RNNs process sequential data, $x^1 \dots x^t$, can process variable size inputs, and scale well
In order to process data of different forms (in this case, length), parameters need to be shared
We have the same parameters across different time indices to share statistical strength among variable size inputs and across different positions in time
A traditional feedforward network will give weights to each different position, which may not be a good thing when the same thing can happen at different positions

Time delay CNNs use the same convolution kernel over the sequence at different timesteps
For RNNs, each member of the output is a function of the previous output members, and the same update rules are applied to the outputs

RNNs operate on a sequence containing vectors x^t w/ the time step t ranging from 1 to τ
RNNs can have backwards connections, provided the entire sequence is provided first

Cycles, in graph structure, let the present value of a variable affect its future value

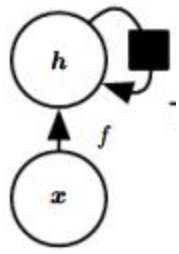
Unfolding the recurrent graph results in the sharing of parameters across a network structure
$s^{(t)} = f(s^{(t-1)}; \theta)$, where s^t is the state
This is recurrent because the definition of s at time t is the same at time t - 1
For a finite number of times τ, the graph can be unfolded by applying the definition τ - 1 times

It can be unfolded into a directed acyclic computational graph
Consider a dynamical system driven by an outside factor x
$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta)$,

Many RNNs use the following to define the values of hidden units, which represent state
$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$,
RNNs may add architectural features which make use of these hidden units

When the network tries to predict the future from the past, h^t is considered a lossy summary since it is constrained to keep a different amount of info than the original inputs

The folded model where a black square represents a delay of one time step is the easiest



Unfolded recurrence can be represented by a function g

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$$
$$= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

g is normally just a function which takes inputs from all time steps

Unfolded recurrent structure lets us factorize g^t into repeated functions of f

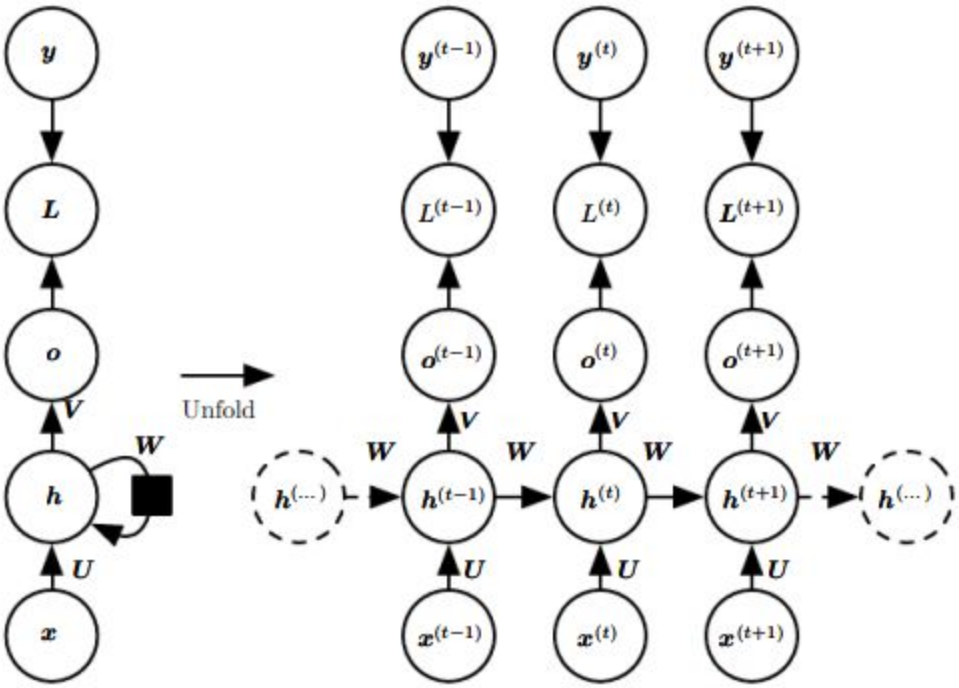
The learned model for unfolded recurrence always has the same input size, since it is not specified in terms of a variable length history of states, but rather as transition between 2 states

- Weight matrix must always be the same size in order to maintain the same amount of hidden units per layer

The same transition function f can be used w/ the same parameters at each time step

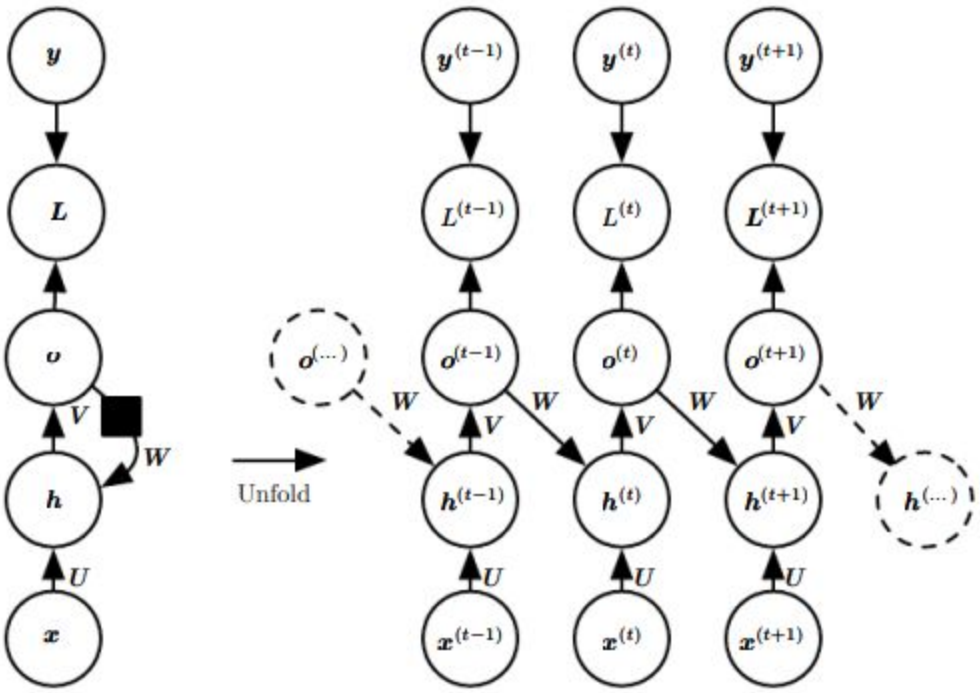
This lets us learn a function f which operates on all time steps, with parameter sharing, rather than using a separate model g^t for each time step; this lets us generalize to examples not in the training set, and allows to generalize to variable length sequence

Recurrent Neural Networks



Different types:

- Recurrent networks producing an output at each time step have recurrent connections between hidden units (figure above) <- this is a representative example we think of
- Some recurrent networks can have only a connection from the output to the hidden unit at the next time step (figure below)
- Some RNNs read in a whole sequence and then produce a single output



RNNs can compute all functions which are computable by a turing machine - outputs must be discretized to match how a turing function works (it takes in binary combinations)

Assume the hyperbolic tangent activation function is used
The output o can be regarded as giving unnormalized log probabilities, which can be softmaxed over
Forward prop begins with an initial hidden state h^0, and then the following steps are applied:

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}$$

Bias vectors are b and c; U, V, and W correspond to hidden-to-input, hidden-to-output, and hidden-to-hidden, respectively
The output sequence is the same length as the input sequence

Total loss for a sequence of x values paired with y values would be given by:

$$L\left(\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)}\}\right)$$
$$= \sum_t L^{(t)}$$
$$= -\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\right),$$

if L is the negative log likelihood of y given x^1 .. x^t where p_{model} is given by reading the entry for y^t from the models output y-hat
Gradient computation is expensive; it is O($\boldsymbol{\tau}$) and cannot be reduced by parallelization since the graph is sequential
Memory cost is O($\boldsymbol{\tau}$) since states from the forward pass must be saved
Backprop applied to an unfolded graph is called backpropagation through time

Recurrent connections from only the output to the hidden units at the next step is weaker since it lacks hidden-to-hidden recurrent connections
Output units must capture all of the information of the past that the network needs to predict future

User must describe the full state as a training target since the output would otherwise be unlikely to contain enough information
Eliminating hidden-to-hidden recurrence lets gradient computation be parallelized for time steps
The training set provides the ideal value of the previous output, so there is no need to compute the output of the previous time step first

Models with recurrent connections with outputs leading back to the model can be trained with teacher forcing, where the model receives the ground truth y^t at time t + 1 (not the previous output)
The conditional max likelihood criterion is

$$\log p\left(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)$$
$$= \log p\left(\boldsymbol{y}^{(2)} \mid \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) + \log p\left(\boldsymbol{y}^{(1)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)$$

Model must maximize the conditional probability of y^2 given both the x sequence and the previous output
Connections should be fed with targets specifying what the correct output should be, rather than feeding the output to itself

Some models may be trained with both BPTT and teacher forcing; as soon as the hidden units become a function of earlier time steps, BPTT is necessary

If the network is later going to be used in open-loop mode, with the network outputs (after all time steps) fed back as inputs, this can be problematic since the inputs that the network receives during training will be different than those it sees during test time

This can be mitigated by training with both teacher forced inputs and free-running inputs (e.g predicting the correct target a number of steps in the future through the unfolded recurrent path)
The network can learn to take into account input conditions not seen during training (the ones it generates itself from the previous step) and map the state to a viable future output
Alternatively, you can randomly choose actual data values or generated values as input, which exploits a curriculum learning strategy to gradually use more generated values as inputs

Use a backprop gradient algorithm on the unfolded computational graph to compute grads
Compute the gradient recursively

$$\frac{\partial L}{\partial L^{(t)}} = 1.$$

Softmax applied to o to receive y-hat

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}.$$

Indexing a particular element of the grad to specify its derivative; note the gradient is its normalized probability minus 1
Loss is negative log likelihood

At the final timestep, h^{τ} only has $sigma^{tau}$ as its descendant, so the gradient is simple:

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^\top \nabla_{\boldsymbol{o}^{(\tau)}} L.$$

V is applied to h^t to obtain o
h^t for $t < \boldsymbol{\tau}$ has descendants $sigma^t$ and h^{t+1}

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

diag in the above equation represents the diagonal matrix containing $1 - (h_i^{(t+1)})^2$, which is the Jacobian of the hyperbolic tangent associated w/ unit i at time t + 1

Once the gradients of the internal nodes are obtained, we can obtain the gradients of the parameter nodes

The $\nabla_{\boldsymbol{W}} f$ operator takes into account the gradient w/ respect to all edges in the graph; to fix this, we use dummy variables W^t which are copies of W used at an individual timestep
The gradient is then given by

$$\nabla_{\boldsymbol{c}} L = \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}} \right)^\top \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L$$

$$\nabla_{\boldsymbol{b}} L = \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}} \right)^\top \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)} \right)^2 \right) \nabla_{\boldsymbol{h}^{(t)}} L$$

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L \right) \boldsymbol{h}^{(t)\top}$$

$$\nabla_{\boldsymbol{W}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)} \right)^2 \right) \left(\nabla_{\boldsymbol{h}^{(t)}} L \right) \boldsymbol{h}^{(t-1)\top}$$

$$\nabla_{\boldsymbol{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)} \right)^2 \right) \left(\nabla_{\boldsymbol{h}^{(t)}} L \right) \boldsymbol{x}^{(t)\top}$$

x^t does not have any parameters as ancestors in the computational graph defining its loss

Usually use the xentropy of the output distribution to define the loss
MSE is the xentropy loss associated w/ an output distribution which is a unit Gaussian
We train the RNN to estimate the conditional distribution of the next sequence element y^t given the past inputs $x^1 \dots x^t$, $\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)})$
or if there are output connections between steps,
$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(t-1)})$.
Decompose the joint probability over the sequence of y values as a series of one-step probabilistic predictions can capture the entire sequence's joint distribution
If there are no connections from previous outputs, the outputs y are conditionally independent given the sequence of x values
If and when we feed the true y values back into the network, the directed graph model contains edges from y^i values in the past to the current y^t value

Consider the RNN models a sequence of random scalar variables $Y = \{y^1 .. y^\tau\}$ w/o additional inputs x
The input at time step t is the output from t - 1, defining a directed graphical model over the y variables

This joint distribution can be parametrized using the chain rule for conditional probabilities

$$P(\mathbb{Y}) = P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\mathbf{y}^{(t)} \mid \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \ldots, \mathbf{y}^{(1)})$$

The negative log likelihood of a set of values is then

$$L = \sum_t L^{(t)} \quad \text{where} \quad L^{(t)} = -\log P(\mathrm{y}^{(t)} = y^{(t)} \mid y^{(t-1)}, y^{(t-2)}, \ldots, y^{(1)}).$$

Since edges correspond to direct interactions between variables, many graphs become more computationally efficient by eliminating weak interactions
It is common to make the Markov assumption that only the $\{y^k \ldots y^{t-1}\}$ inputs are important, rather than all the previous ones
RNNs are good for when an input from a long time ago y^i affects y^t, but in a way not captured by y^{t-1}

An RNN can be viewed as simply edges between each of the inputs (e.g from y^1 to y^k)
Regarding h^t as random variables, the RNN provides a very efficient parametrization of the joint distribution over the observations

Suppose we represent a joint distribution over discrete values with a tabular representation - an array containing a separate entry for each possible assignment of values, with the entry giving the probability of that assignment
It would need to have $O(k^t)$ parameters
The number of parameters in the RNN is O(1) as a function of sequence length, and may be adjusted
RNN uses recurrent applications of the same function f and same params $\boldsymbol{\Theta}$ at each time step

Incorporating h^t decouples the past and future, acting as an intermediate quantity
The model can be efficiently parametrized by using the same conditional probability distributions at each time step, and when all the variables are observed, the probability of the joint assignment of all variables can be evaluated efficiently

It is difficult to predict missing values in the middle of the sequence
The price for reduced number of params in an RNN is harder optimization

Parameter sharing relies on the assumption the same params can be used at different steps, which is equivalent to saying the conditional probability distribution at time t + 1 given the vars at time t is stationary (meaning the relationship does not depend on time)
It is possible to use t as an additional input to t + 1, which is better than using a different conditional probability distribution for each t, but the network would have to extrapolate for new t

The main operation we need is to sample the conditional distribution at each time step

The RNN must have some mechanism for determining the length of the sequence
When the output is taken from vocabulary, you can add a special symbol corresponding to the end of a sequence; in the training step, we insert this sequence at the end of x^{τ}

Another option is to introduce a new Bernoulli output which uses the sigmoid unit to determine whether to continue or stop generation; this output is trained based on whether it correctly predicted whether or not to continue

Another way is to add an output which predicts τ itself, sampling τ steps worth of data
This requires adding an extra input, τ or τ - t at each time step, so the recurrent update is aware of whether it is near the end of the generated timestep
Without this, the RNN might end abruptly, since it won't know when it's reaching near the end
This approach is based on the decomposition

$$P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}) = P(\tau) P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)} \mid \tau).$$

Graphical model of a conditional distribution of y given x
We can represent a conditional distribution p(y|w) with w = $\boldsymbol{\Theta}$; if we make w a function of x, we can extend it using a similar function P(y|w)

We can make a single vector x as an extra input generating the y sequence
This can be done by providing it as an extra input at each time step, making it the initial state h^0, or doing both
In the first method, the product $x^T R$ is an additional input to the hidden unit - the choice of x influences $x^T R$, which can be considered a bias parameter of each hidden unit
The weights are independent of the input; taking parameters $\boldsymbol{\Theta}$ of a nonconditional model, and turning them into w, where the biases within w are functions of the input

The RNN may receive a sequence of vectors x^t as the input
$P(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)})$ corresponds to a conditional independence assumption
meaning the distribution factorizes as $\prod_t P(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)})$.
Adding connections from the output at time t to the hidden unit at t + 1 removes the conditional independence assumption, which lets the model represent arbitrary distributions over y
This model representing a sequence given another sequence is restricted by the condition that the length of both sequences (x and y) must be the same

Bidirectional RNNs
Sometimes we have to look at the whole input sequence (the past and the future) such as in handwriting recognition and speech recognition
Bidirectional RNNs combine an RNN which moves forward from the beginning of time and an RNN which moves backward from the end of time

h^t represents the forward state and g^t represents the backward state
Output units o^t can compute a representation depending on both the past and future, but one which is most sensitive to the input values around t
2D inputs, such as images, can be iterated over by 4 RNNs, which can capture relationships from across the feature maps
Forward prop for recurrent networks is similar to using a bottom-up conv layer before capturing features from across the map

Encoder-Decoder Sequence-to-Sequence Architectures
An RNN can map a fixed size vector to a sequence, an input sequence to a fixed size vector, and an input sequence to an output sequence of the same length
An RNN can be trained to map an input sequence to an output sequence which is not the same length
The input is called the context C, and we want to produce a representation of this context C - this might be a vector or sequence of vectors $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n_x)})$.

Encoder-decoder sequence, or sequence-to-sequence architeture, maps a variable length sequence to another variable length sequence
For an encoder-decoder sequence: encoder emits the context C as a simple function of its final hidden state, while a decoder RNN is conditioned on the fixed length vector to generate an output sequence
$\boldsymbol{Y} = (\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n_y)})$
The lengths can vary so that n^x =/= n^y =/= $\boldsymbol{\tau}$

- Sentence for example - 10 words as an input, 5 words as an output, which was not possible before

Sequence-to-sequence: two RNNs jointly maximize the average of $\log P(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n_y)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n_x)})$ over all pairs of x and y sequences in the training state -> h_{nx} , the last state of the encoder, is used as a representation of C to go into a decoder

If the context C is a vector, then the decoder is a vector-to-sequence RNN where the input can be provided as the initial state of an RNN or can be connected to the hidden units at each step
The encoder does not need to have the same size of hidden layer as the decoder

- Encoder can have more neurons, different number of steps, etc.

One problem can be when the context C output by the encoder RNN has a dimension which is too small to summarize a long sequence
C could be a variable length sequence w/ an attention mechanism which associates elements of C with elements of the output sequence

Deep Recurrent Networks

RNNs have three blocks of associated parameters and transformations: input to hidden state, previous hidden state to next hidden state, and hidden state to output
Each of these 3 blocks is associated w/ its own weight matrix - when the network is unfolded, each of these corresponds to a shallow transformation (one which can be represented as a layer in an MLP)

Fig 10.13
It seems advantageous to introduce depth into each of these operations
Decomposing the state of an RNN into multiple layers - lower hidden state h and upper hidden state z, in a similar way to a convnet
It's also possible to add extra depth for each of the 3 blocks; this makes the shortest path between adjacent timesteps longer, and could make it harder to optimize the network
This can be mitigated by using skip connections in the hidden-to-hidden path

Recursive Neural Nets
Fig 10.14
Recursive nets are structured as a deep tree, rather than a chained graph

Recursive nets have been successful in processing data structures as inputs to neural nets in NLP and computer vision
The depth can be reduced from $O(\tau)$ to $O(\log(\tau))$ which can help with long-term dependencies

One way to structure a tree is to not make it depend on the data - e.g structuring it as a balanced binary tree
Another way is to structure it in a way to match the data - for example, processing a sentence, structure the tree so it matches the sentence parsing structure
Ideally, the model will learn the structure itself

Variants are possible - e.g associate data w/ a tree structure, inputs and targets are individual nodes of the tree, and the computation does not have to be a typical linear transformation w/ a nonlinearity after it

Challenge of Long Term Dependencies
Vanishing and exploding gradients are a problem w/ RNNs
Even if we assume the weights are stable, exponentially smaller weights are given to long term interactions compared to short ones, causing weak long term dependencies

Figure 10.15
Recurrent networks, which involve the composition of the same functions, multiple times in one step, result in nonlinear behavior
The result is highly nonlinear after multiple compositions, w/ extremely small or large derivatives

The recurrence relation can be given by $\boldsymbol{h}^{(t)} = \boldsymbol{W}^\top \boldsymbol{h}^{(t-1)}$ as a simple network lacking inputs and a nonlinear activation function

This describes a power relation and can be simplified to $\boldsymbol{h}^{(t)} = \left(\boldsymbol{W}^t\right)^\top \boldsymbol{h}^{(0)}$ if W can be eigendecomposed in the form $\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$
With an orthogonal Q (orthonormal rows / cols, etc.) it can be simplified to further $\boldsymbol{h}^{(t)} = \boldsymbol{Q}^\top \boldsymbol{\Lambda}^t \boldsymbol{Q}\boldsymbol{h}^{(0)}$.
Eigenvalues are raised to the power of t causing those w/ magnitude < 1 to decay to 0, and those w/ magnitude > 1 to explode
Components of h^0 not aligned w/ the largest eigenvector will be discarded

To obtain a desired variance in a feedforward net, we choose the individual weights w/ variance $v = \sqrt[n]{v^*}$, since the variance of the products is $O(v^n)$
Deep feedforward network with carefully chosen scaling can avoid vanishing and exploding gradients

The RNN must enter a region of param space where gradients vanish in order to resist small perturbations
Signals of long term dependencies tend to be overshadowed by short term dependencies, since the long term interaction gradients have a much smaller magnitude

Echo State Networks
Recurrent weights, defined as mapping from h^{t-1} to h^t, and input weights mapping from x^t to h^t are some of the most dififcult parameters to learn in a recurrent network
Set the recurrent weights (hidden-to-hidden) such that the recurrent hidden units do a good job of capturing past inputs and then learn only the output weights
This is known as reservoir computing - the hidden units form of reservoir of temporal features capture different aspects of the history of inputs

Reservoir computing networks are similar to kernels - they map an arbitrary length seqence to a fixed size vector on which a linear predictor can be applied to solve a problem
The training criterion may be designed to be convex as a function of the output weights (e.g linreg with MSE as loss)

To choose input and recurrent weights, view the recurrent net as a dynamic system and set the input and recurrent weights such that the dynamic system is near the edge of its stability
Eigenvalues of the Jacobian of state-to-state transition function should be close to 1

Eigenvalue spectrum of the Jacobians $\boldsymbol{J}^{(t)} = \frac{\partial s^{(t)}}{\partial s^{(t-1)}}$.
The spectral radius is the maximum of the absolute value of the eigenvalues

Consider the case of backprop where J does not change with t
After n steps of backprop, we have $J^n g$
Backpropagating a perturbed version of g, g + dv, after n steps, the divergence between g and g + dv is $\delta J^n v$
If v is a unit eigenvector of J with eigenvalue lambda, then multiplication by the Jacobian simply scales the difference at each step
When v corresponds to the largest value of lambda, this perturbation caused by d achieves the largest possible separation

When |lambda| > 1, grows exponentially large; if lambda < 1, it grows exponentially small
This assumes the Jacobian is the same at every timestep, corresponding to a network w/ no nonlinearity -> networks with nonlinearities have derivatives at 0 at many timesteps, preventing an explosion

Everything said about backprop applies equally to forward prop w/ a simple model: $h^{t+1} = (h^t)^T W$
When a linear map W^T always shrinks h, measured by L^2 norm, the map is contractive
When the spectral radius < 1, the mapping from h^t to h^{t+1} is contractive, causing the network to forget info about the past when using a finite level of precision (e.g 32 bit ints)

Jacobian tells us how a small change of h^t propagates one step forward (or h^{t+1} backwards)
W nor J need to be symmetric (just square and real) so they can have complex-valued eigenvalues, with imaginary components corresponding to oscillating behavior

h^t can be expressed in terms of such a complex-valued basis
What matter is what happens to the complex absolute value / magnitude of these basis coefficients when multiplying the matrix by a vector
An eigenvalue w/ magnitude > 1 corresponds to magnification or shrinking

A small initial variation can turn into a large one, even with a nonlinear map
A squashing nonlinearity, e.g tanh, can cause recurrent dynamics to become bounded
Backprop can retain unbounded dynamics even when recurrent ones are bounded - for example, if a sequence of tanh units are connected and in the middle of their linear activation

Echo state networks aim to fix the spectral radius at some value such that info is carried over across time but does not explode due to saturating activations

Many techniques used to set the weights in ESNs could be used to initialize weights in a *fully* trainable recurrent network

Leaky Units and Other Strategies for Multiple Time Scales
One way to deal w/ long term dependencies is to develop a model with parts which deals with short, precise details while other parts deal w/ distant past

Skip connections - direct connections from variables in the distant past to the present (delay)

Gradients may vanish or explode exponentially w/ respect to the # of timesteps
Recurrent connections with a delay d cause gradients to diminish exponentially as a function of τ / d rather than τ, causing better recognition of long term dependencies

Obtaining paths on which the product of the derivative is close to 1 prevents explosion
Another way to obtain these paths is to have units w/ linear self connections w/ wts nearby

$\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1-\alpha)v^{(t}$ When α is near one, the running average remembers info from the distant past
Hidden units w/ linear self connections behave similarly to such averages - these are leaky units

The use of a linear self connection with a weight near one is a way of ensuring that the unit can access values from the past - it prevents the need to set an explicit d, as used in skip

Setting the time constant can be done by manually fixing them to constant values (e.g from some distribution) or by learning them as parameters

Organize the state of the RNN at multiple time scales, with info flowing more easily from long distance at slower time scale
Actively remove length-one connections and replace them w/ longer connections (rather than just adding them on), forcing them to operate on a long time scale
Units receiving this effect may choose to operate on a short time scale

One way is to make recurrent units leaky but have different groups of units associated w/ different fixed time scales
Another way is to have explicit updates taking place at different times with a different frequency for different groups of units

LSTM and Gated RNNS
Gated RNNs include long short term memory and other units based on gated recurrent unit
Gated RNNs prevent gradient vanish / explosion and generalize the idea behind leaky units by choosing connection weights which may change at each time step

Leaky units allow the network to accumulate useful information over a long duration - once that info has been used, it might be beneficial for the neural net to forget the old state
Gated RNNs learn when to forget the state (e.g it learns if we need to reset it at every n-th timestep)

LSTM introduces self-loops to produce paths where the gradient can flow for long durations

Making the weight on this self-loop conditioned by context, instead of fixed, is beneficial

By making the weight of this self loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically
The time scale of integration can change based on the input sequence, even for a fixed LSTM, since the time constants are output by the model itself

Fig 10.16
Cells are recurrently connected to one another, replacing hidden units
Input feature computed with a regular artificial neuron unit
If the sigmoidal input gate allows it, its value can be accumulated into the state
The state has a linear self-loop with a weight controlled by the forget gate
The output of a cell can be shut off by the output gate
Gating units have a sigmoid nonlinearity, while the input unit has a squashing nonlinearity
State unit can be provided as extra input to the gates

LSTM cells have an internal recurrence (self-loop), followed by an outer RNN recurrence
Each cell has the same inputs and outputs, but more parameters and gating units
The state unit s_i^t which has a linear self-loop similar to a leaky unit has a weight controlled by the forget gate which sets it to a value between 0 and 1 using

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$$

x is the input vector, h^t is the current hidden layer vector, b, U, and W are biases, input weights, and recurrent weights respectively (if provided as extra input) for the forget gate

Internal state is updated as below, with a conditional self-loop weight

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$$

The external input gate unit g_i^t is computed similarly to the forget gate but with its own params

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$$

The output gate q_i^t can shut off the output h of an LSTM cell; it uses a sigmoid unit as well

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$$

Choosing to use the cell state as extra input, with its weight, into the 3 gates, requires 3 extra params

The special thing about an LSTM is that a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit
The update equation is given by:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right),$$

where u is the update gate and r is the reset gate

$$u_i^{(t)} = \sigma \left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)} \right)$$

$$r_i^{(t)} = \sigma \left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)} \right)$$

The reset gates can ignore parts of the state vector
Update gates are like leaky conditional integrators that can linearly gate any dimension, choosing to copy it (at one end of the sigmoid) or completely ignoring it
Reset gates control which parts of the state get used to compute the next target state, introducing additional nonlinearity

Many architectural variants can be designed - reset gate could be shared across multiple hidden units
Alternatively, the product of a global gate (covering an entire group of units, e.g per layer) and a local gate, per unit, can be used to combine global and local control
Adding a bias of 1 to the LSTM gate makes it as strong as any other discovered architecture

Optimization for Long Term Dependencies
Second derivatives may vanish at the same time first derivatives vanish
Second order optimization algorithms divide the first derivative by the second derivative (or multiplying the gradient by the inverse Hessian)
The ratio will remain relatively constant if the first and second derivatives shrink at similar rates
Second order methods are hard to optimize due to necessary batch size and other factors

Strongly nonilinear functions (e.g computed over many timesteps by an RNN) have derivatives very small or very large in magnitude -> flat regions separated by cliffs
The gradient tells us the direction that corresponds to steepest descent within an infinitesimal region surrounding the current parameters; outside this region, the cost function may ascend

Learning rates with small decays have approx. same LRS during consecutive steps

A step size which is good for a linear part is not good for a curved part, which may come directly after the linear part

There are multiple ways to clip a gradient:

- Clip the parameter gradient element-wise just before the update
- Clip the norm of the gradient just before the param. update
- $$\text{if } ||\boldsymbol{g}|| > v$$
$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}v}{||\boldsymbol{g}||}$$

The latter ensures that the direction is maintained, but experimentally, both methods are good
Clipping the gradient norm per minibatch does not change the direction, but averaging over many minibatches is not equivalent to the epoch gradient

Creating paths in the computational graph of the unfolded architecture along which the gradient product is near 1 can address vanishing gradients
Besides using an LSTM, we can regularize parameters to encourage information flow
We would like the gradient vector being backpropagated to maintain its magnitude

$(\nabla_{\boldsymbol{h}^{(t)}} L) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$ should be equal in magnitude to $\nabla_{\boldsymbol{h}^{(t)}} L.$

The following regularizer is good

$$\Omega = \sum_t \left(\frac{\left| |(\nabla_{\boldsymbol{h}^{(t)}} L) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}| \right|}{||\nabla_{\boldsymbol{h}^{(t)}} L||} - 1 \right)^2.$$

This is computationally efficient if the backpropagated vectors are treated as constant in order to achieve a certain approximation
This regularizer works well with norm clipping, and combined they can learn many dependencies needed in an RNN
Gradient clipping is needed to prevent gradient explosion from stopping learning in this case

Explicit Knowledge
Neural networks excel at storing implicit knowledge, but are not good at memorizing facts
This is because neural nets lack a working memory, which humans use to manipulate info which are needed to solve a particular goal (e.g using a fact to reason)
Memory networks include a set of cells that can be accessed through an addressing mechanism

The neural turing machine learns to read and write arbitrary content to memory cells without explicit supervision, and is trained through a soft attention mechanism

The network outputs an internal state which chooses which cell to read from or write to, similar to how memory accesses a digital computer

Optimizing functions which produce exact integer addresses is hard
To mitigate this, NTMs take a weighted average when they read and modify cells by different amounts; the coefficients are chosen to be focused on a small number of cells
Weights w/ nonzero derivatives allow the function controlling memory access to be optimized using gradient descent -> gradient is typically only large for memory addresses receiving a large coefficient / weight

Memory cells are typically augmented to contain a vector instead of a single scalar, in order to offset the computational cost in producing viable coefficients for only a few addresses
Vector-valued memory cells allow for content based addressing where the weight used to read or write to a cell is a function of the cell
This allows us to retrieve a complete vector-valued memory if we are able to produce a pattern which matches some of the elements of the vector - this is similar to remembering the lyrics of a song based on the lyrics that come before it
Location based addressing cannot refer to the content of the memory - it is good when memory cells are small

If the content of a memory cell is copied, its information keeps flowing, allowing gradients to be propagated far through time

Instead of taking a weighted average over the memory cells, we can interpret the memory coefficients as addressing probability distributions and use the probability distribution to randomly choose which cell to access

In machine translation amd memory networks, the focus of the attention mechanism can move all over the place during consecutive timesteps