Natural Language Processing
CS 6320

Lecture 8

Sequence Processing with Recurrent Networks

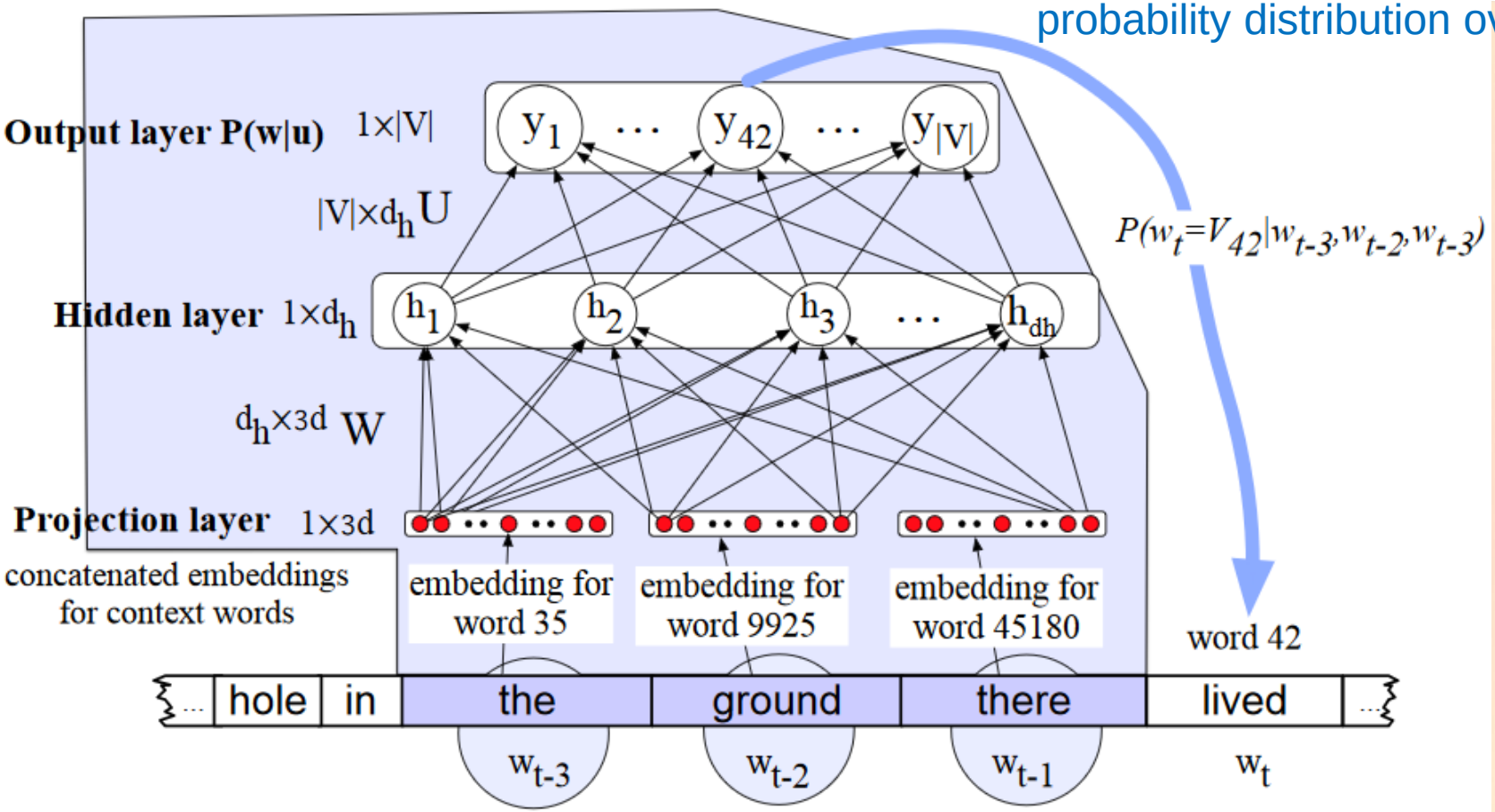
Instructor: Sanda Harabagiu

Today

- *We have explored feedforward neural networks along with their applications to neural language models.*
  - *We saw that feedforward networks can be trained to make predictions about the next word in a sequence, given a limited context of preceding words — an approach that is reminiscent of the Markov approach to language modeling.*
  - *These models operated by accepting a small fixed-sized window of tokens as input;*
  - *Longer sequences are processed by sliding this window over the input making incremental predictions, with the end result being a sequence of predictions spanning the input.*

Simplified View

The output is a *softmax* with a probability distribution over words.



A simplified FFNNLM with N=3; we have **a moving window** at time *t* with an embedding vector representing each of the 3 previous words (w_{t-1}, w_{t-2}, and w_{t-3}). These 3 vectors are concatenated together to produce *x*, the input layer.

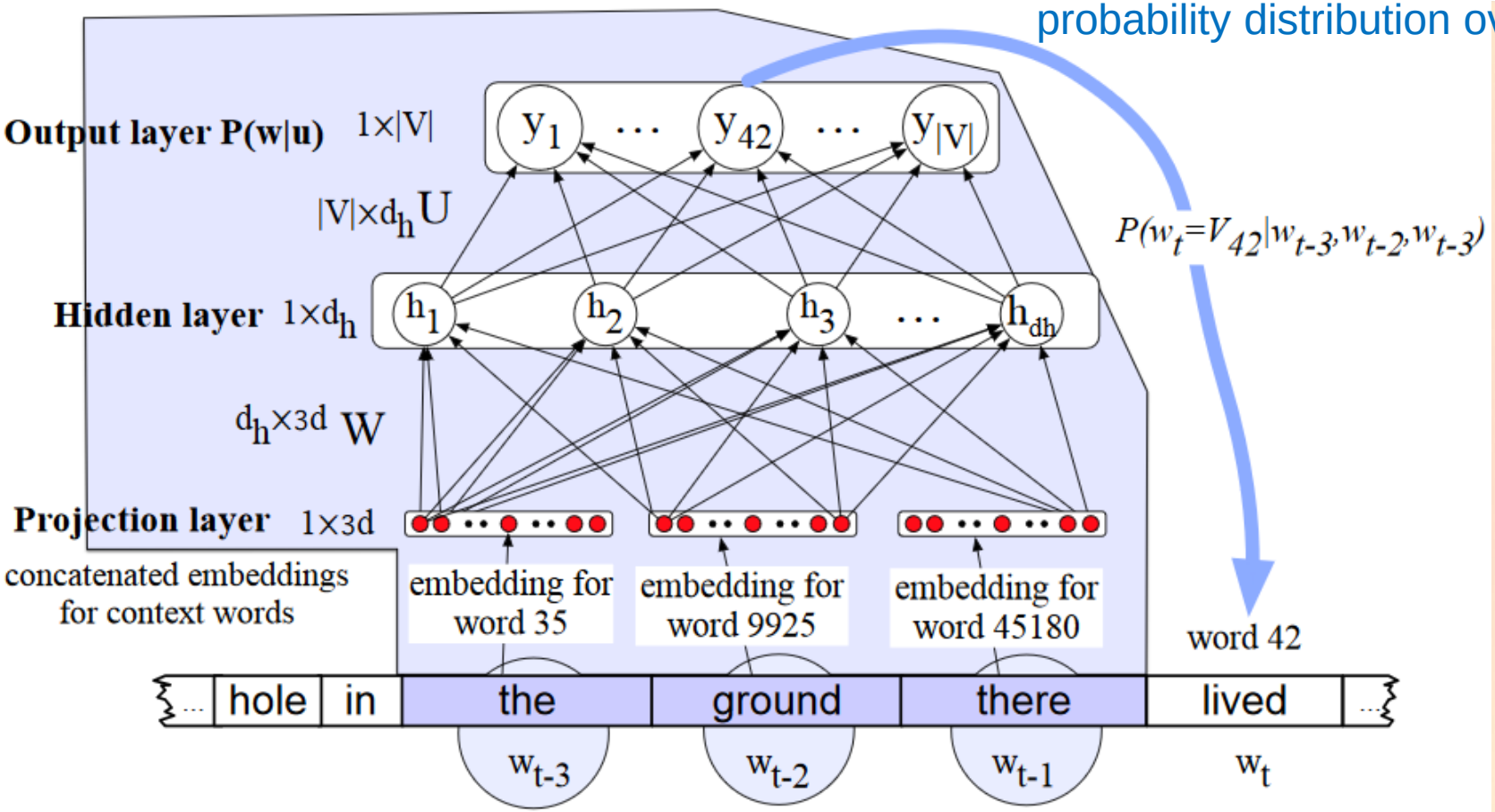
The problems

Unfortunately, the sliding window approach is problematic for a number of reasons:

1. *First, it shares the primary weakness of Markov approaches in that it limits the context from which information can be extracted; anything outside the context window has no impact on the decision being made. This is problematic since there are many language tasks that require access to information that can be arbitrarily distant from the point at which processing is happening.*
2. *Second, the use of windows makes it difficult for networks to learn systematic patterns arising from phenomena like constituency*

Example

The output is a *softmax* with a probability distribution over words.

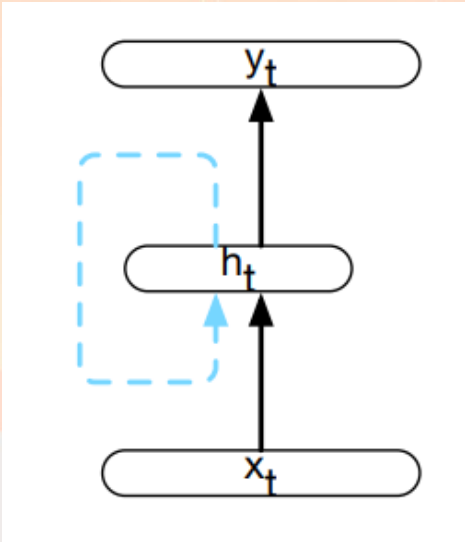


The phrase the ground appears twice in different windows: once, as shown, in the first and second positions in the window, and in in the preceding step in the second and third slots, thus forcing the network to learn two separate patterns for a single constituent!

Simple Recurrent Networks

- *The solution!!!*

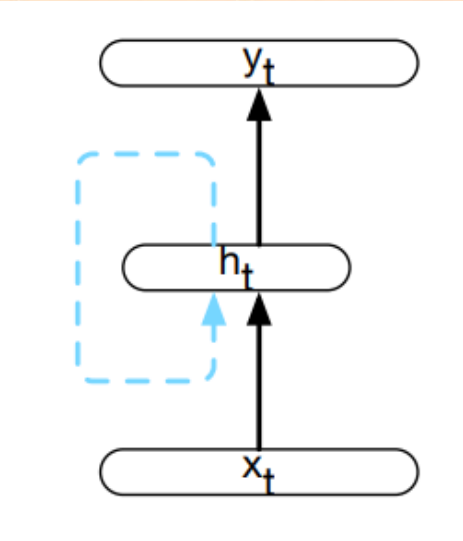
➢ *A recurrent neural network is any network that contains is **a cycle** within its network connections. That is, any network where the value of a unit is directly, or indirectly, dependent on its own output as an input.*



➢ *have proven to be extremely useful when applied to language problems!*

Simple Recurrent Networks (SRNs)

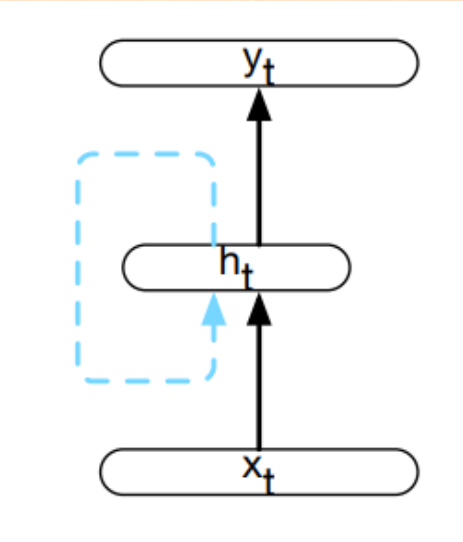
A.k.a ***Elman networks****.*



- *The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on:*
  - *the current input as well as*
  - *the activation value of the hidden layer from the previous timestep.*

An input vector representing the current input element, x_t , is multiplied by a weight matrix and then passed through *an activation function* to compute an activation value for a layer of hidden of units.
This hidden layer is, in turn, used to calculate a corresponding output, y_t .

Key Difference FFN vs. RNN

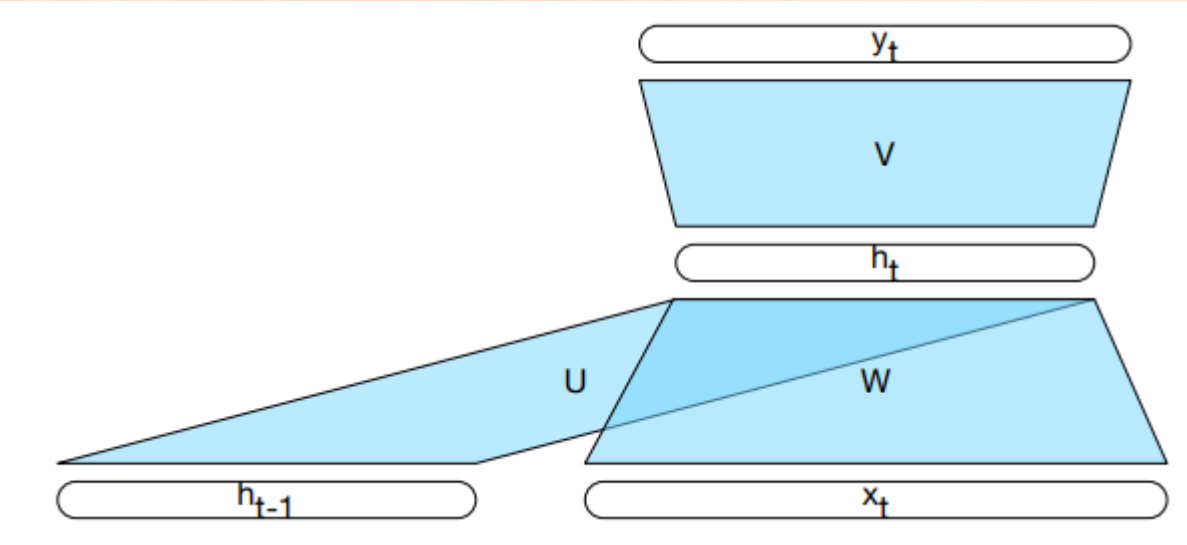


- *The key difference from a feedforward network (FFN) lies in the recurrent link shown in the figure with the dashed line.*
- *This link augments the input to the hidden layer with the activation value of the hidden layer from the preceding point in time.*

The hidden layer from the previous timestep provides a form of **memory**, or **context**, that encodes earlier processing and informs the decisions to be made at later points in time.

- The RNN architecture does not impose a fixed-length limit on prior context;
  - the context embodied in the previous hidden layer includes information extending back to the beginning of the sequence.

Clarifying the nature of recurrence



- *The most significant addition lies in the new set of weights, U, that connect the hidden layer from the previous timestep to the current hidden layer. These weights determine how the network should make use of past context in calculating the output for the current input.*
- *As with the other weights in the network, these connections will be trained via backpropagation*

Inference in RNNs (part 1)

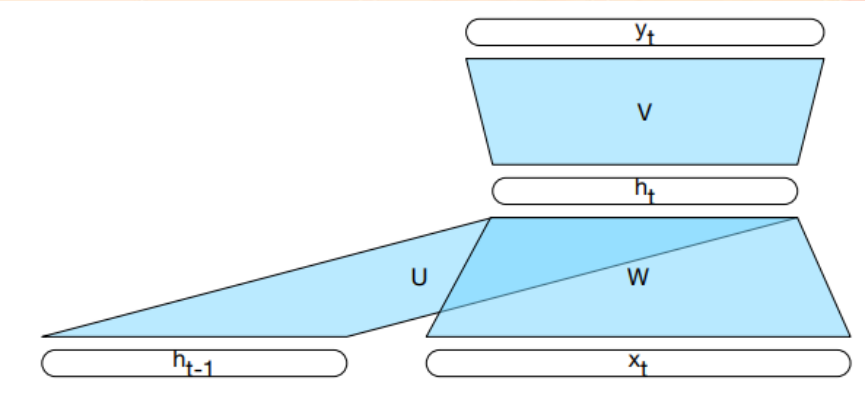
Forward inference *(i.e. mapping a sequence of inputs to a sequence of outputs) in an RNN is nearly identical to what we've already seen with feedforward networks.*

- *To compute an output y_t for an input x_t ,*

we need the activation value for the hidden layer h_t .

To calculate this, we compute the dot product of the input x_t with the weight matrix W, and the dot product of the hidden layer from the previous time step h_{t-1} with the weight matrix U.

We add these values together and pass them through a suitable activation function, g, to arrive at the activation value for the current hidden layer, h_t .



$$h_t = g(Uh_{t-1} + Wx_t)$$

Inference in RNNs (part 2)

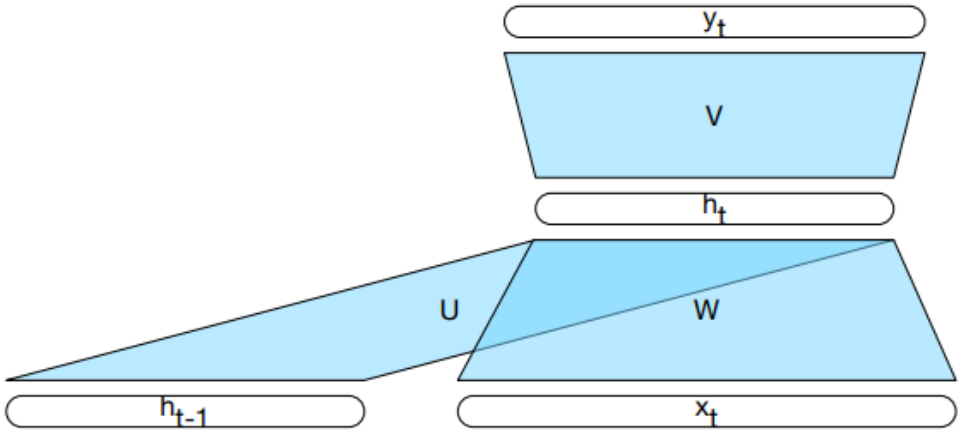
Forward inference *(i.e. mapping a sequence of inputs to a sequence of outputs) in an RNN is nearly identical to what we've already seen with feedforward networks.*

$$h_t = g(Uh_{t-1} + Wx_t)$$

- *Once we have the values for the hidden layer, we proceed with the usual computation to generate the output vector.*

$$y_t = f(Vh_t)$$

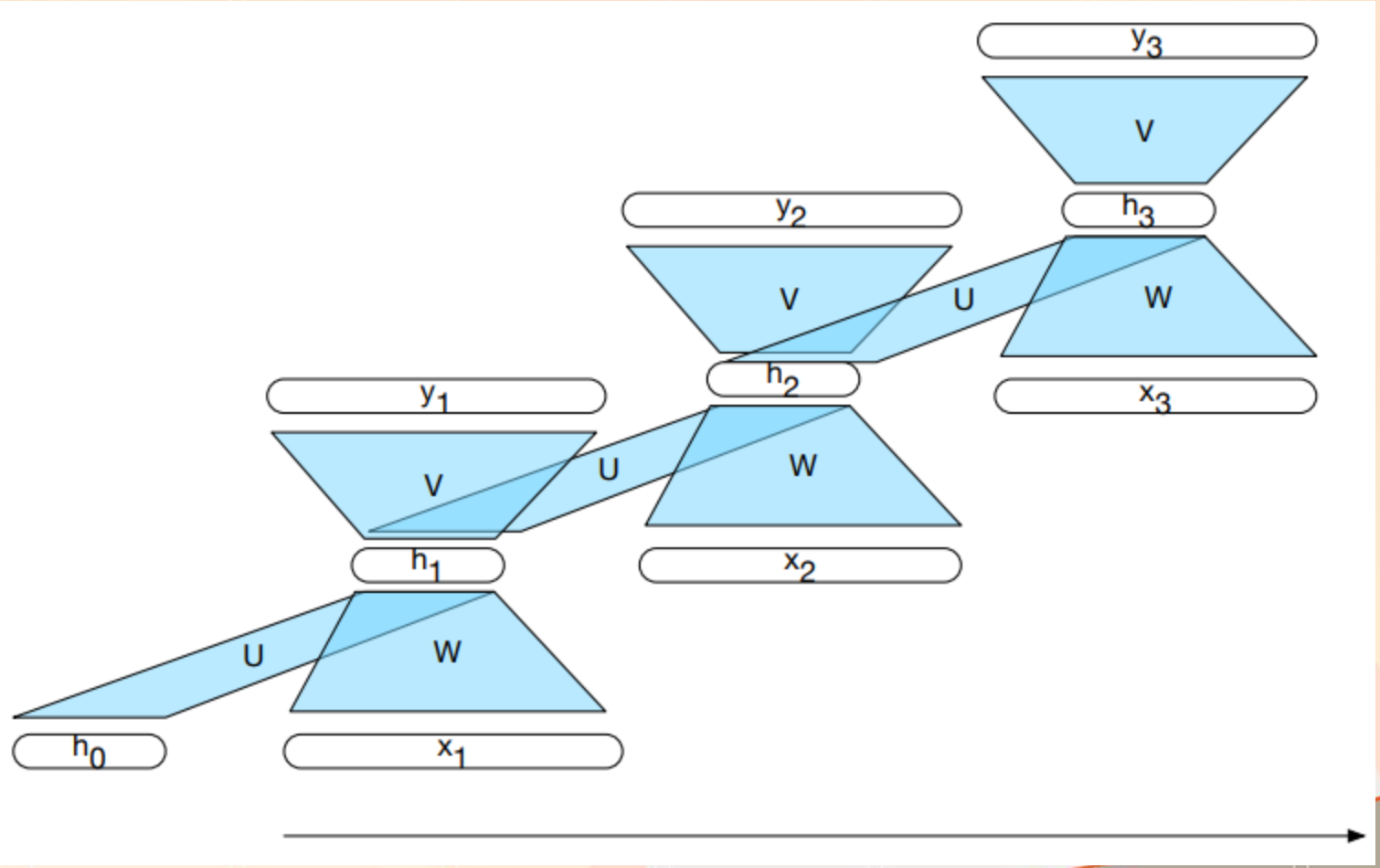
- *How????* *In the commonly encountered case of* ***soft classification****, finding y_t consists of a softmax computation that provides a normalized probability distribution over the possible output classes.*



$$y_t = softmax(Vh_t)$$

Unrolling the RNN

*The **sequential nature** of simple recurrent networks can be illustrated by unrolling the network in time!*

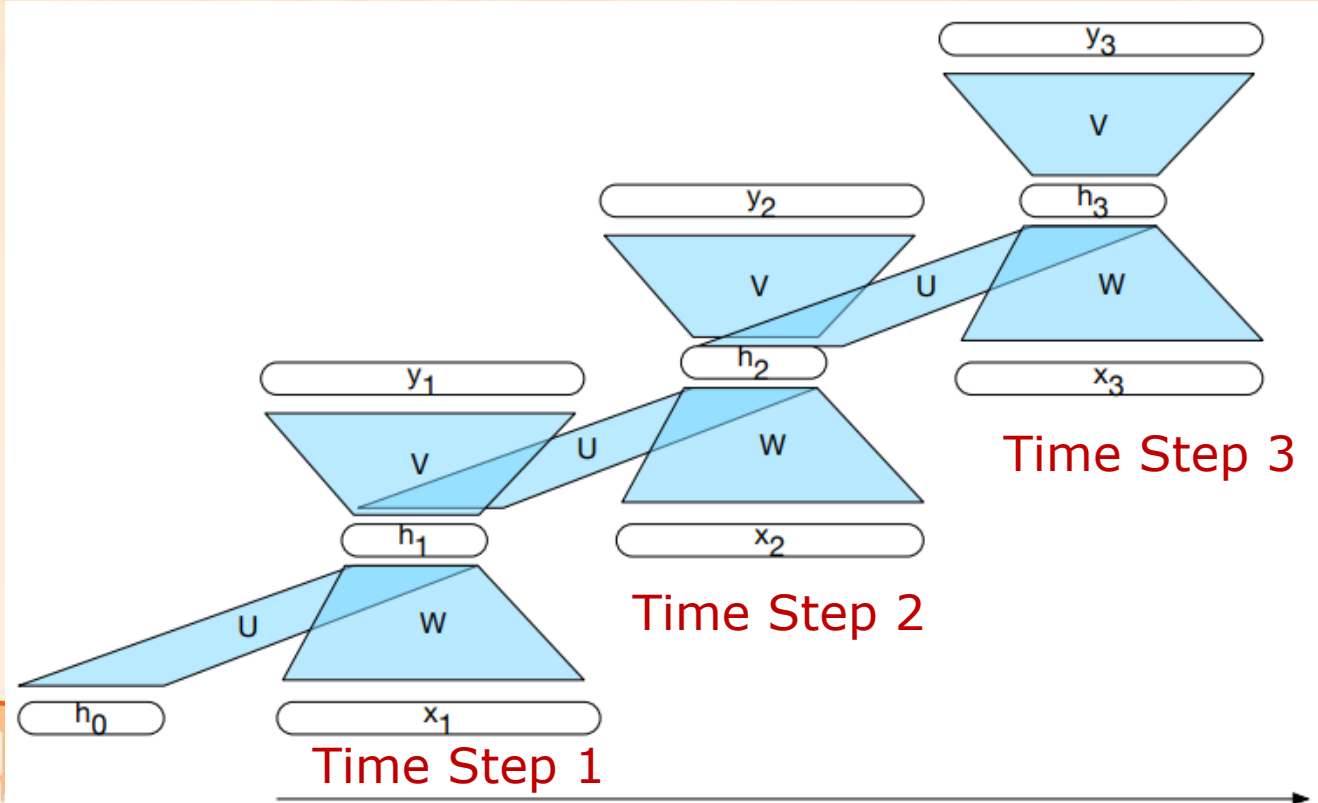


Unrolling the RNN

- *Various layers of the RNN are copied for each time step to illustrate that they will have differing values over time.*

However **the weights are shared across the various timesteps**.

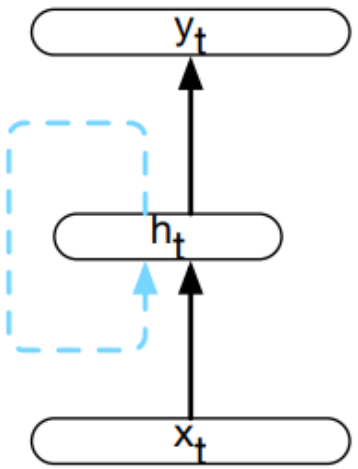
The fact that the computation at time t requires the value of the hidden layer from time t −1 mandates an incremental inference algorithm that proceeds from the start of the sequence to the end as shown in the Figure.



Forward inference with the RNN

function FORWARDRNN(x, *network*) **returns** output sequence y

$h_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH(x) **do**
  $h_i \leftarrow g(U\ h_{i-1} + W\ x_i)$
  $y_i \leftarrow f(V\ h_i)$
return y

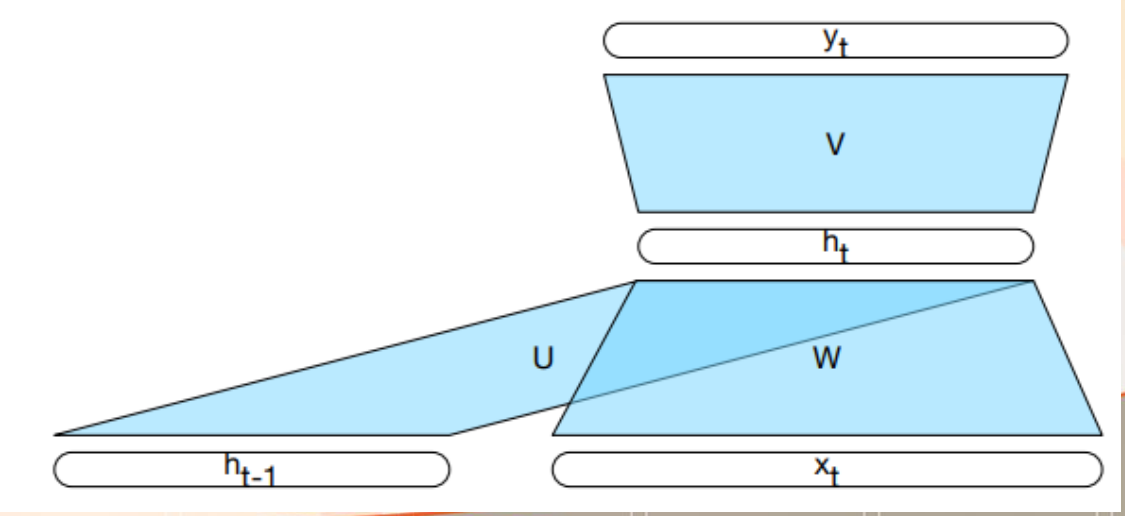


$$h_t = g(Uh_{t-1} + Wx_t)$$

$$y_t = f(Vh_t)$$

Training the RNN

- *It involves:*
  - *use a training set;*
  - *use a loss function, and*
  - *backpropagation to adjust the sets of weights in RNN.*
    - *we now have 3 sets of weights to update:*
    1. *W, the weights from the input layer to the hidden layer;*
    2. *U, the weights from the previous hidden layer to the current hidden layer, and*
    3. *V, the weights from the hidden layer to the output layer*



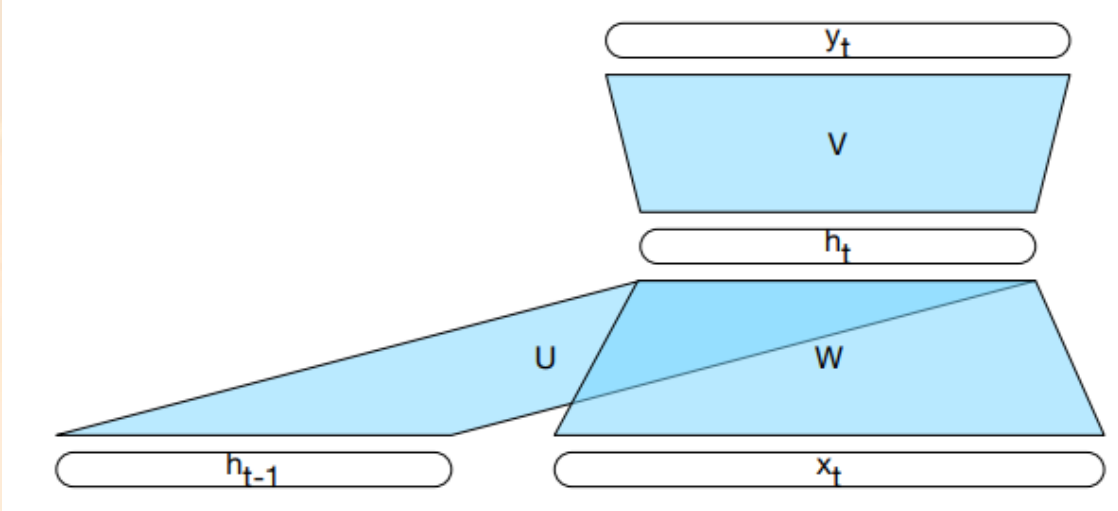
Some notations

*Assuming a network with an input layer **x** and a non-linear activation function **g**, we'll use:*

a[i] to refer to the activation value from a layer i, which is the result of applying g to z[i]= the weighted sum of the inputs to that layer.

??? A simple two-layer feedforward network with W and V as the first and second sets of weights respectively, would be characterized as follows:

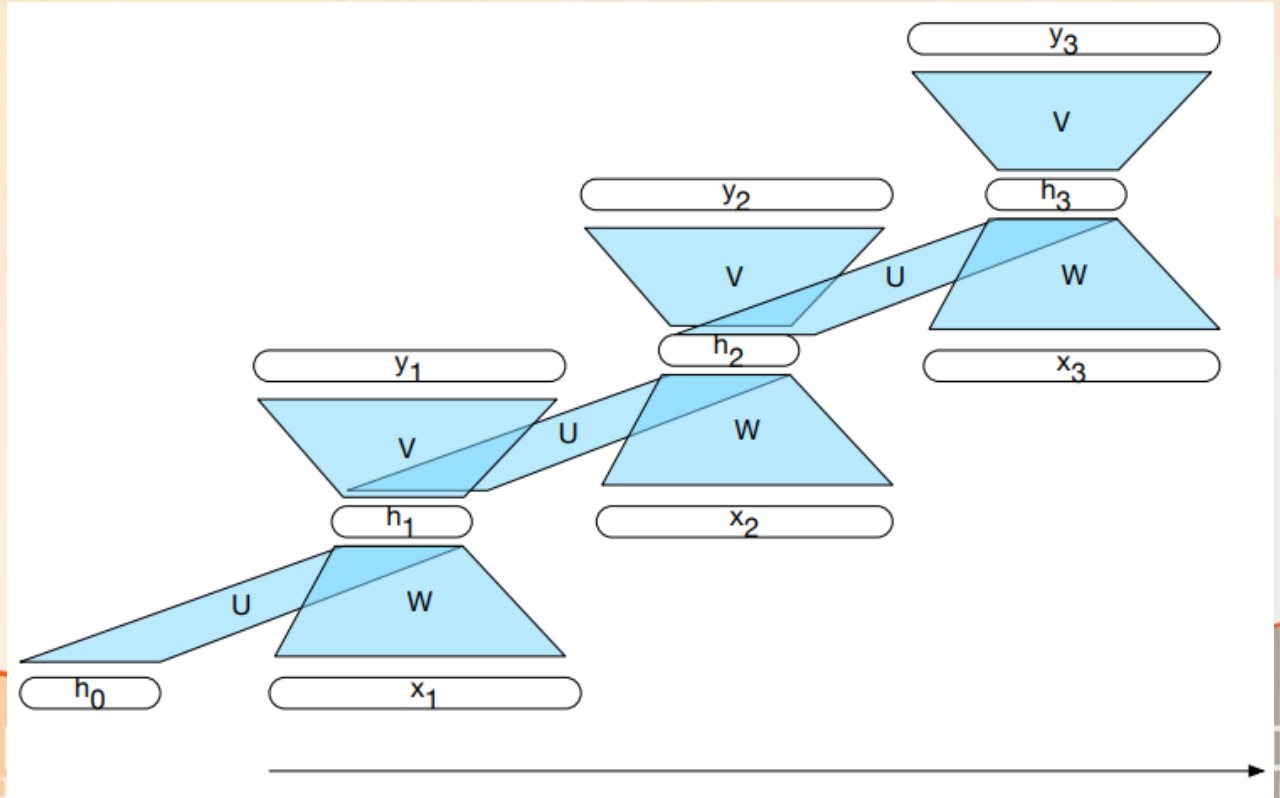
$$
\begin{aligned}
z^{[1]} &= Wx \\
a^{[1]} &= g(z^{[1]}) \\
z^{[2]} &= Ua^{[1]} \\
a^{[2]} &= g(z^{[2]}) \\
y &= a^{[2]}
\end{aligned}
$$



Considerations

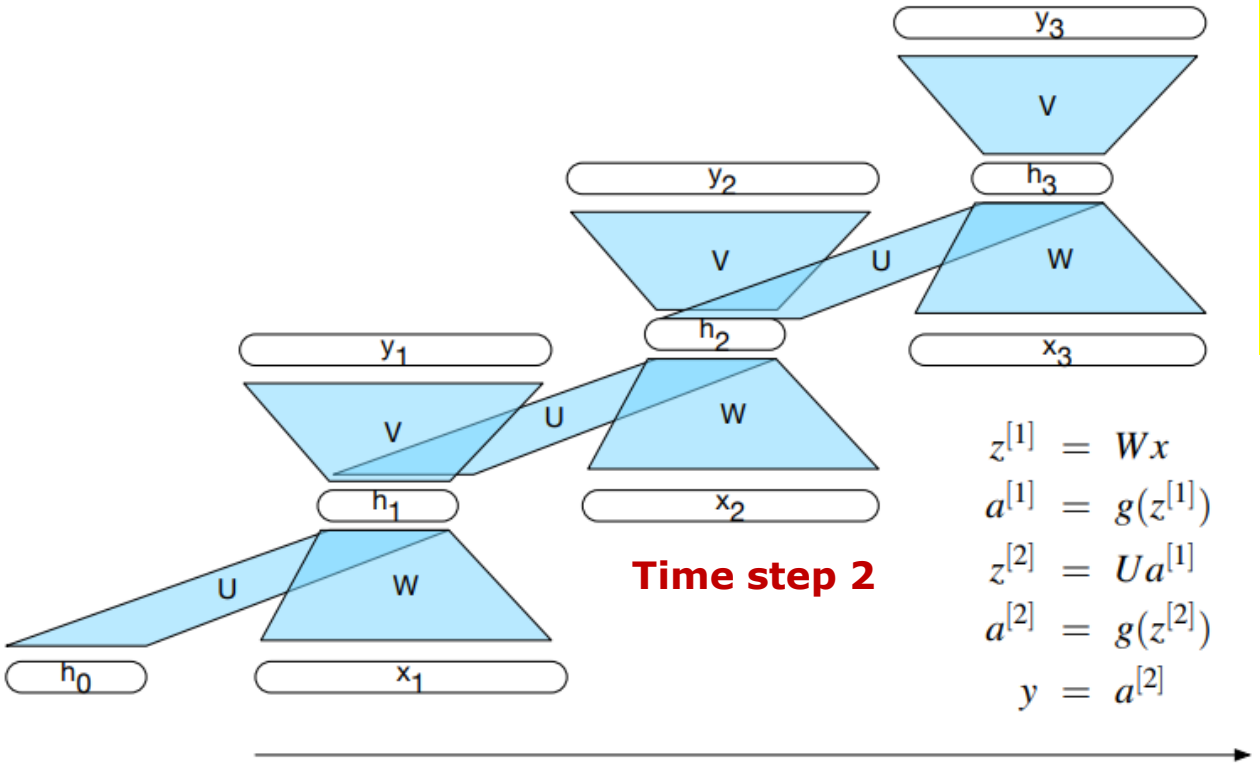
Two considerations to worry about for backpropagation in RNNs:

1. *To compute **the loss function** for the output at time t we need the hidden layer from time t − 1.*
2. *The hidden layer at time t influences both the output at time t and the hidden layer at time t +1 (and hence the output and loss at t +1). It follows from this that to assess the error accruing to h_t , we'll need to know its influence on both the current output as well as the next one.*



Example

- *Consider the situation where we are examining an input/output pair at time 2.*
- *What do we need to compute the gradients needed to update the weights U, V, and W here?*
  - *how we compute the gradients required to update V?*
    - *we need to compute the derivative of the loss function **L** with respect to the weights **V***

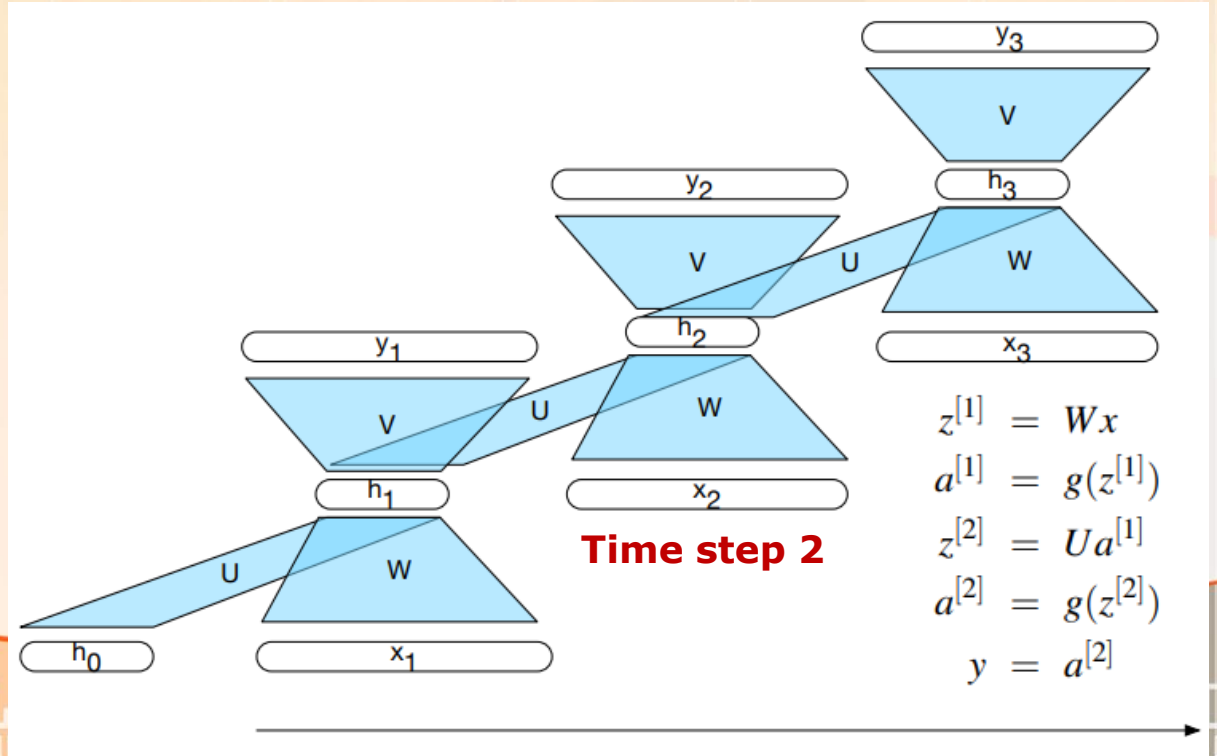


How???
Because the loss is not expressed directly in terms of the weights, we apply the chain rule to get there indirectly:

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V}$$

More on this Example

- *The first term is just the derivative of the loss function with respect to the network output, which is just the activation of the output layer, **a**.*
- *The second term is the derivative of the network output with respect to the intermediate network activation z, which is a function of the activation function g.*
- *The third term is the derivative of the network activation with respect to the weights **V**, which is just the activation value of the current hidden layer h_t .*



$$z^{[1]} = Wx$$
$$a^{[1]} = g(z^{[1]})$$
$$z^{[2]} = Ua^{[1]}$$
$$a^{[2]} = g(z^{[2]})$$
$$y = a^{[2]}$$

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V}$$

Are we done yet? NO

- *Let us define the **error terms!***
- ***δ_{out}** is an error term that represents how much of the scalar loss is attributable to each of the units in the output layer.*

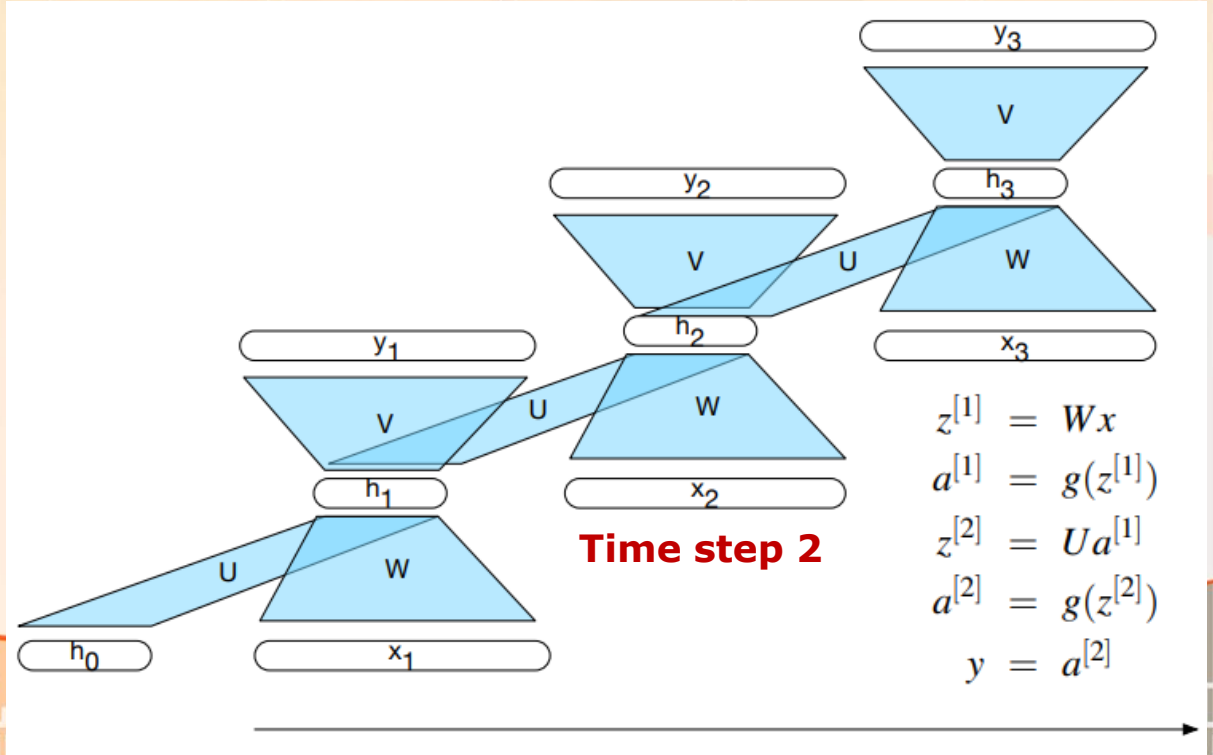
1st Error term

$$\delta_{out} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}$$

$$\delta_{out} = L' g'(z)$$

Therefore, the final gradient we need to update the weight matrix ***V*** is just:

$$\frac{\partial L}{\partial V} = \delta_{out} h_t$$



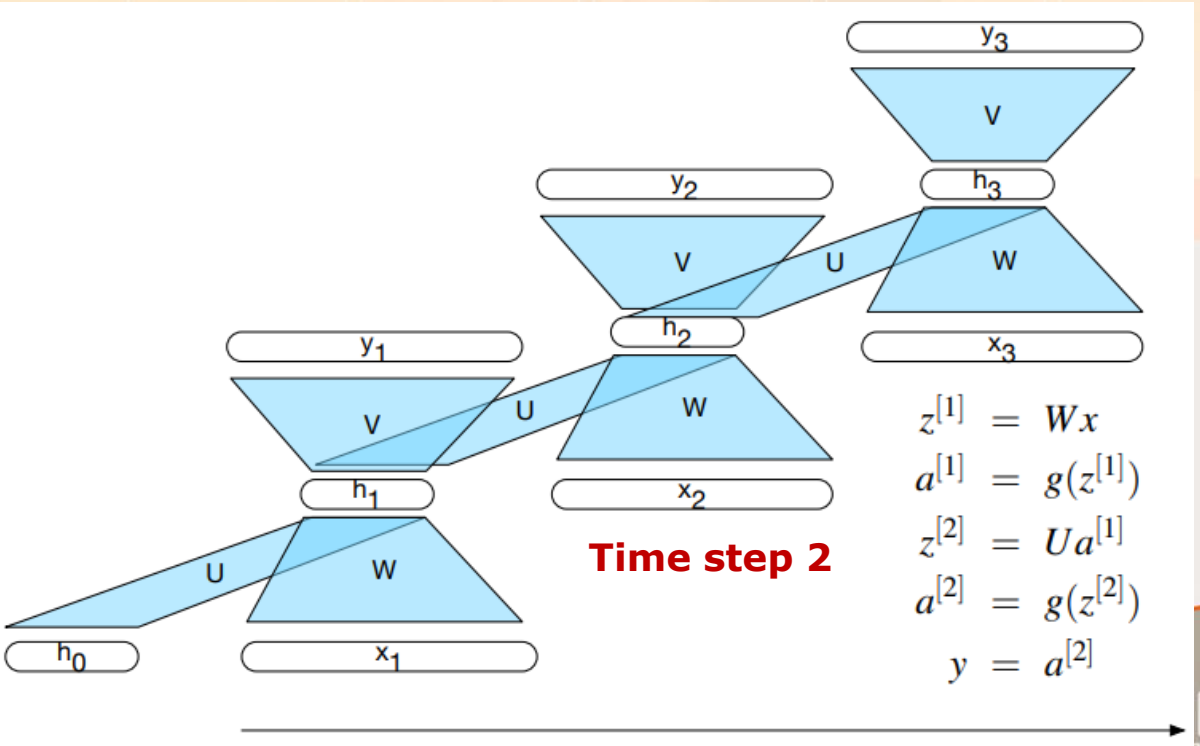
$$z^{[1]} = Wx$$
$$a^{[1]} = g(z^{[1]})$$
$$z^{[2]} = Ua^{[1]}$$
$$a^{[2]} = g(z^{[2]})$$
$$y = a^{[2]}$$

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V}$$

compute the gradients for matrices W and U

- ***How???*** $\frac{\partial L}{\partial W}$ $\frac{\partial L}{\partial U}$
- ***Problem:*** *The hidden state at time **t** contributes to the output and associated error at time **t** and to the output and error at the next timestep, t +1.*

Therefore, the error term, δ_h, for the hidden layer must be the sum of the error term from the current output and its error from the next time step.



$$z^{[1]} = Wx$$
$$a^{[1]} = g(z^{[1]})$$
$$z^{[2]} = Ua^{[1]}$$
$$a^{[2]} = g(z^{[2]})$$
$$y = a^{[2]}$$

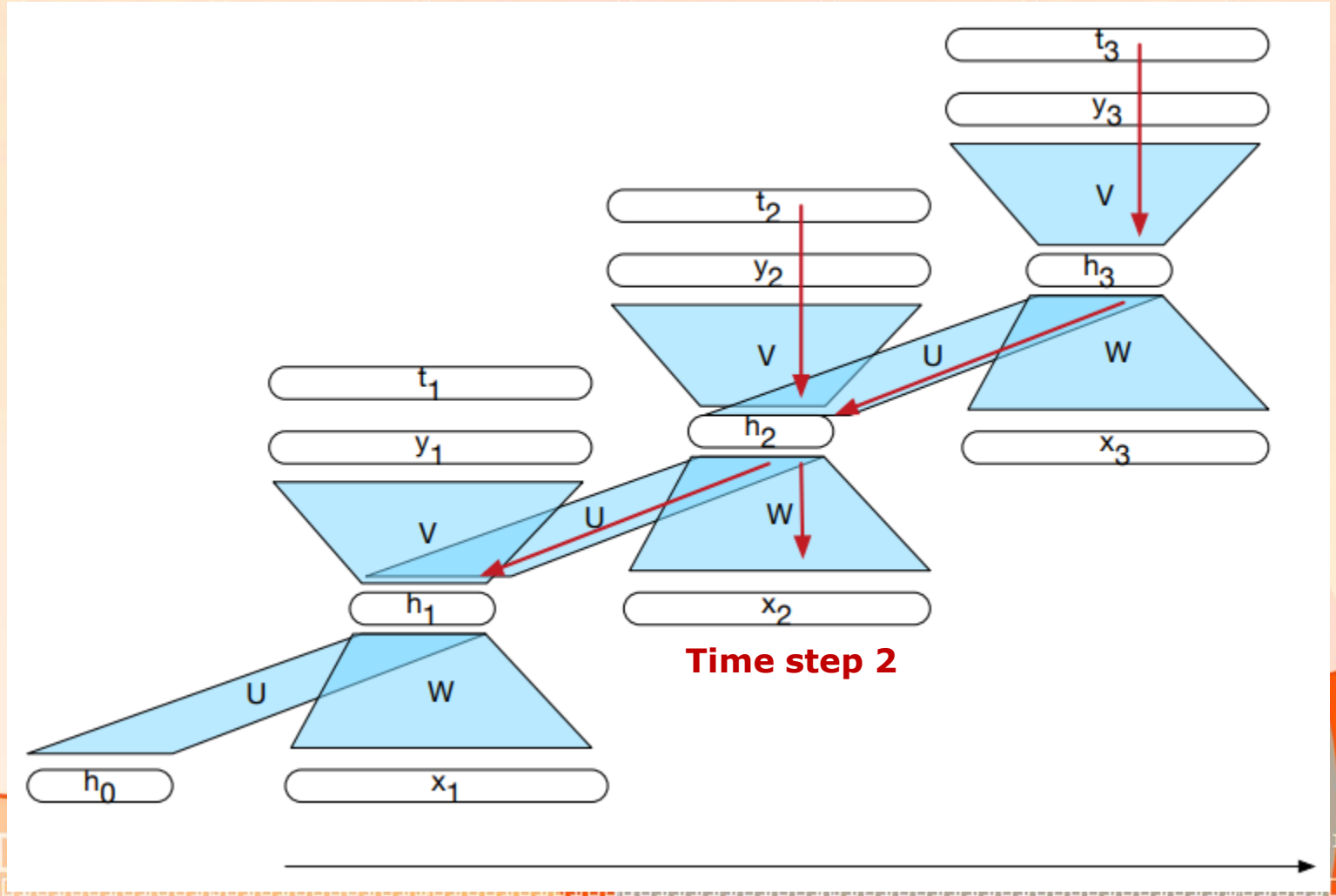
2nd Error term

$$\delta_h = g'(z)V\delta_{out} + \delta_{next}$$

The total error term for the hidden layer

The backpropagation of errors in an RNN

The t_i vectors represent the targets for each element of the sequence from the training data. The red arrows illustrate the flow of backpropagated errors required to calculate the updates for *U*, *V* and *W* at time 2. The two incoming arrows converging on h_2 signal that these errors need to be summed.



compute the gradients for matrices W and U

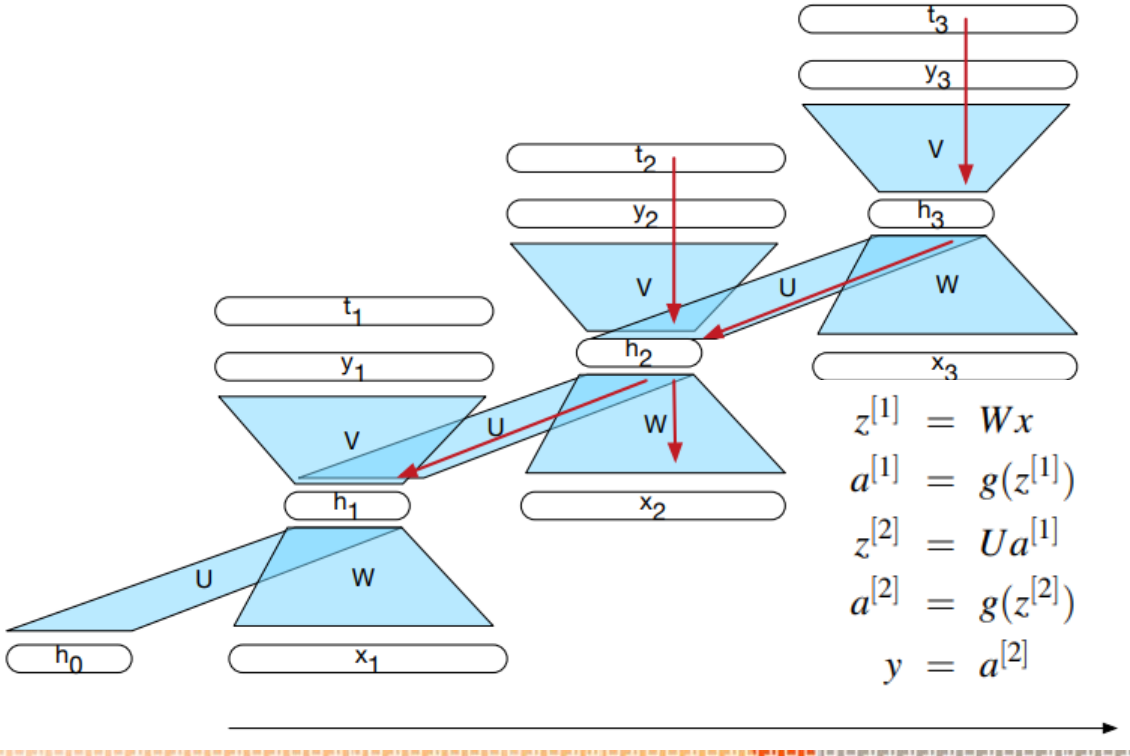
- *We can now compute the gradients for the weights U and W in the usual way, using the chain rule.*

$$\delta_h = g'(z)V\delta_{out} + \delta_{next}$$

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{da}\frac{da}{dW}$$
$$\frac{dL}{dU} = \frac{dL}{dz}\frac{dz}{da}\frac{da}{dU}$$

$$\frac{\partial L}{\partial W} = \delta_h x_t$$
$$\frac{\partial L}{\partial U} = \delta_h h_{t-1}$$

These gradients provide us with the information needed to update the matrices U and W through ordinary backpropagation.



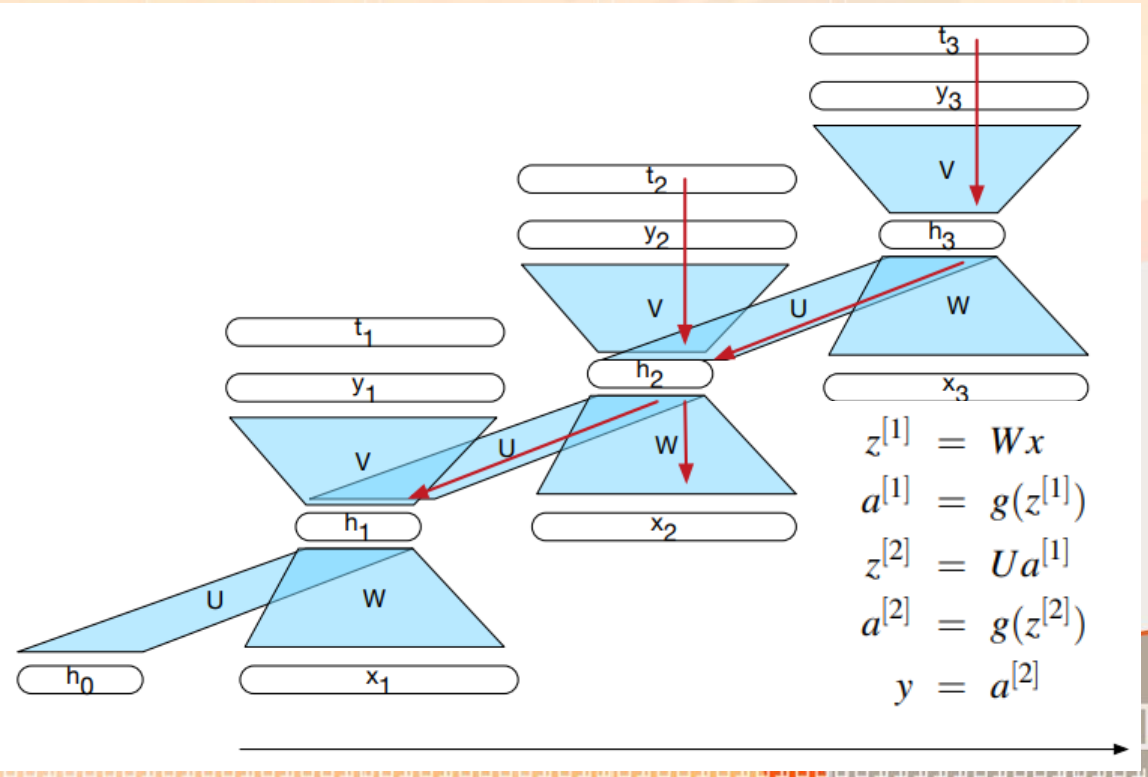
$$z^{[1]} = Wx$$
$$a^{[1]} = g(z^{[1]})$$
$$z^{[2]} = Ua^{[1]}$$
$$a^{[2]} = g(z^{[2]})$$
$$y = a^{[2]}$$

One more error term!

- *We still need to assign proportional blame (compute the error term) back to the previous hidden layer h_{t-1} for use in further processing.*
- *This involves backpropagating the error from δ_h to h_{t-1} proportionally based on the weights in U.*

3rd Error term

$$\delta_{next} = g'(z)U\,\delta_h$$



$$z^{[1]} = Wx$$
$$a^{[1]} = g(z^{[1]})$$
$$z^{[2]} = Ua^{[1]}$$
$$a^{[2]} = g(z^{[2]})$$
$$y = a^{[2]}$$

How many error terms?????

1. δ_{out}
2. δ_h
3. δ_{next}

Backpropagation training through time

- *It is a two-pass algorithm for training the weights in RNNs.*
  1. *In the first pass, we perform forward inference, computing h_t , y_t , and a loss at each step in time, saving the value of **the hidden layer** at each step for use at the next time step.*
  2. *In the second phase, we process the sequence in reverse, computing the required **error terms gradients** as we go, computing and saving the error term for use in the hidden layer for each step backward.*

function BACKPROPTHROUGHTIME(*sequence*, *network*) **returns** gradients for weight updates
forward pass to gather the loss
backward pass compute error terms and assess blame

IMPORTANT: Unfortunately, computing the gradients and updating weights for each item of a sequence individually would be extremely time-consuming.

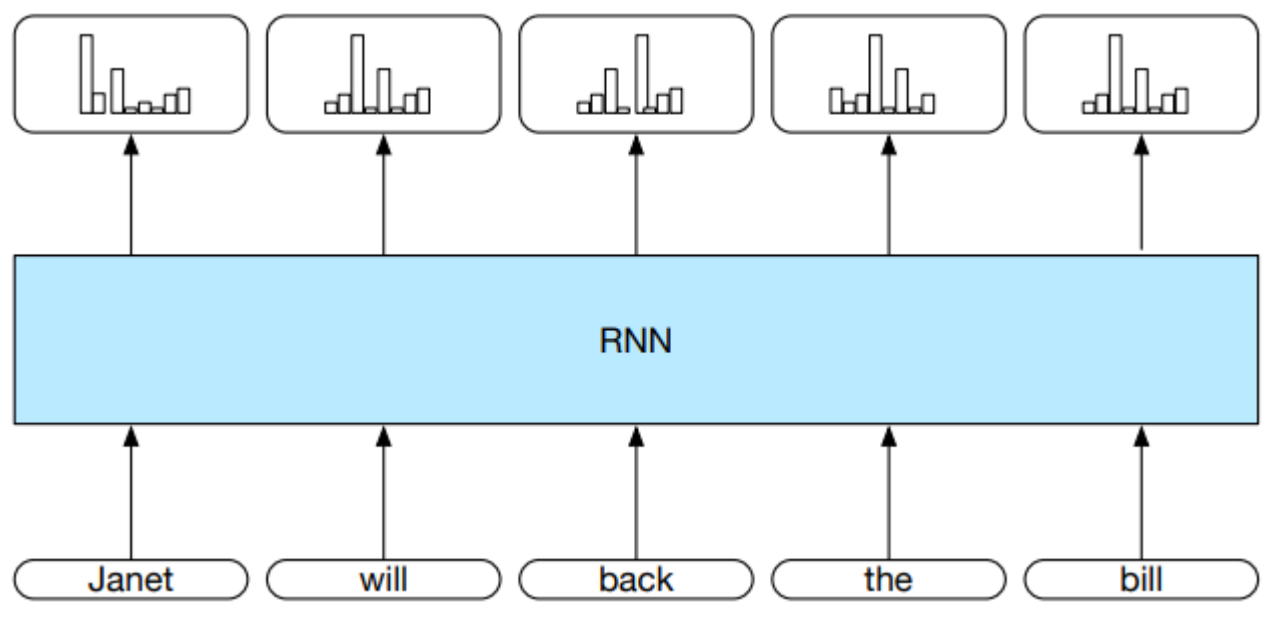
- Instead, we accumulate gradients for the weights incrementally over the sequence, and then use those accumulated gradients in performing weight updates.

Depending on applications!

- *For applications that involve much longer input sequences, such as speech recognition, character-by-character sentence processing, or streaming of continuous inputs, unrolling an entire input sequence may not be feasible.*
- *In these cases, we can unroll the input into manageable fixed-length segments and treat each segment as a distinct training item. This approach is called* ***Truncated Backpropagation Through Time (TBTT).***

RNNs for sequence labeling

- *In sequence labeling, the RNN's job is to **assign a label** to each element of a sequence chosen from a small fixed set of labels.*
  - *The canonical example of such a task is part-of-speech tagging.*



The inputs at each time step are pre-trained word embeddings corresponding to the input tokens. The RNN block is an abstraction that represents an unrolled RNN consisting of (1) an input layer, (2) hidden layer, and (3) output layer at each time step, as well as the shared *U*, *V* and *W* weight matrices that comprise the network. The outputs of the network at each time step represent the distribution over the POS tagset generated by a *softmax* layer.

RNNs for Named Entity Recognition

- *An application of sequence labeling is to find and classify **spans of text** corresponding to items of interest in **named entity recognition** — the problem of finding all the spans in a text that correspond to names of people, places or organizations.*
  - *Cast NER as a per-word sequence labeling task, by using a technique called **IOB encoding**. ???*
  - *We'll label any token that begins a span of interest (i.e a name!!!) with the label **B**, tokens that occur inside a span are tagged with an **I**, and any tokens outside of any span of interest are labeled **O**.*
  - *Example:*

United	*cancelled*	*the*	*flight*	*from*	*Denver*	*to*	*San*	*Francisco.*
B	O	O	O	O	B	O	B	I

  - *specialize the B and I tags:*

United	*cancelled*	*the*	*flight*	*from*	*Denver*	*to*	*San*	*Francisco.*
B-ORG	O	O	O	O	B-LOC	O	B-LOC	I-LOC

NER Recognition

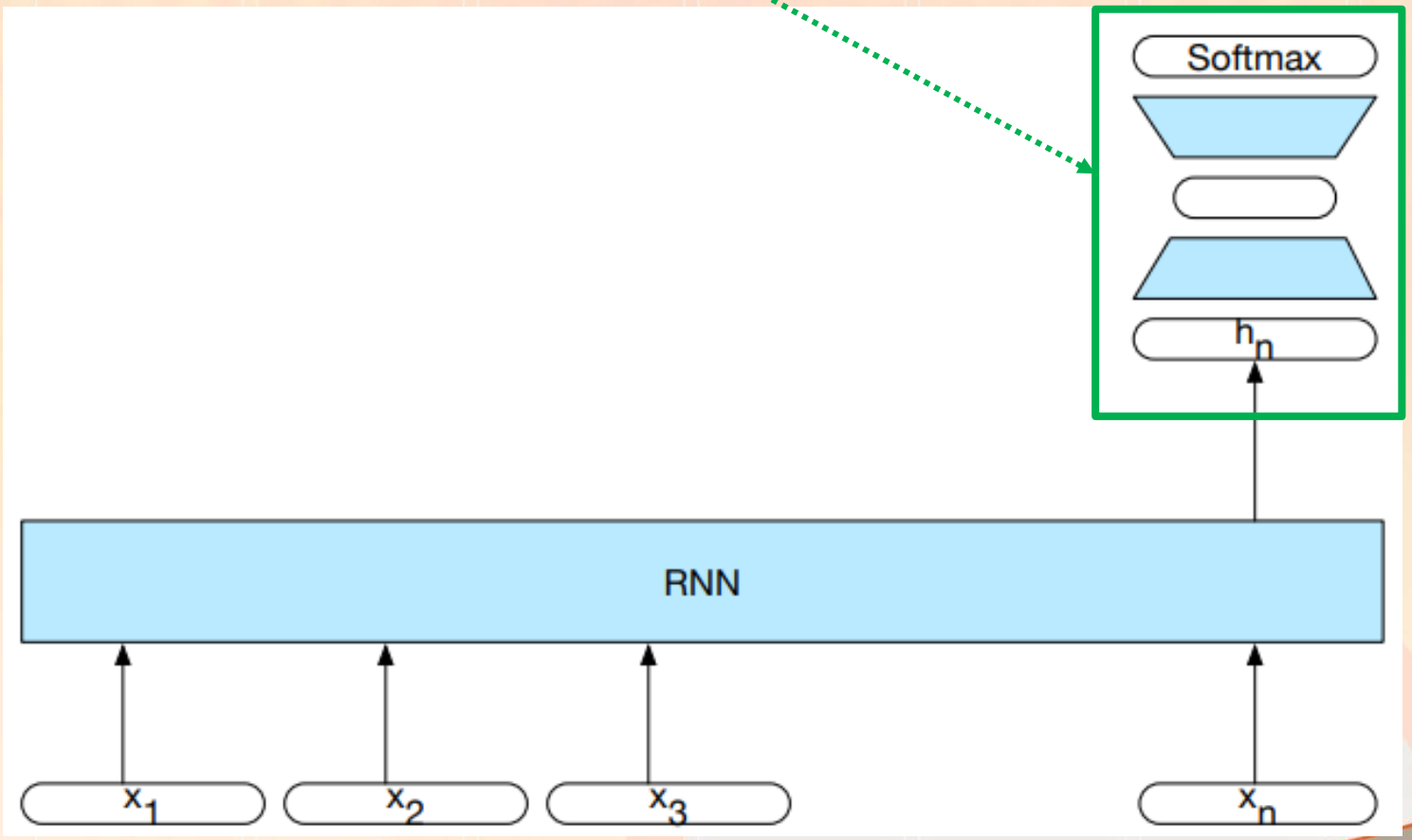
- *With such an encoding, **the inputs** are the usual word embeddings and **the output** consists of a sequence of softmax distributions over the tags at each point in the sequence.*

United	*cancelled*	*the*	*flight*	*from*	*Denver*	*to*	*San*	*Francisco.*
B-ORG	O	O	O	O	B-LOC	O	B-LOC	I-LOC

- *Neural Architecture: To apply RNNs in this setting, the hidden layer from the final state of the network is taken to constitute a compressed representation of the entire sequence. This compressed sequence representation can then in turn serve as the input to a feed-forward network trained to select the correct class.*

Deep NN for NER recognition

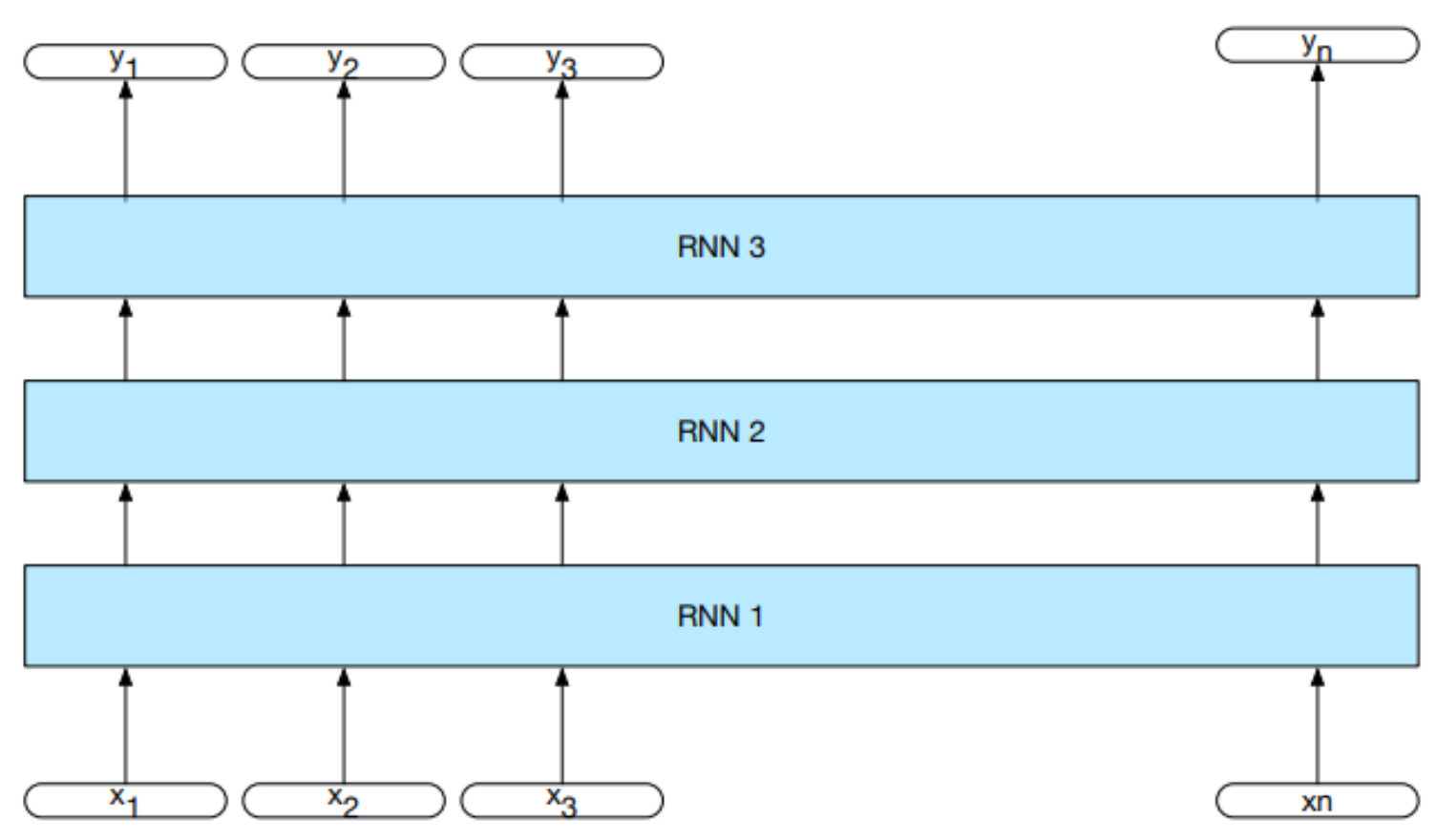
- *NER as sequence classification using a simple RNN combined with a feedforward network.*



The loss is backpropagated all the way through the weights in the feedforward classifier through to its input, and then through to the three sets of weights in the RNN

Deep Networks: Stacked RNNs

- *Use the entire sequence of outputs from one RNN as an **input sequence** to another one.*
- *Stacked RNNs Stacked RNNs consist of multiple networks where the output of one layer serves as the input to a subsequent layer!*



Stacked RNNS

- *It has been demonstrated across numerous tasks that stacked RNNs can outperform single-layer networks.*
- *Why??? One reason for this success has to do with the networks ability to induce representations at different levels of abstraction across layers.*
  - *the initial layers of stacked networks can induce representations that serve as useful abstractions for further layers — representations that might prove difficult to induce in a single RNN.*

Bi-directional RNNs

- *In an simple recurrent network, the hidden state at a given time t represents everything the network knows about the sequence up to that point in the sequence.*
  - *The hidden state at time t is the result of a function of the inputs from the start up through time t. We can think of this as the context of the network to the left of the current time:*

$$h_t^{forward} = SRN_{forward}(x_1 : x_t)$$

- *Where $h_t^{forward}$ corresponds to the normal hidden state at time t, and represents everything the network has gleaned from the sequence to that point.*

The whole context!

- *It is helpful to take advantage of the context to the right of the current input as well.*
- *How??? Train an RNN on an input sequence in reverse! With this approach, the hidden state at time t will represent information about the sequence to the right of the current input:*

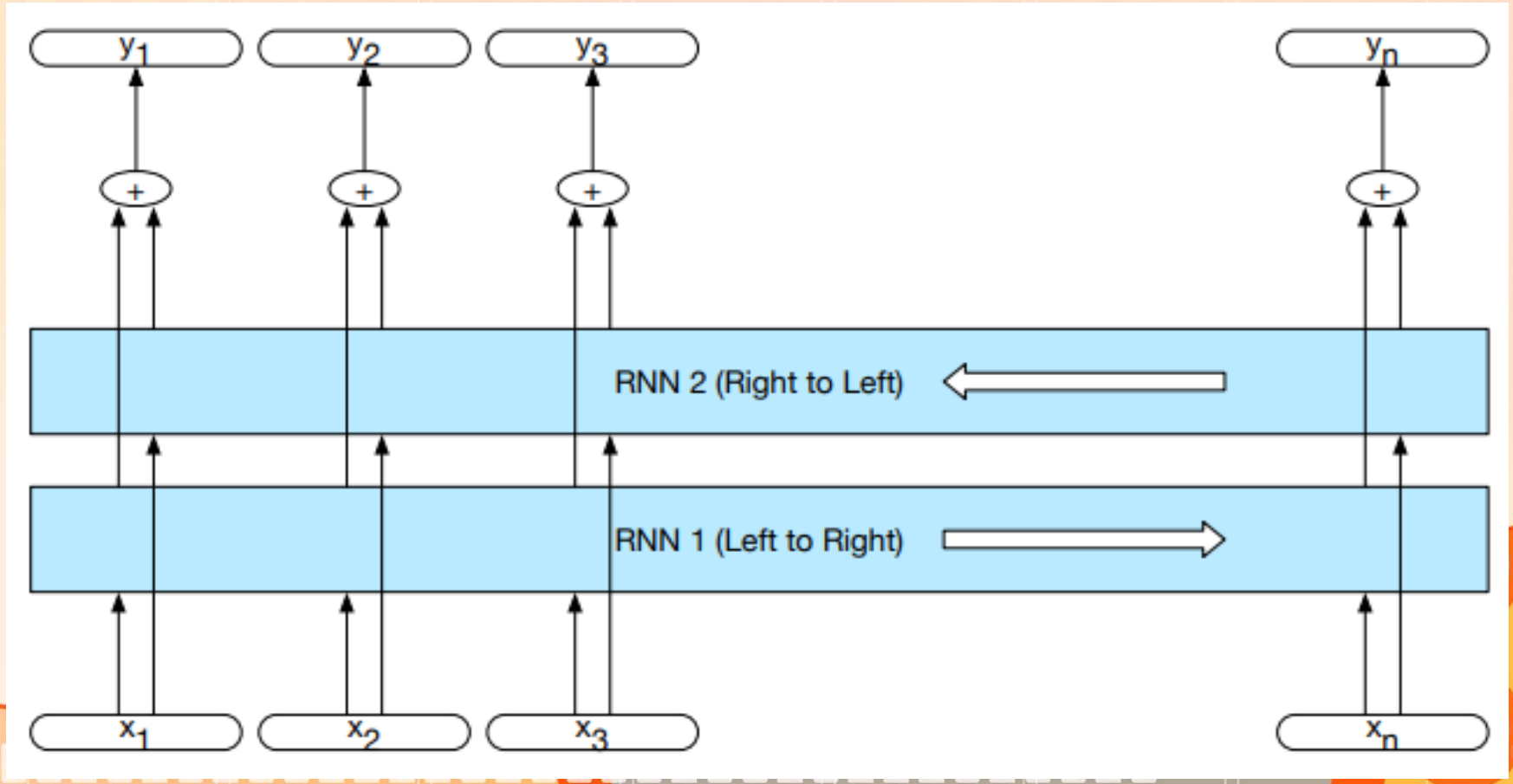
$$h_t^{backward} = SRN_{backward}(x_n : x_t)$$

- *the hidden state $h_t^{backward}$ represents all the information we have discerned about the sequence from t to the end of the sequence.*

Let us put both networks together!

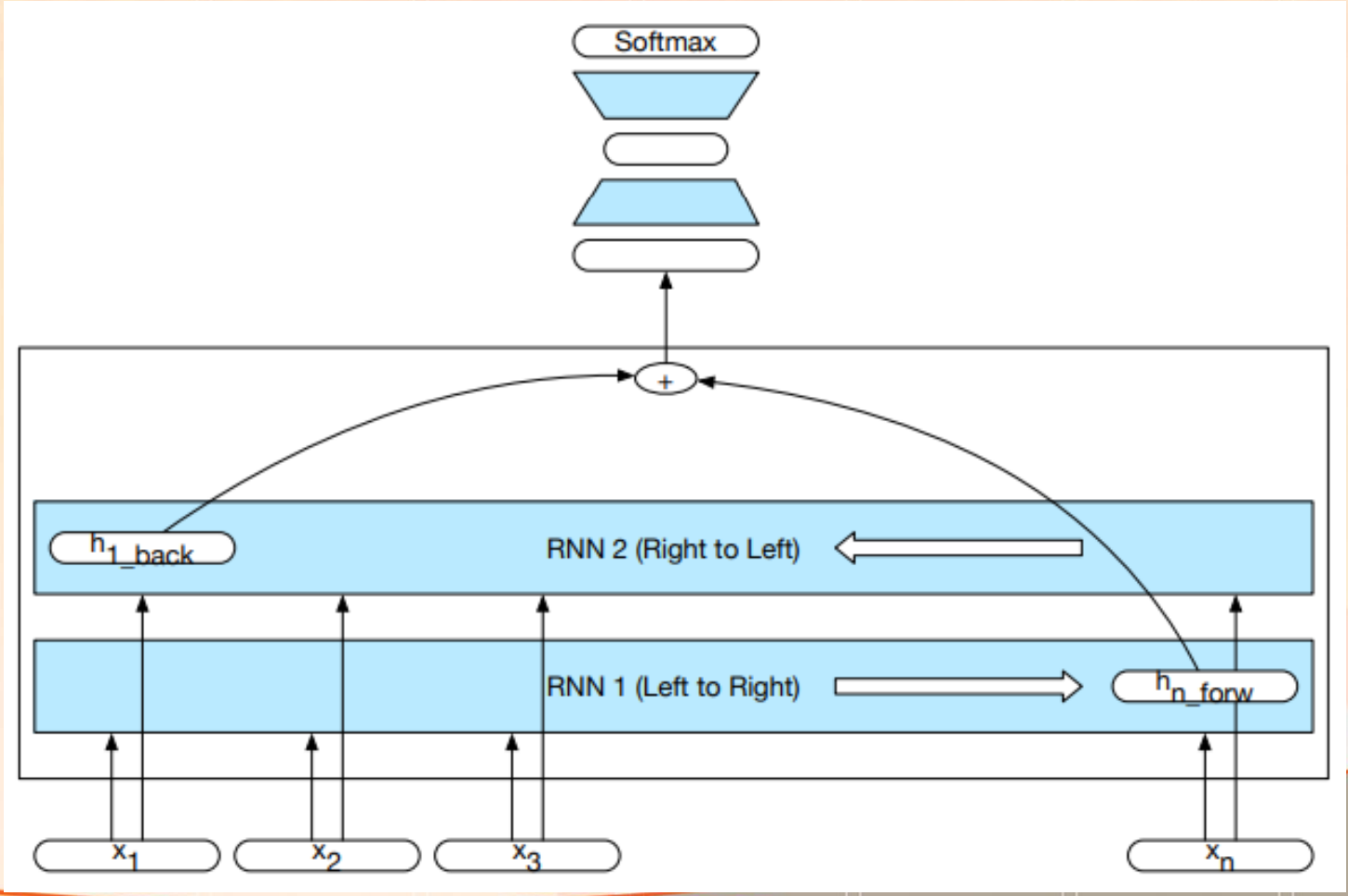
Bi-Directional Architecture

- *Separate models are trained in the forward and backward directions with the output of each model at each time point concatenated to represent the state of affairs at that point in time:* $h_t = h_t^{forward} \oplus h_t^{backward}$



Bi-directional RNN for Sequence Classification

- *The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.*



Managing Context in RNNs

- *Despite having access to the entire preceding sequence, the information encoded in hidden states tends to be fairly local, more relevant to the most recent parts of the input sequence and recent decisions. However, it is often the case that **long-distance information** is critical to many language applications.*
- *Example:*

The flights the airline was cancelling were full.

Assigning a high probability to "*was*" following "*airline*" is straightforward since "*was*" provides a strong local context for the singular agreement.

Assigning an appropriate probability to "*were*" is quite difficult, not only because the plural "*flights*" is quite distant, but also because the more recent context contains singular constituents.

Why is this difficult?

1. *The hidden layer in SRNs, and, by extension, the weights that determine the values in the hidden layer, are being asked to perform two tasks simultaneously:*
   a) *provide information useful to the decision being made in the current context, and*
   b) *updating and carrying forward information useful for future decisions.*
2. *When training simple recurrent networks we need to backpropagate training errors back in time through the hidden layers. The hidden layer at time t contributes to the loss at the next time step since it takes part in that calculation. As a result, during the backward pass of training, the hidden layers are subject to repeated dot products, as determined by the length of the sequence. A frequent result of this process is that the gradients are either driven to zero or saturate. Situations that are referred to as* ***vanishing gradients*** *or exploding gradients, respectively.*

Solution

- *More complex network architectures have been designed to explicitly manage the task of maintaining contextual information over time.*
- *These networks treat context as a kind of memory unit that needs to be managed explicitly. More specifically, the network needs to forget information that is no longer needed and to remember information as needed for later decisions.*

The Solution

- *More complex network architectures have been designed to explicitly manage the task of maintaining contextual information over time.*
  - *These approaches treat context as **a kind of memory unit** that needs to be managed explicitly.*
  - *More specifically, the network needs to forget information that is no longer needed and to remember information as needed for later decisions.*
- ***Long short-term memory (LSTM)** networks*

LSTMs

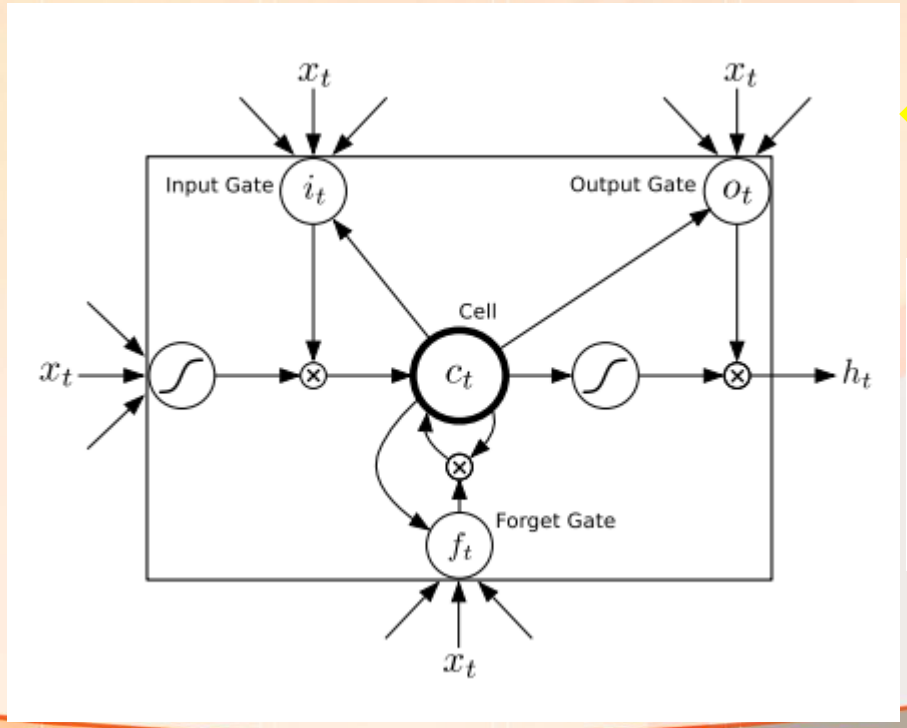
- *Long short-term memory (LSTM) networks, divide the context management problem into two sub-problems:*

1. *removing information no longer needed from the context, and*
2. *adding information likely to be needed for later decision making.*

*The key to the approach is **to learn how to manage this context** rather than hard-coding a strategy into the architecture.*

LSTM operation

- *LSTMs manage context through the use of **specialized neural units** that make use of gates that control the flow of information into and out of the units that comprise the network layers. These gates are implemented through the use of additional sets of weights that operate sequentially on the context layer.*

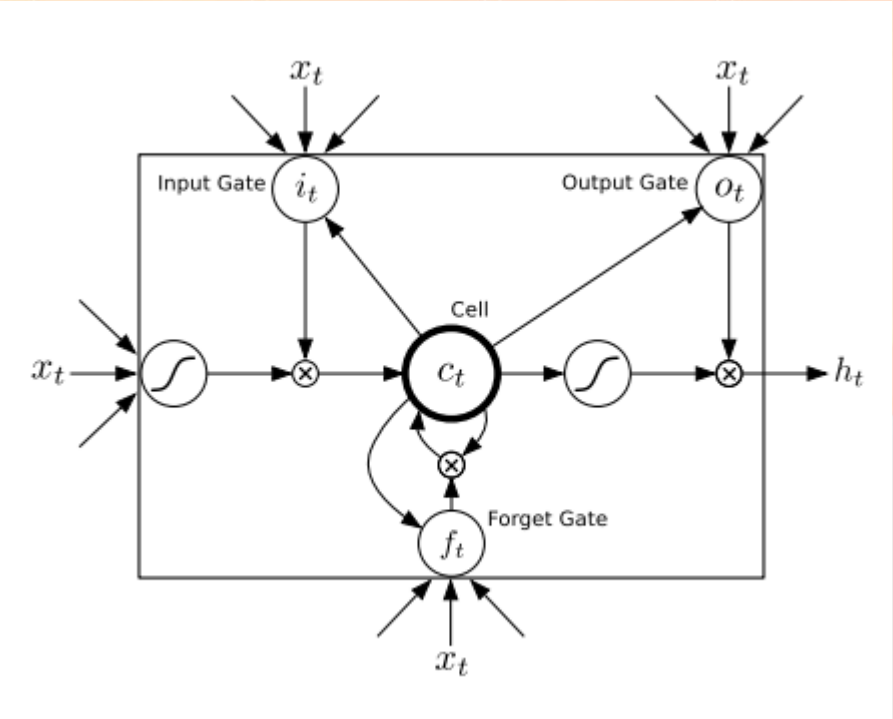


LSTM unit

$$
\begin{aligned}
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right) \\
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right) \\
c_t &= f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right) \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

Neural networks with LSTMs

- *An LSTM unit uses input and output gates to control the flow of information through the cell. T*
- *the input gate should be kept sufficiently active to allow the signals in. Same rule applies to the output gate.*
- *The forget gate is used to reset the cell's own state*

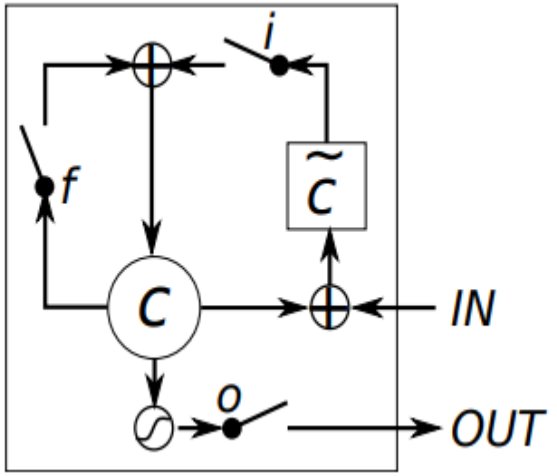


LSTM explained

The Long Short-Term Memory (LSTM) unit was initially proposed by Hochreiter and Schmidhuber [1997]. Since then, a number of minor modifications to the original LSTM unit have been made. We follow the implementation of LSTM as used in Graves [2013].

- *Unlike to the recurrent unit which simply computes a weighted sum of the input signal and applies a nonlinear function, each j-th LSTM unit maintains a memory c_t^j at time t. The output h_t^j, or the activation, of the LSTM unit is then:*

$$h_t^j = o_t^j \tanh\left(c_t^j\right),$$



*where o_t^j is **the output gate** that modulates the amount of memory content exposure. The output gate is computed:*

$$o_t^j = \sigma\left(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + V_o \mathbf{c}_t\right)^j$$

where σ is the sigmoid function. V_o is a diagonal matrix.

*The **memory cell** c_t^j is updated by partially forgetting the existing memory and adding a new memory content $\tilde{c}_t^j$:*

$$c_t^j = f_t^j c_{t-1}^j + i_t^j \tilde{c}_t^j,$$

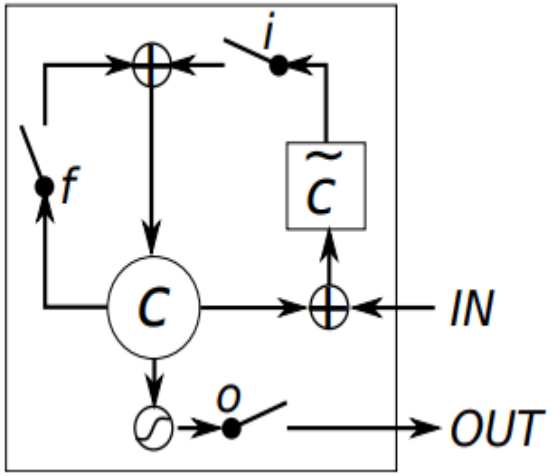
LSTM explained-2

*The **memory cell** c_t^j is updated by partially forgetting the existing memory and adding a new memory content $\tilde{c}_t^j$:*

$$c_t^j = f_t^j c_{t-1}^j + i_t^j \tilde{c}_t^j$$

where the new memory content is:

$$\tilde{c}_t^j = \tanh\left(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1}\right)^j$$



*The extent to which the **existing memory** is **forgotten** is modulated by a **forget gate** f_t^j, and the degree to which the new memory content is added to the memory cell is modulated by an **input gate** i_t^j. Gates are computed by:*

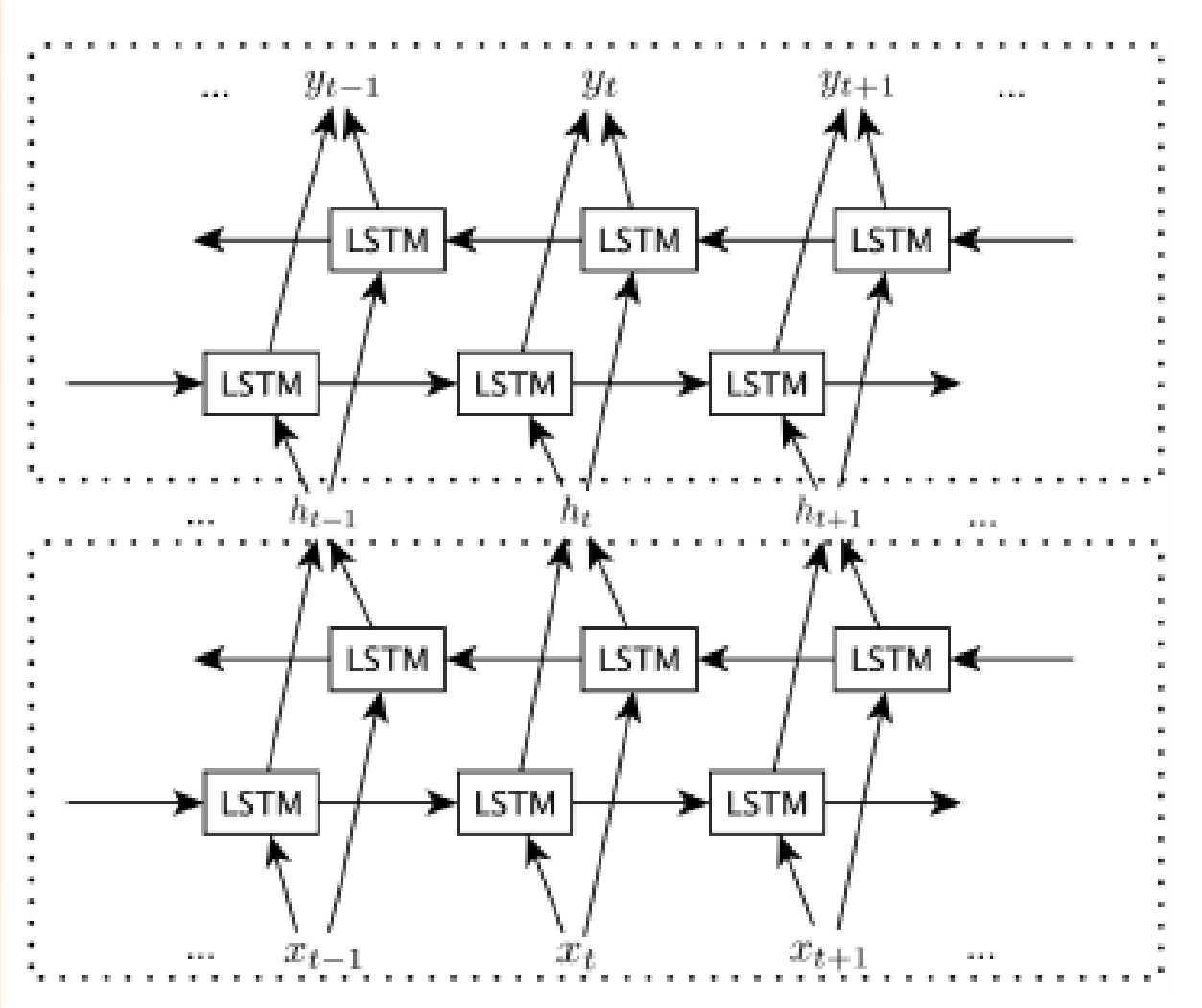
$$f_t^j = \sigma\left(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + V_f \mathbf{c}_{t-1}\right)^j$$
$$i_t^j = \sigma\left(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + V_i \mathbf{c}_{t-1}\right)^j .$$

where V_f and V_i are diagonal matrices.

Unlike to the traditional recurrent unit which overwrites its content at each time-step, an LSTM unit is able to decide whether to keep the existing memory via the gates. Intuitively, if the LSTM unit detects an important feature from an input sequence at early stage, it easily carries this information (the existence of the feature) over a long distance, hence, capturing potential long-distance dependencies.

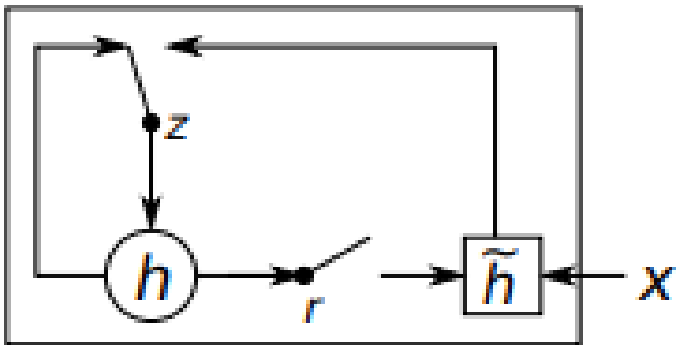
Putting everything together!!!

- *A stacked bi-directional LSTM network:*



Gated Recurrent Units

- *While relatively easy to deploy, LSTMs introduce a considerable number of parameters to neural networks, and hence carry **a much larger training burden**.*
- ***Gated Recurrent Units** (GRUs) try to ease this burden by collapsing the forget and add gates of LSTMs into a single update gate with a single set of weights.*

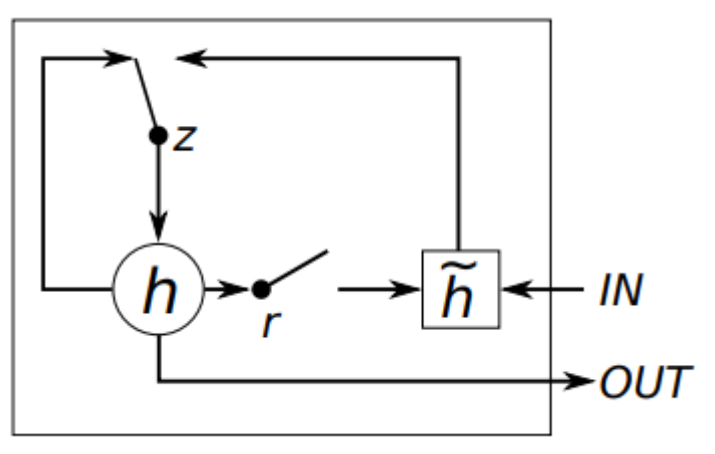


Gated Recurrent Unit(GRU)

the GRU has gates that modulate the flow of information inside the unit, however, without having separate memory cells!!!

GRU Operation

The activation h_t^j of the GRU at time t is a linear interpolation between the previous activation h_{t-1}^j and the candidate activation $\tilde{h}_t^j$:



$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \tilde{h}_t^j,$$

where an update gate z_t^j decides how much the unit updates its activation, or content. The update gate is computed by:

$$z_t^j = \sigma \left(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1} \right)^j$$

and candidate activation $\tilde{h}_t^j$ is computed like in the traditional recurrent unit:

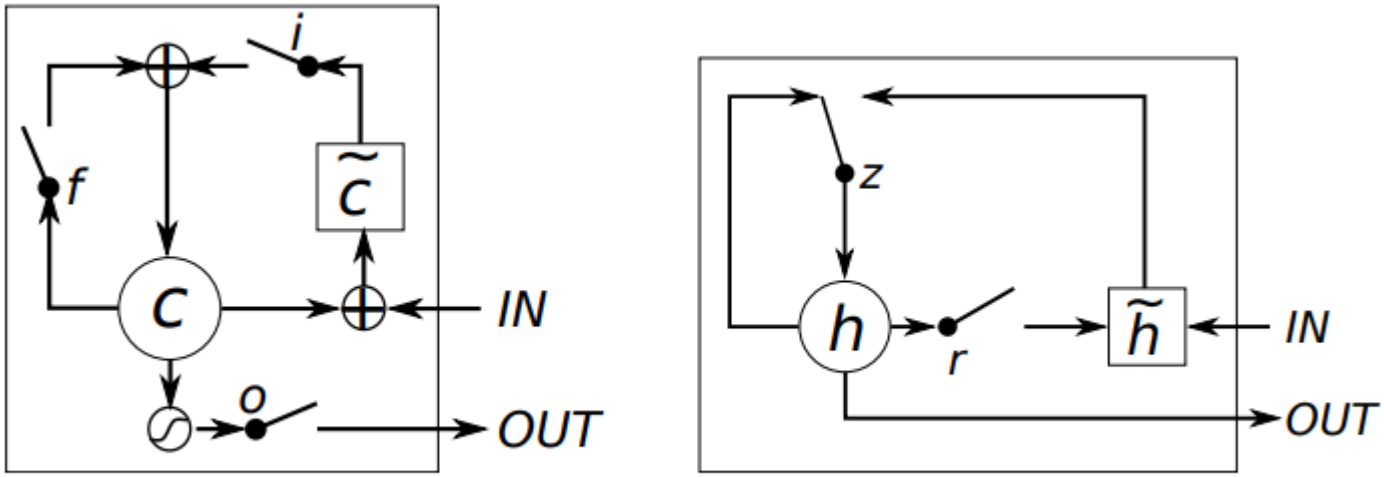
$$\tilde{h}_t^j = \tanh \left(W \mathbf{x}_t + U \left(\mathbf{r}_t \odot \mathbf{h}_{t-1} \right) \right)^j$$

The reset gate r_t^j is computed similarly to the update gate:

$$r_t^j = \sigma \left(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1} \right)^j$$

Similarities between LSTMs and GRUs

- *It is easy to notice similarities between the LSTM unit and the GRU*



(a) Long Short-Term Memory (b) Gated Recurrent Unit

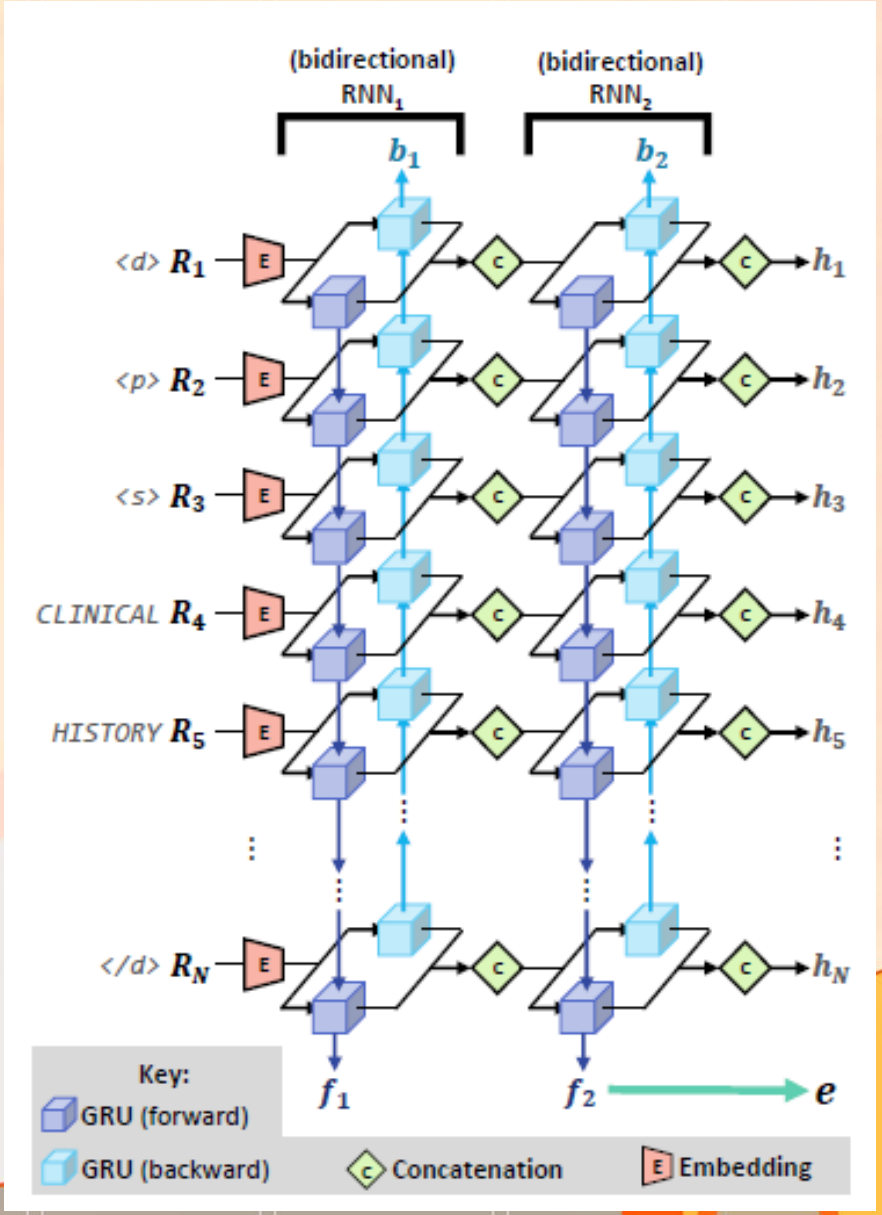
- *The most prominent feature shared between these units is the additive component of their update from t to t + 1, which is lacking in the traditional recurrent unit. The traditional recurrent unit always replaces the activation, or the content of a unit with a new value computed from the current input and the previous hidden state.*
- *Both LSTM unit and GRU keep the existing content, and add the new content on top of it*

Using GRUs

Goodwin and Harabagiu 2017

we implemented an Extractor using a deep neural architecture which relies

on five neural layers to produce **feature vectors** for each word in a medical report ($h_1..h_N$) as well as a feature vector characterizing the entire report (**e**).

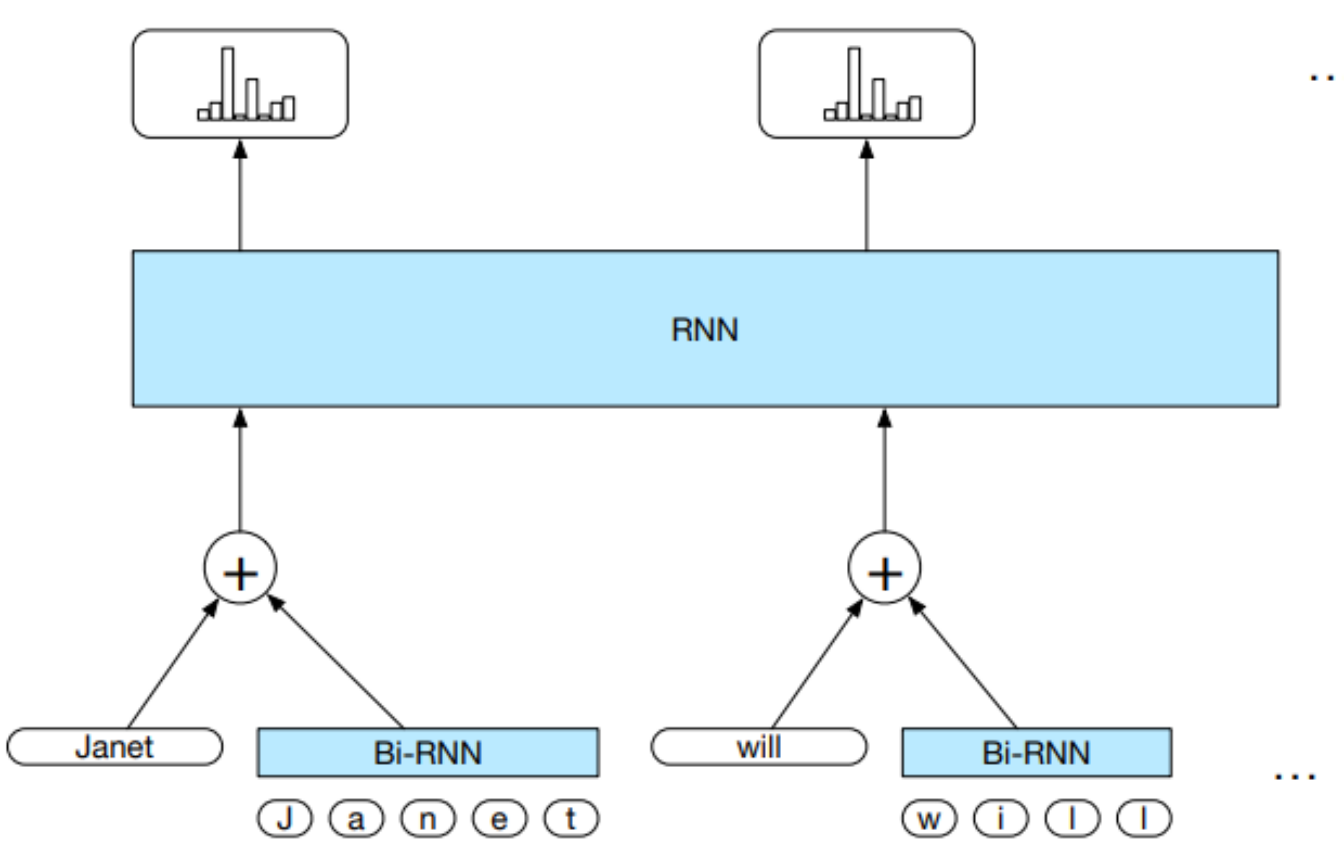


Word or Character Embeddings as inputs

- *We have assumed that the inputs to our sequential networks would be either pretrained or trained word embeddings.*
- *Word-based embeddings are great at finding distributional (syntactic and semantic) similarity between words. However, there are significant issues with any solely word-based approach:*
  - *For some languages and applications, the lexicon is simply too large to practically represent every possible word as an embedding. Some means of composing words from smaller bits is needed.*
  - *No matter how large the lexicon, we will always encounter unknown words due to new words entering the language, misspellings and borrowings from other languages.*
  - *Morphological information, below the word level, is clearly an important source of information for many applications. Word-based methods are blind to such regularities.*

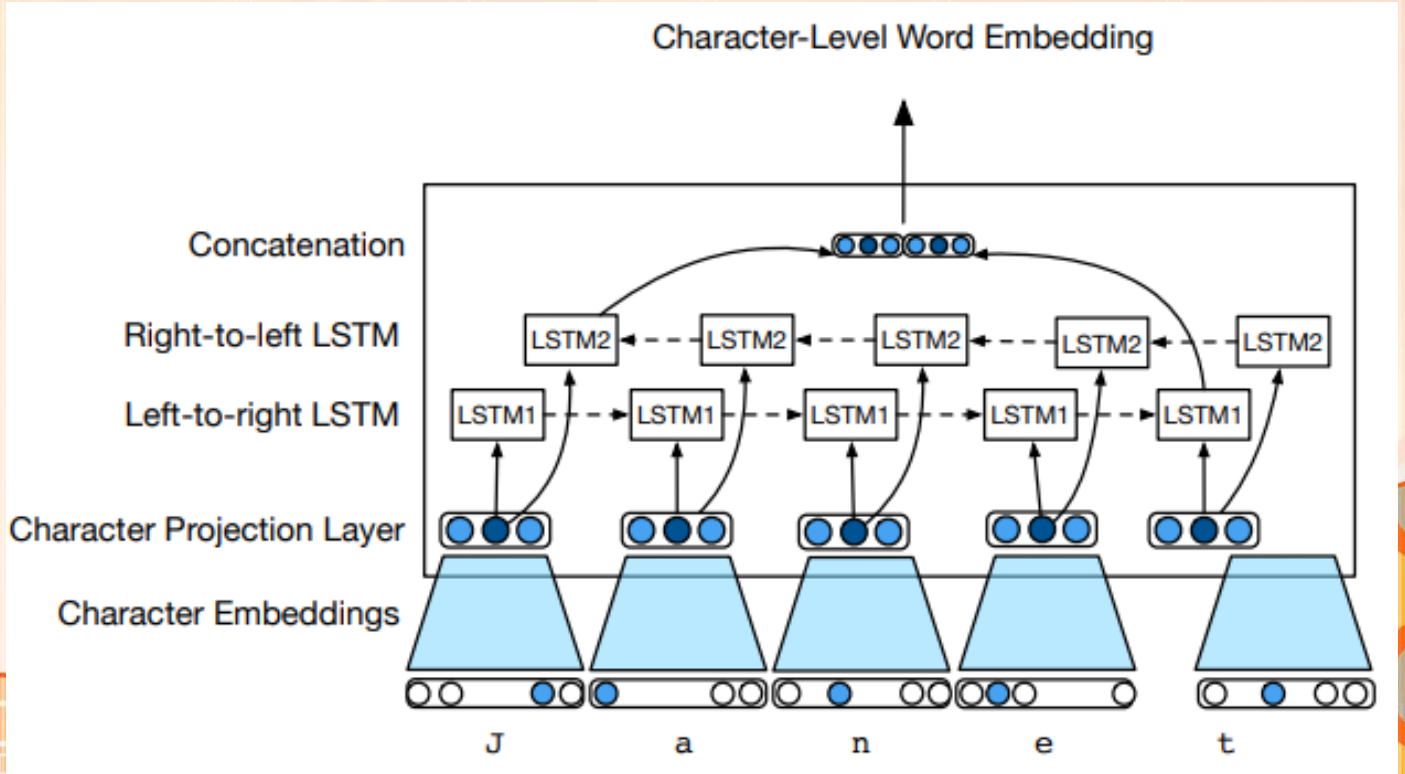
Overcoming issues related to word embeddings

- *by augmenting our input word representations with embeddings derived from the characters that make up the words.*
- *Sequential model for part-of-speech-tagging:*



Character-level word embeddings

- *The character sequence for each word in the input is run through a bidirectional RNN consisting of two independent RNNs — one that processes the sequence left-to-right and the other right-to-left.*
- *the final hidden states of the left-to-right and right-to-left networks are concatenated to represent the composite character-level representation of each word.*
- *Critically, these character embeddings are trained in the context of the overall task; the loss from the part-of-speech softmax layer is propagated all the way back to the character embeddings.*



Summary

- *Introduced parts-of-speech and part-of-speech tagging:*

• Two common approaches to sequence modeling are a generative approach, HMM tagging, and a discriminative approach, MEMM tagging.

• The probabilities in HMM taggers are estimated by maximum likelihood estimation on tag-labeled training corpora. The Viterbi algorithm is used for decoding, finding the most likely tag sequence.

• Beam search is a variant of Viterbi decoding that maintains only a fraction of high scoring states rather than all states during decoding.

• Maximum entropy Markov model or MEMM taggers train logistic regression models to pick the best tag given an observation word and its context and the previous tags, and then use Viterbi to choose the best sequence of tags.

• Modern taggers are generally run bidirectionally.