemblance to our CNN operating over a history of inputs, with the following crucial modification: instead of passing in all inputs $\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}$, we can instead store a state variable $\mathbf{s}_{t-1}$ that succinctly summarizes all these past inputs, and use this variable to generate the output $\mathbf{z}_t$.

Motivation for recurrent neural networks (cont.)

A recurrent neural network is a particular type of dynamical system, where the state is stored via the activations of the artificial neurons in the network at time t, which we denote $\mathbf{h}_t$.

Along these lines, $\mathbf{h}_t$ should be defined by a recurrence relationship to $\mathbf{h}_{t-1}$, which means the artificial neurons in the network should not only have feedforward connections, but *feedback* connections as well.

Motivation for recurrent neural networks

Recurrent neural networks are neural networks that, in addition to feedforward connections, have feedback connections.

- A network that is purely feedforward has no *internal dynamics*. Such a network always produces the same output $\mathbf{y}_t = f(\mathbf{x}_t)$ no matter the time.
- In a recurrent neural network (RNN), the feedback connections cause input activations to persist for some amount of time. Because artificial units provide inputs to each other with feedback, an input provided to the network will, after one time step, still propagate within the network through its recurrent connectivity.
- In this manner, RNNs have internal dynamics, so that the output at any given time is also a function of the activation of the hidden units at that time, i.e., $\mathbf{z}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$.

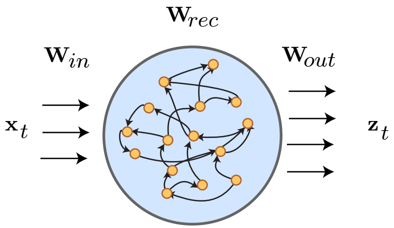
Connection to biology

In biology, neurons are known to be recurrently connected. Evidence suggests that while earlier areas of sensorimotor processing have more to do with encoding external stimuli, later areas such as the motor cortex are inherently *dynamical*. In such dynamical circuits, neurons, through their recurrent connectivity, drive themselves in lawful ways through time.

In a similar way, recurrent neural networks have recurrent connectivity that define a dynamical system that governs how it evolves through time. Note that a CNN does not capture these dynamical features of neural activity.

Continuous time RNN at a high-level

At a high-level, the RNN can be diagrammed as follows:



The RNN has three major components:

- $\mathbf{W}_{\text{in}}$: An input at time t, denoted $\mathbf{x}_t$, is transformed via $\mathbf{W}_{\text{in}}$ onto artificial neurons, whose activations are $\mathbf{h}_t$.
- $\mathbf{W}_{\text{rec}}$: Each artificial neuron is the network is denoted by an orange circle, and these artificial neurons have recurrent connections. recurrent connections are defined by the matrix $\mathbf{W}_{\text{rec}}$.
- $\mathbf{W}_{\text{out}}$: Finally, the artificial neuron activations are mapped linearly to the output $\mathbf{z}_t$ through the matrix $\mathbf{W}_{\text{out}}$.

Continuous time RNN model

Let the inputs to a network at time t be defined by the vector $\mathbf{x}_t \in \mathbb{R}^m$ and let the RNN be composed of n recurrently connected units, whose activations at time t are specified by the vector $\mathbf{h}_t \in \mathbb{R}^n$. For convenience, we let $\mathbf{s}_t$ denote the pre-nonlinearity activations, i.e., $\mathbf{h}_t = f(\mathbf{s}_t)$, where $f(\cdot)$ is the activation function. We call $\mathbf{s}_t$ the RNN state.

The RNN's recurrent dynamics are defined by the following equation:

$$\tau \dot{\mathbf{s}}_t = -\mathbf{s}_t + \mathbf{W}_{\text{rec}} f(\mathbf{s}_t) + \mathbf{b}_{\text{rec}} + \mathbf{W}_{\text{in}} \mathbf{x}_{t+1}$$

where

- $\dot{\mathbf{s}}_t$ is the derivative of the state at time t.
- $f(\cdot)$ defines an activation function (e.g., $\mathrm{ReLU}(\cdot)$, $\tanh(\cdot)$, etc.).
- $\mathbf{W}_{\text{rec}} \in \mathbb{R}^{n \times n}$ defines how the recurrent units are connected.
- $\mathbf{W}_{\text{in}} \in \mathbb{R}^{n \times m}$ defines how the inputs affect each artificial unit.
- τ is the *time-constant* of the network. We will explore the effect of τ in the next few slides.

First order Euler approximation of the RNN

Using a first order Euler approximation of $\dot{\mathbf{s}}_t$, i.e.,

$$\dot{\mathbf{s}}_t = \frac{\mathbf{s}_{t+\Delta t} - \mathbf{s}_t}{\Delta t}$$

we can re-write the RNN as a discrete state update equation. For notational simplicity, we'll fix Δt to some step size, and denote $\mathbf{s}_t$ to be activations one bin width Δt after $\mathbf{s}_{t-1}$.

Then, the RNN equation can be re-written as:

$$\mathbf{s}_t = (1-\alpha)\mathbf{s}_{t-1} + \alpha\left(\mathbf{W}_{\text{rec}} f(\mathbf{s}_{t-1}) + \mathbf{b}_{\text{rec}} + \mathbf{W}_{\text{in}}\mathbf{x}_t\right)$$

where $\alpha = \Delta t/\tau$. We will assume that $\Delta t \le \tau$, since it is not sensible to sample the network state at time scales larger than the time constant of the network.

First order Euler approximation of the RNN (cont.)

$$\mathbf{s}_t = (1-\alpha)\mathbf{s}_{t-1} + \alpha\left(\mathbf{W}_{\mathrm{rec}} f(\mathbf{s}_{t-1}) + \mathbf{b}_{\mathrm{rec}} + \mathbf{W}_{\mathrm{in}}\mathbf{x}_t\right)$$

- Hence, the updated network state can be seen as a convex combination of the prior network state, and an update as a result of the nonlinear dynamics.
- When $\alpha \approx 0$, indicating that τ is very large, then the system is driven almost entirely by slow exponential decay. There is very little contribution from recurrent nonlinear dynamics, $\mathbf{W}_{\mathrm{rec}}$, as well as the input to the system, $\mathbf{W}_{\mathrm{in}}\mathbf{x}_t$.
- When $\alpha \approx 1$, indicating τ is relatively small, then the contribution of the exponential decay is negligible, and the recurrent dynamics are almost entirely defined by the nonlinear dynamics through $\mathbf{W}_{\mathrm{rec}}$.

Common formulation of the vanilla RNN

Note that a common formulation of the vanilla RNN is to set $\alpha = 1$, meaning that the RNN equations update every τ. In this scenario, the equation of the RNN is:

$$\mathbf{s}_t = \mathbf{W}_{\text{rec}} f(\mathbf{s}_{t-1}) + \mathbf{b}_{\text{rec}} + \mathbf{W}_{\text{in}} \mathbf{x}_t$$

In this formulation, it can be re-written by taking $f(\cdot)$ of both sides, so that:

$$\mathbf{h}_t = f(\mathbf{W}_{\text{rec}} \mathbf{h}_{t-1} + \mathbf{b}_{\text{rec}} + \mathbf{W}_{\text{in}} \mathbf{x}_t)$$

Note, this is the vanilla RNN formulation used in Goodfellow (equations 10.8 and 10.9).

For the rest of this lecture, we will assume this formulation of the vanilla RNN for simplicity. However, note that changing Δt and τ can modify the behavior of your network. You may also, post-training, sample the RNN activations at different rates by modifying Δt.

RNN output

The output of the RNN is typically a linear mapping of the RNN's activations.

$$\mathbf{z}_t = \mathbf{W}_{\text{out}}\mathbf{h}_t + \mathbf{b}_{\text{out}}$$

This can be used e.g., for regression, or the linear outputs could e.g., be passed to a softmax classifier if the goal is to classify each time point. For example, the probability of class i at time t can be denoted via:

$$\hat{y}_{t,i} = \text{softmax}_i(\mathbf{z}_t)$$

RNN cost function

Loss functions are straightforward:

- In the case of regression, loss functions include the mean-square error, where $\hat{y}_t = \mathbf{z}_t$ and the cost function is to minimize the sum of residuals

  $$\sum_t \|\hat{\mathbf{y}}_t - \mathbf{y}_t\|^2$$

  across all time (modulo some scaling constant).
- In the case of classification, loss functions include those derived from maximum-likelihood. If $\mathcal{L}_t$ is the softmax loss at time t, then the cost function can be to minimize the sum of losses

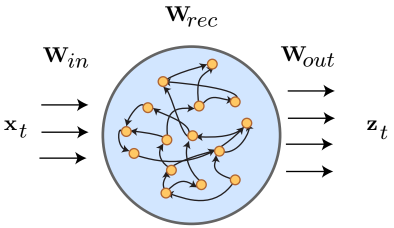
  $$\sum_t \mathcal{L}_t$$

  across all time (modulo some scaling constant).
- Sometimes, we only care about the output after some time τ. Maybe we'll even just care about the last output at the horizon of the data, T. In these scenarios, the loss would be

  $$\sum_{t \geq \tau} \mathcal{L}_t$$

RNN training

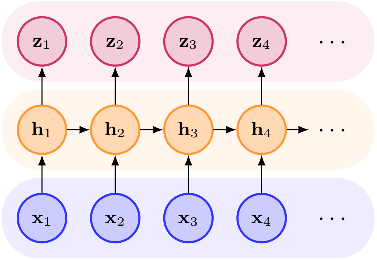
Training an RNN is not immediately as straightforward as a feedforward neural network. This is because the RNN has recurrent connections with loops, and backpropagation is not straightforward.



The upstream gradients at any given time come from units who are themselves potentially receiving inputs (directly or indirectly) from the node we're trying to calculate the gradient of. Further, the activations at any given node for an input depends on time; for even an input $\mathbf{x}_t$ that is static across time, the activations $\mathbf{h}_t$ will not be static across time.

RNN training (cont.)

To get around this confound, we consider the RNN as a computational graph through time.



Importantly, each of the $\mathbf{h}_t$ are the activations of all the hidden units at a given time. If we expand the computational graph through time, then we effectively see that its a feedforward neural network, where the number of layers is the number of time steps.

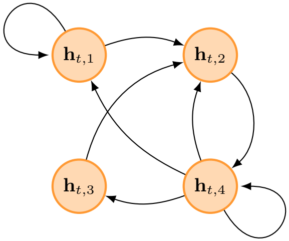
It is for this reason that Schmidhuber views RNNs as the deepest of neural networks (Schmidhuber, 2016), since they are effectively feedforward networks with depth given by time.

RNN training (cont.)

For example, consider a matrix $\mathbf{W}_{\text{rec}}$ that looks like the following:

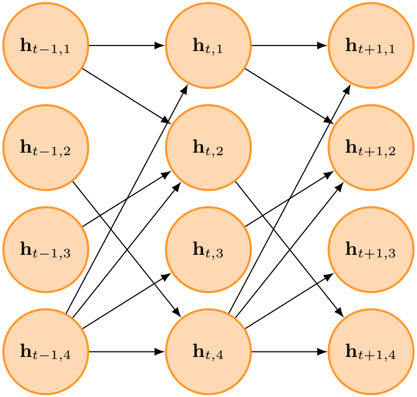
$$\mathbf{W}_{\text{rec}} = \begin{bmatrix} 1 & 0 & 0 & 0.4 \\ 0.9 & 0 & 0.6 & 0.5 \\ 0 & 0 & 0 & 0.3 \\ 0 & 0.8 & 0 & 0.7 \end{bmatrix}$$

This RNN looks like the following.



RNN training (cont.)

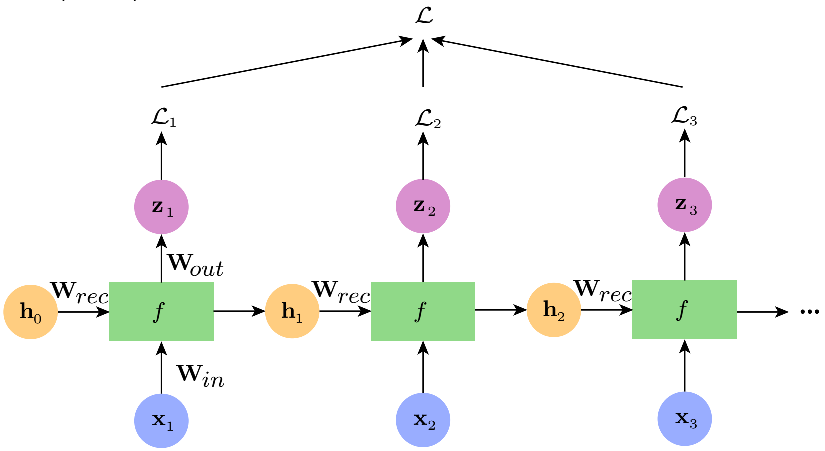
Let's take the following computational graph and unroll it.



At this point, the RNN becomes a feedforward network, with the number of layers being the number of time steps.

RNN training (cont.)

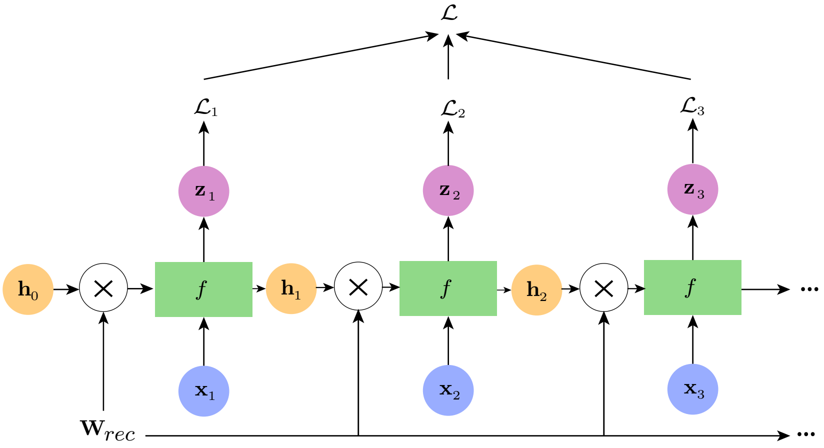
To train the network, we calculate gradients on the unrolled graph. Doing backpropagation through the unrolled graph is called *backpropagation through time* (BPTT).



Note that sometimes, one may only care about the last loss $\mathcal{L}_T$, in which case $L_t = 0$ for $t < T$.

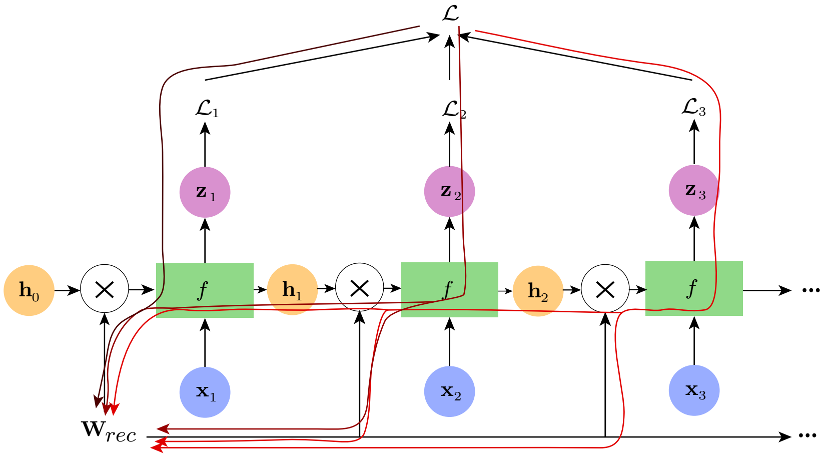
RNN training (cont.)

This graph can be redrawn as follows:



RNN training (cont.)

Redrawing the graph in this way, we see that to do backpropagation, there are several gradient paths to the parameters. Here, simply consider $\mathbf{W}_{\text{rec}}$. The red lines are all paths from the loss to $\mathbf{W}_{\text{rec}}$ through backpropagation.



Vanishing and exploding gradients

There are a few important considerations then for backpropagation through time. The first of these is vanishing and exploding gradients.

As the number of layers to backpropagate through is the length of your input sequence in time, training these networks is like training a deep network with a bit of gradient injected at every time step (through $\mathcal{L}_t$). With RNNs, the problem can be more precisely formulated.

Every backpropagation step from $\mathbf{h}_{t+1}$ to $\mathbf{h}_t$ will require backpropagating through the nonlinearity f, and then a matrix multiply with $\mathbf{W}_{\text{rec}}$. If our nonlinearity is ReLU and our upstream gradient is $\partial\mathcal{L}_t/\partial\mathbf{h}_{t-k}$, then backpropagating to $\mathbf{h}_{t-k-1}$ involves the following computation:

$$\frac{\partial\mathcal{L}_t}{\partial\mathbf{h}_{t-k-1}} = \mathbf{W}_{\text{rec}}^T \left(\mathbb{I}(\mathbf{h}_{t-k-1} \geq 0) \odot \partial\mathcal{L}_t/\partial\mathbf{h}_{t-k}\right)$$

And hence going back Δt layers in time requires repeated multiplication by $\mathbf{W}_{\text{rec}}^T$ (Δt times).

Vanishing and exploding gradients (cont.)

Let $\mathbf{W}_{\text{rec}}^T$ have eigenvalue decomposition $\mathbf{U}\Lambda\mathbf{U}^{-1}$. Then multiplication by by this matrix Δt times is equivalent to:

$$\begin{aligned}\left(\mathbf{W}_{\text{rec}}^T\right)^{\Delta t} &= \left(\mathbf{U}\Lambda\mathbf{U}^{-1}\right)^{\Delta t} \\ &= \mathbf{U}\Lambda\mathbf{U}^{-1}\mathbf{U}\Lambda\mathbf{U}^{-1}\cdots \\ &= \mathbf{U}\Lambda^{\Delta t}\mathbf{U}^{-1}\end{aligned}$$

Let λ_i be the ith eigenvalue of Λ. Then, along eigenvectors where $\lambda_i < 1$, the gradients will be attenuated to zero (vanishing gradients). Along dimensions where $\lambda_i > 1$, the gradients will grow exponentially (exploding gradients). These cause problems for gradient descent at earlier layers.

Addressing vanishing and exploding gradients

There are a few ways to address the problem of vanishing and exploding gradients. But first, a question:

Consider the gradient injected by $\mathcal{L}_1$. Doesn't this help to train the weights $\mathbf{W}_{\text{rec}}$, and if so, why should I be concerned that gradients from $\mathcal{L}_t$ are inaccurate? (e.g., think of the gradient injected by $\mathcal{L}_1$ as analogous to the auxiliary classifiers of GoogLeNet. Why doesn't this solve the problem of inaccurate gradients at earlier time steps?)

Addressing vanishing and exploding gradients (cont.)

A general way to address the vanishing and exploding gradients problem is to do *truncated* backpropagation through time. In truncated backpropagation through time, instead of backpropagating all gradients back to the activations at time 1, we only backpropagate them p layers through time.

Addressing vanishing and exploding gradients (cont.)

A general way to address the vanishing and exploding gradients problem is to do *truncated* backpropagation through time.

Exploding gradients. Like in CNNs, we can address the exploding gradients problem by constraining their norm to be a certain value. Let's say that the maximum gradient norm is a constant c. Then if the gradient, $\mathbf{g}$, is such that $\|\mathbf{g}\| \geq c$, then

$$\mathbf{g} \leftarrow \frac{c}{\|\mathbf{g}\|}\mathbf{g}$$

Vanishing gradients. Usually the way to handle vanishing gradients is via changing the architecture (later on in this lecture). However, there are ways to regularize RNNs to help ameliorate the vanishing gradients problem. Pascanu and colleagues (2012) propose the following regularization, which is added to the loss function:

$$\Omega = \sum_t \left(\frac{\left\| \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \right\|}{\left\| \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \right\|} - 1 \right)^2$$

Weight initialization strategies

In CNNs, we saw that the Xavier and He initializations can make a big difference in training neural networks. How should we initialize RNN's? The following strategies are for networks using ReLU's.

- Le et al., 2015, suggest an "initialization trick" of setting $\mathbf{W}_{\text{rec}} = \mathbf{I}$ and $\mathbf{b}_{\text{rec}} = \mathbf{0}$. The intuition for why this is good is that it preserves the gradients going back in time (at least to begin), i.e., its eigenvalues are all 1 so gradients at the start are not heavily attenuated or amplified.
- Talathi et al., 2016, hypothesize that an initialization where one eigenvalue is equal to 1 and the rest are less than 1 is better. Their initialization is as follows:
    - Sample a matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ whose values are drawn from $\mathcal{N}(0, 1)$, and N is the number of units in the RNN.
    - Compute $\mathbf{B} = \frac{1}{N}\mathbf{A}\mathbf{A}^T$ and let λ_{max} be the largest eigenvalue of the matrix $\mathbf{B} + \mathbf{I}$.
    - Initialize $\mathbf{W}_{\text{rec}} = \frac{1}{\lambda_{\text{max}}}\mathbf{B} + \mathbf{I}$.

  Empirically this is better than initializing $\mathbf{W}_{\text{rec}} = \mathbf{I}$.

The long short-term memory

The long short-term memory (LSTM) is a particular RNN architecture that is well-suited for addressing the problem of vanishing and exploding gradients. It was proposed by Hochreiter and Schmidhuber in 1997. It is one of the most commonly used RNN architectures today.

The name refers to its ability to store *short-term* memory for a *long* period of time. Hopefully the next few slides will help unpack what this really means.

LSTM

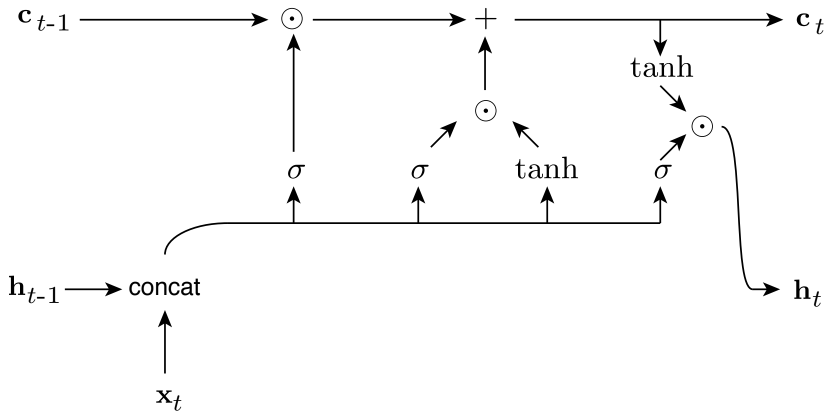
The standard LSTM is defined as follows:

$$\begin{aligned}
\mathbf{f}_t &= \sigma\left(\mathbf{W}_f \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_f\right) \\
(i)_t &= \sigma\left(\mathbf{W}_i \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_i\right) \\
\mathbf{v}_t &= \tanh\left(\mathbf{W}_v \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_v\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}_o \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_o\right) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + (i)_t \odot \mathbf{v}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}$$

where $\sigma(\cdot)$ is the sigmoid function. These functions at first glance are opaque. Here, we will unpack what they mean and how they help solve the vanishing and exploding gradients problem.

LSTM, block diagram

Here is a block diagram of the LSTM cell.



LSTM, cell state

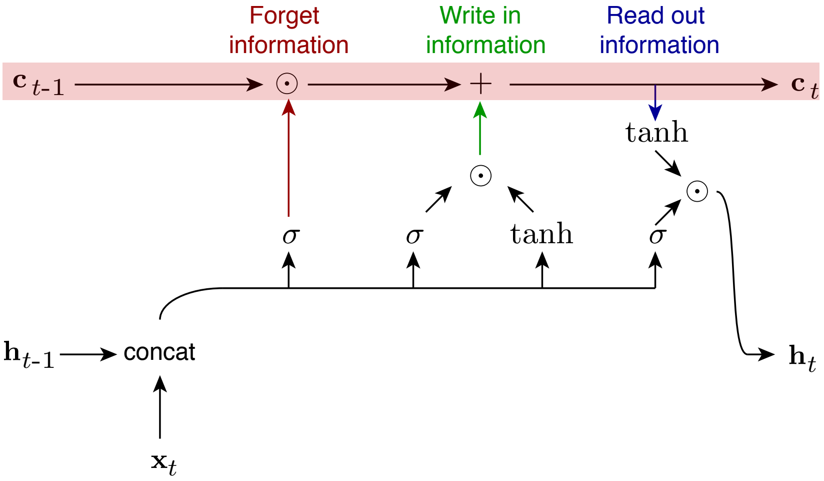
The LSTM cell state, $\mathbf{c}_t$, is the central component of the LSTM. We think of the cell state as some memory or tape; it holds some value and remembers it. But the key thing is that we can alter this cell state (i.e., we can alter the memory or tape). At each point in time, there are three things we can do to the cell state:

1. Forget information.
2. Write-in information.
3. Read-out information.

This is illustrated on the next page.

Note that the next hidden state, $\mathbf{h}_t$, is essentially a read out of the cell state $\mathbf{c}_t$, as $\mathbf{h}_t = f(\mathbf{c}_t)$, so the cell state contains all the vital information in the LSTM.

LSTM, cell state



Forgetting information, writing in information, and reading out information are all mediated by gates.

LSTM, forgetting information

To forget information, the LSTM uses the forget gate, $\mathbf{f}_t$. This comprises the update, $\mathbf{c}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$. Note:

- If $\mathbf{f}_t$ is close to 1, then $\mathbf{c}_t \approx \mathbf{c}_{t-1}$. Thus, when the forget gate is large, most information is remembered.
- If $\mathbf{f}_t$ is close to 0, then $\mathbf{c}_t \approx 0$. Thus, when the forget gate is small, most information is forgotten.

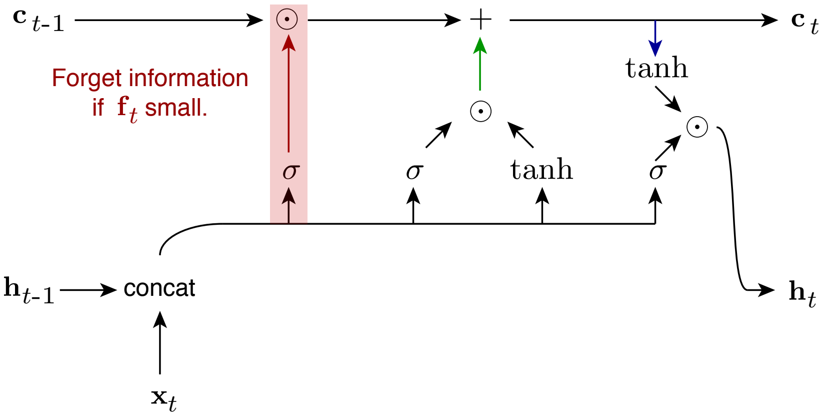
This gate is illustrated on the next page.

LSTM, forget gate

The network computes a linear transformation

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_f\right)$$

and if $\mathbf{f}_t$ is small, it forgets the information. If $\mathbf{f}_t$ is large, it remembers the information.



LSTM, writing in information

To write in information, the LSTM needs to compute two values. The following is not convention, but it helps me remember. We'll call these gates the value gate, $\mathbf{v}_t$, and the input gate, $(i)_t$. These are calculated as:

$$\begin{aligned} \mathbf{v}_t &= \tanh\left(\mathbf{W}_v \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_v\right) \\ (i)_t &= \sigma\left(\mathbf{W}_i \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_i\right) \end{aligned}$$

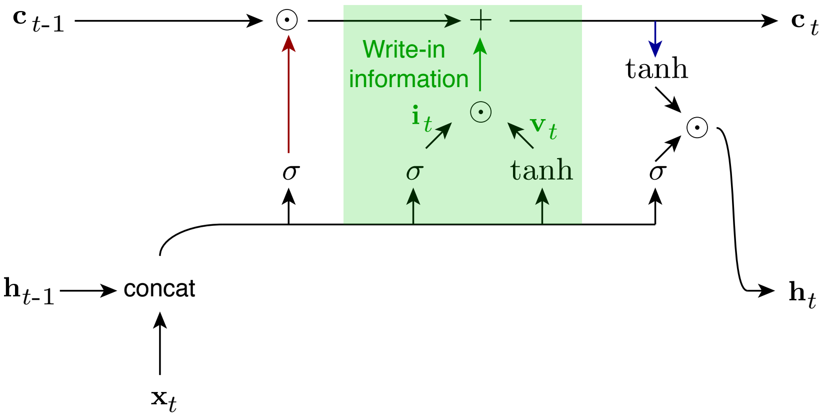
- The value gate, $\mathbf{v}_t \in (-1, 1)$ tells us the value we want to add to the cell state.
- The input gate, $(i)_t \in (0, 1)$ tells us how much of the value gate, $\mathbf{v}_t$ to write to the cell state.

Then, the total information we get to write in to the cell state is the multiplication of these two, i.e., this update looks like:

$$\mathbf{c}_t = \mathbf{c}_{t-1} + (i)_t \odot \mathbf{v}_t$$

If either $\mathbf{v}_t$ or $(i)_t$ is zero, we write no information to the cell state. If $(i)_t = 1$, we write whatever update information $\mathbf{v}_t$ into the cell.

LSTM, input gate



LSTM, reading out information

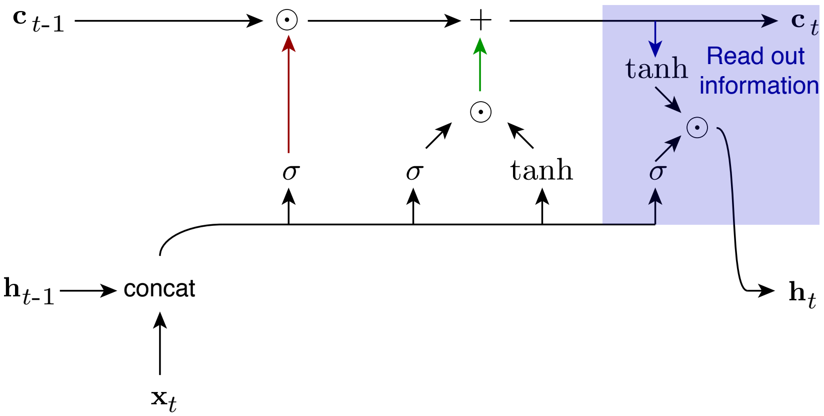
Now that we've updated the cell state by both forgetting information and writing in information, we are now ready to read out information to update the hidden state of the LSTM. This is fairly simple: we take our cell state, put it through the $\tanh$ nonlinearity, and then use our final gate, the output gate $\mathbf{o}_t$.

- The output gate, $\mathbf{o}_t$, tells us how much of the cell state is going to be read out to the hidden state.
- If $\mathbf{o}_t$ is small, very little will be read out.
- If $\mathbf{o}_t$ is large, a lot of the cell state will be read out.

Formally, the readout is:

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

LSTM, output gate



LSTM training

From this representation of the LSTM, we now can glean insight into why this works well.

- The cell state, $\mathbf{c}_t$, during backpropagation, has almost uninterrupted gradient flow. It's analogous to a gradient highway, much like in ResNets.
- In particular, the $+$ operation passes the gradient back.
- The gradient may be attenuated by the forget gate, $\mathbf{f}_t$. The gradient

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} \odot \mathbf{f}_t$$

  and therefore if the forget gate is small, then $\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t-1}}$ will be small, too.

LSTM, last comments

In practice, LSTMs are easier to train than vanilla RNNs using first order gradient descent techniques. However, note that the LSTM has (!) $4\times$ the number of parameters of a vanilla RNN.

To address this problem, another type of recurrent unit is used, called the **gated recurrent unit**.

Gated recurrent units (GRUs)

The standard GRU is defined as follows:

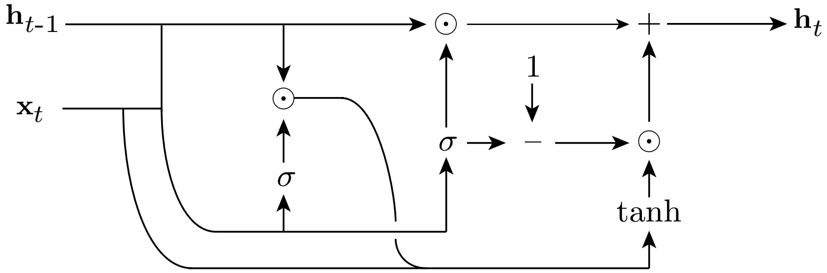
$$\begin{aligned}
\mathbf{r}_t &= \sigma\left(\mathbf{W}_r \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_r\right) \\
\mathbf{u}_t &= \sigma\left(\mathbf{W}_u \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_u\right) \\
\tilde{\mathbf{h}}_t &= \tanh\left(\mathbf{W}_h \begin{bmatrix} \mathbf{r}_t \odot \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_h\right) \\
\mathbf{h}_t &= \mathbf{u}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{u}_t) \odot \tilde{\mathbf{h}}_t
\end{aligned}$$

Again, it looks opaque at first, so let's unpack it. We call:

- $\mathbf{r}_t$ the reset gate.
- $\mathbf{u}_t$ the update gate.
- $\tilde{\mathbf{h}}_t$ the candidate activation.

GRU, block diagram

The GRU has the following block diagram:



Like the LSTM, we see that the hidden state essentially has a gradient highway for backpropagation.

GRU, update gate

The first thing we note about the GRU is that it has no cell state. So let's start with the update gate, $\mathbf{u}_t$. The update gate tells us how the hidden state updates.

- If the update gate $\mathbf{u}_t$ is close to 1, then the hidden state stays approximately constant: $\mathbf{h}_t \approx \mathbf{h}_{t-1}$.
- If the update gate $\mathbf{u}_t$ is close to 0, then the hidden state forgets its previous value and adopts the candidate activation, $\tilde{\mathbf{h}}_t$.

Hence, the GRU colloquially uses the hidden state as the "memory" as opposed to instantiating a new cell state. We can think of $\mathbf{u}_t$ being close to 1 as information not being forgotten, and $\mathbf{u}_t$ being close to 0 as information being forgotten and new information (i.e., the candidate activation) being written in.

GRU, reset gate

At this point, we're left with only one thing left to understand about the GRU – what is the new value $\tilde{\mathbf{h}}_t$ that may be written into the hidden state (if the update gate $\mathbf{u}_t$ is close to 0)?

This is mediated by the reset gate.

- If the reset gate $\mathbf{r}_t$ is close to 1, then the value $\tilde{\mathbf{h}}_t$ looks like a standard vanilla RNN update.
- If the reset gate $\mathbf{r}_t$ is close to 0, then the value $\tilde{\mathbf{h}}_t$ is completely set by the input.

The intuition is that if the reset gate $\mathbf{r}_t$ is close to 0, then it'll be as if the GRU was being reset, only looking at the input.

GRU, gradient flow

Much like the LSTM, the GRU is able to learn long term dependencies since there's a gradient highway on the hidden states $\mathbf{h}_t$. When it needs to remember long-term sequences, then the update gate will be close to (1), allowing the gradient to flow back in time through $\mathbf{h}_t$.

GRU, other comments

In this formulation of the GRU, it has $3\times$ the parameters of a vanilla RNN, which is less than the LSTM. (There is also a minimal GRU, which only has $2\times$ the parameters, and operates similarly to the GRU.) Compared to the LSTM:

- The GRU uses less memory and computation than the LSTM, since it doesn't maintain a cell state.
- GRUs have been empirically observed to train faster than LSTMs.
- LSTMs ought to be able to remember longer sequences by using a dedicated cell state.
- GRUs empirically tend to perform better than LSTMs, but this isn't always the case. In particular, in your application, it could be that a GRU does better than an LSTM, or vice versa. My recommendation is that you try both.

A final note on training RNNs

Since RNNs tend to have less units, second order methods may be more plausible. For example, Martens, Sutskever and Hinton (2011) reported a "Hessian-free" conjugate gradient method to optimize RNNs, and even a more expressive architecture called a multiplicative RNN. Here, second order methods helped substantially.

How to train?

In total, we have suggested four ways you might go about training recurrent neural networks.

- Use a vanilla RNN with gradient clipping (to ameliorate exploding gradients) and Pascanu's suggested regularization (to ameliorate vanishing gradients).
- Use an LSTM.
- Use a GRU.
- (Extension of 2 and 3: use some other gating unit variant (there are a bunch).)
- Consider using second order optimization methods.

In addition to this, it may be helpful to batch normalize the cell state of the LSTMs (e.g., Cooijmans et al., 2017).

Finally, it is possible to apply dropout to recurrent units. Refer to Semeniuta et al., 2016, as to how to do this without memory loss; in summary, with an LSTM, you should perform dropout on the value gate.