Recurrent Neural Networks (RNNs) and Long Short Term Memory Networks (LSTMs)

- **RNNs**
- **LSTMs**

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs)

- Language is an inherently **temporal phenomenon**.
  - Language is a **sequence of events over time**.
- This **temporal nature** is reflected in some language processing algorithms.
  - HMM part-of-speech tagging proceeds through the input a word at a time.
  - Text classification tasks don’t have this temporal nature. They assume simultaneous access to all aspects of their input.
- The **feedforward networks** also assumed simultaneous access, although they also had a simple model for time.
  - Feedforward networks to language modeling look at a **fixed-size window of words**, and then sliding this window over the input.
- **Recurrent Neural Networks (RNNs)**, and their variants like **LSTMs** are ***deep learning architectures*** that offers an *alternative way of representing time*.

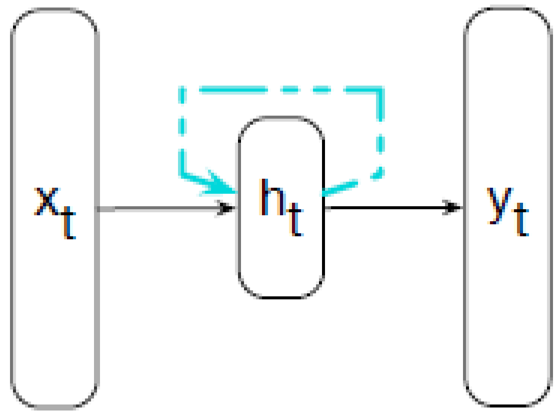
Recurrent Neural Networks (RNNs)

- **RNNs** have a mechanism that deals directly with the sequential nature of language, allowing them to handle the temporal nature of language without the use of arbitrary fixed-sized windows.
- The **recurrent network** represents the prior context, in its **recurrent connections**, allowing the model's decision to depend on hundreds of words in the past.
- A **recurrent neural network (RNN)** is ***any network that contains a cycle within its network connections***,
  - An input value of some unit is depends on its own earlier outputs as an input.

Recurrent Neural Networks (RNNs)

simple recurrent neural network

- **Simple recurrent neural network**
- The **hidden layer** includes a **recurrent connection** as part of its input.
  - 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 time step.



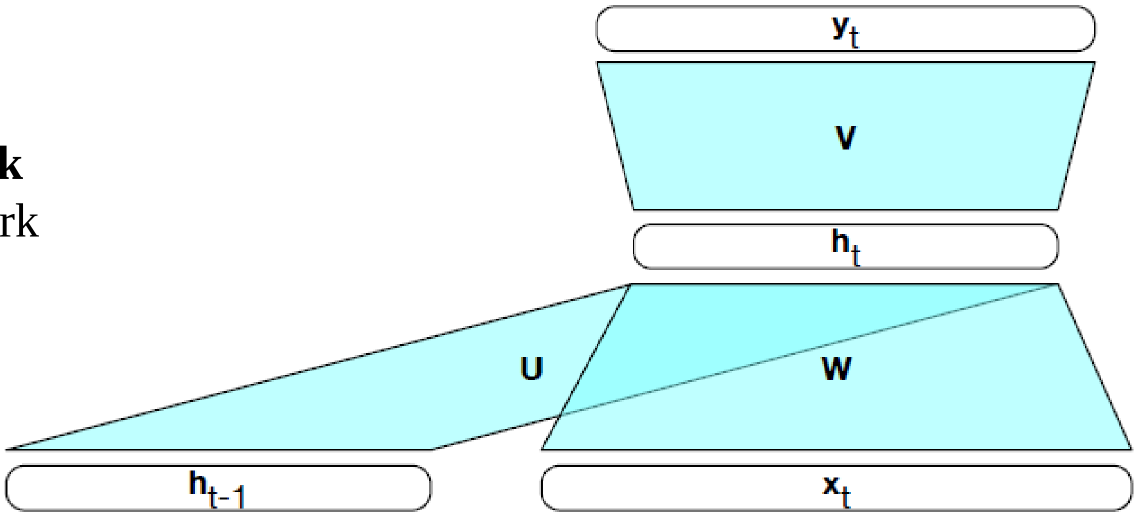
- The ***hidden layer from the previous time step*** provides a form of **memory**, or **context**, that encodes earlier processing and decisions can be made at later points in time.
  - This approach does not impose a fixed-length limit on this prior context.
  - The context embodied in the previous hidden layer can include information extending back to the beginning of the sequence

Recurrent Neural Networks (RNNs)

Inference in RNNs

- **Forward inference** (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.

Simple recurrent neural network
illustrated as a feedforward network



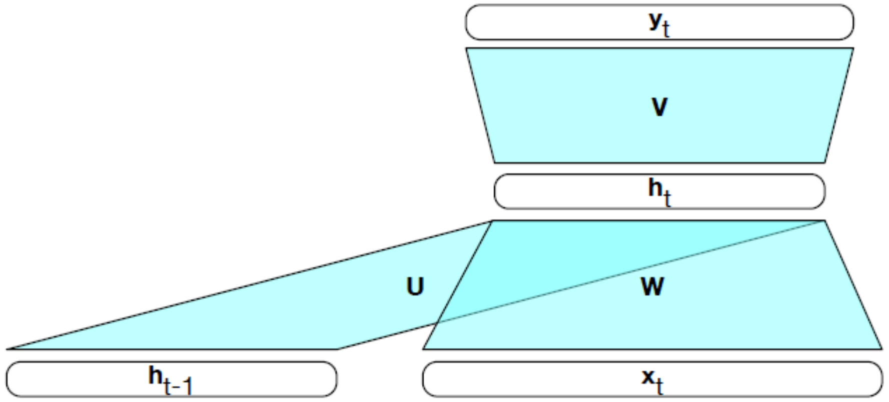
- Once we have the values for the hidden layer, we proceed with the usual computation to generate the output vector.

$$
\begin{aligned}
\mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \\
\mathbf{y}_t &= f(\mathbf{V}\mathbf{h}_t)
\end{aligned}
$$

Recurrent Neural Networks (RNNs)

Inference in RNNs

```
function FORWARDRNN(x, network) returns output sequence y
    $h_0 \leftarrow 0$
    for $i \leftarrow 1$ to LENGTH(x) do
        $h_i \leftarrow g(Uh_{i-1} + Wx_i)$
        $y_i \leftarrow f(Vh_i)$
    return y
```

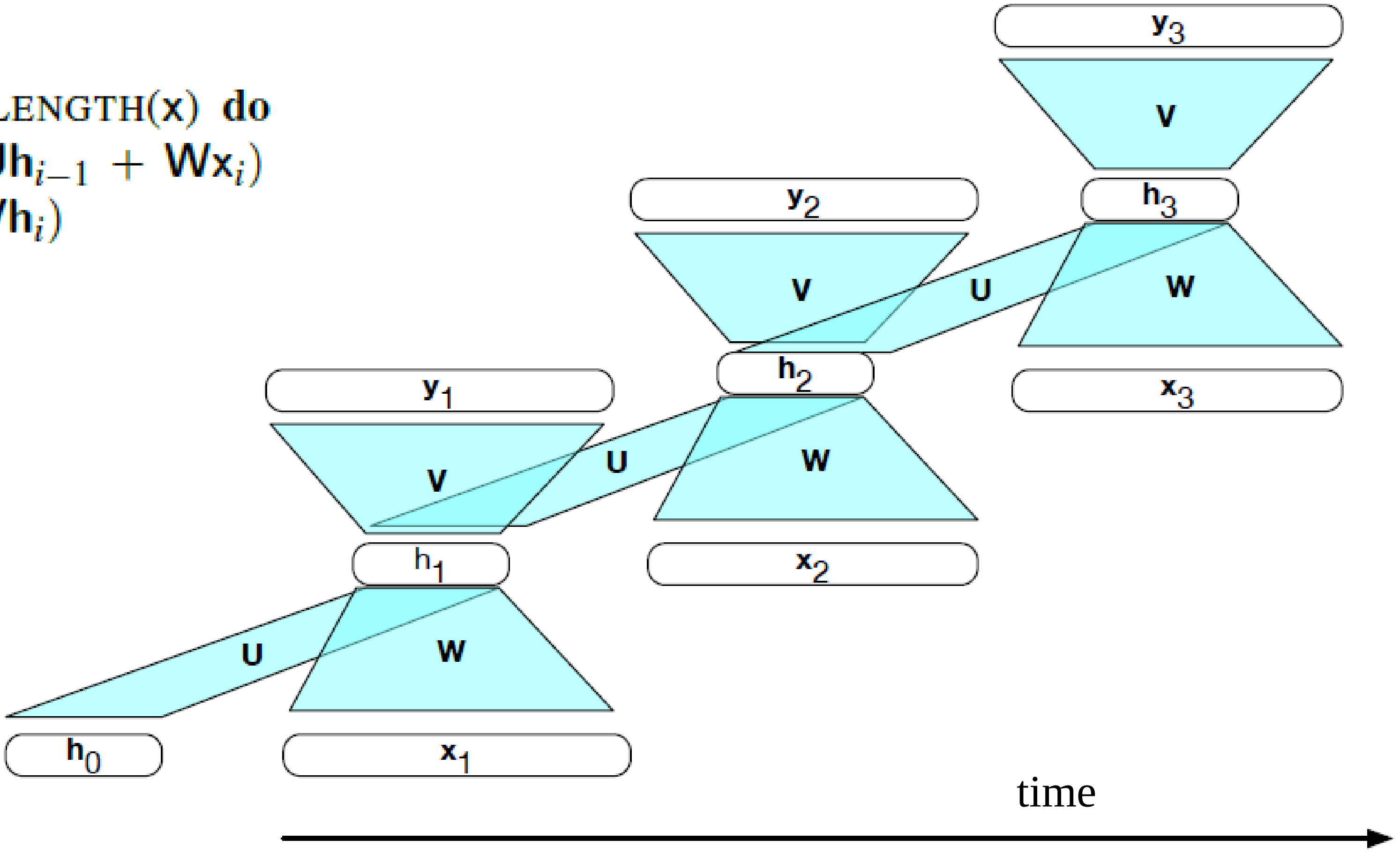


- The matrices U, V and W are shared across time, while new values for h and y are calculated with each time step.

Recurrent Neural Networks (RNNs)

Inference in RNNs

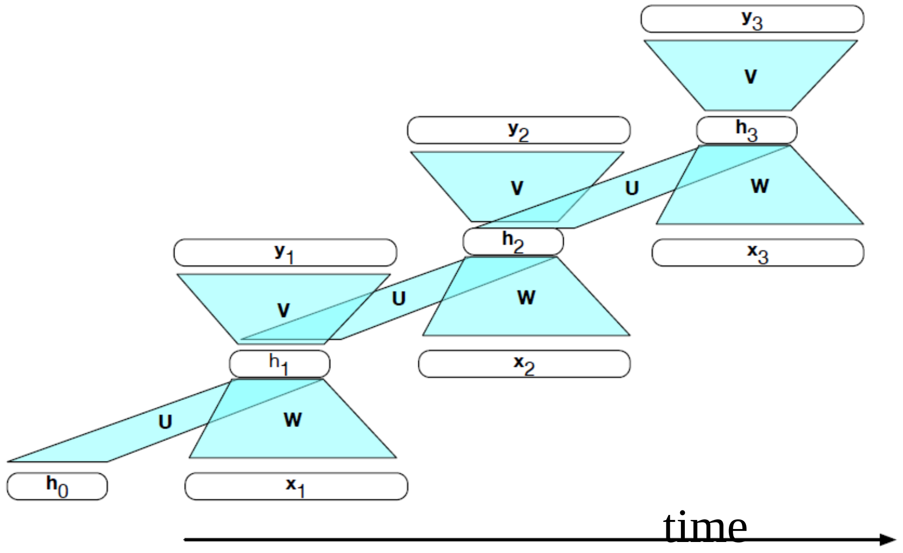
```
function FORWARDRNN($\mathbf{x}$, $network$)

    $\mathbf{h}_0 \leftarrow 0$
    for $i \leftarrow 1$ to LENGTH($\mathbf{x}$) do
        $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
        $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
    return $y$
```



Recurrent Neural Networks (RNNs)

Training RNNs

- Fortunately, with modern computational frameworks and adequate computing resources, there is no need for a specialized approach to training RNNs.
- Enrolling a recurrent network into a feedforward computational graph eliminates any explicit recurrences, allowing the network weights to be trained directly.
- When presented with a specific input sequence, we can generate an unrolled feedforward network specific to that input, and use that graph to perform forward inference or training via ordinary backpropagation.



Recurrent Neural Networks (RNNs)

- ***RNNs as Language Models***

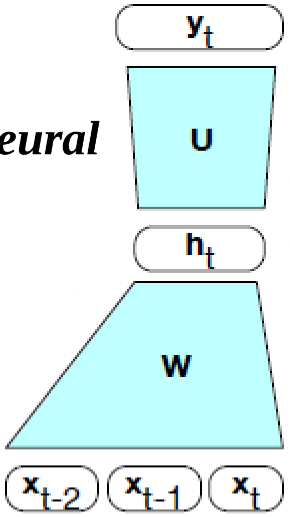
RNNs as Language Models

- **Language models** predict the next word in a sequence given some preceding context.
- Language models can assign ***a conditional probability to every possible next word***.
- They can also assign ***probabilities to entire sequences*** by combining these conditional probabilities with the chain rule:

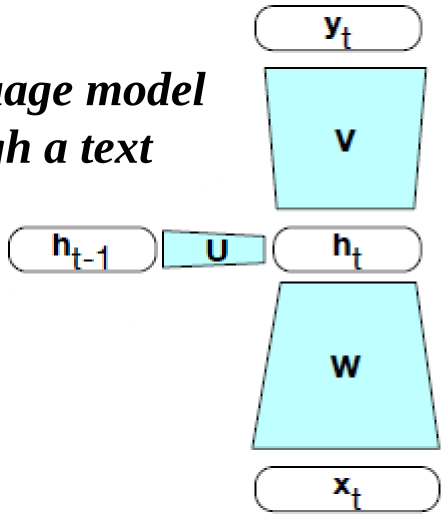
$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{<i})$$

RNNs as Language Models

a feedforward neural language model



an RNN language model moving through a text



- The **n-gram language models** compute the probability of a word given counts of its occurrence with the n-1 prior words. The ***context size is n-1***.
- For the **feedforward language models**, the ***context size is the window size***.
- **RNNs** thus ***don't have the limited context problem***, since the hidden state can in principle represent information about all of the preceding words.
  - RNN language models process the input sequence one word at a time, attempting to predict the next word from the current word and the previous hidden state.

RNNs as Language Models

Forward Inference

- **Forward inference** is the task, given an input, of running a forward pass on the network to produce a probability distribution over possible outputs (next words).
- The **input sequence** $X=[x_1,\ldots,x_N]$ consists of a series of words each represented as a ***one-hot vector*** of size $|V|\times 1$,
- The **output prediction**, y, is a vector representing a probability distribution over the vocabulary.
- At each step, the model uses the word embedding matrix E to retrieve the embedding for the current word, and then combines it with the hidden layer from the previous step to compute a new hidden layer.
- This hidden layer is then used to generate an output layer which is passed through a ***softmax*** layer to generate a probability distribution over the entire vocabulary.

RNNs as Language Models

Forward Inference

- At time t:

$$\begin{aligned} \mathbf{e}_t &= \mathbf{E}\mathbf{x}_t \\ \mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \\ \mathbf{y}_t &= \text{softmax}(\mathbf{V}\mathbf{h}_t) \end{aligned}$$

- E is embeddings (size: $d\times|V|$), x_t is one-hot vector (size: $|V|\times 1$) for the word w_t in the input sequence, e_t is the embedding (size: $d\times 1$) of the word w_t.
- h_t is the hidden layer output at time t (size: $d_h\times 1$), W is the weight matrix for the input word (size: $d_h\times d$), U is the weight matrix for the context (size: $d_h\times d_h$).
- V is the weight matrix for the output layer (size: $|V|\times d_h$).
- Vh_t can be thought of as a set of scores over the vocabulary given the evidence provided in h_t.
- y_t : Passing these scores through the softmax normalizes the scores into a probability distribution over the vocabulary.

RNNs as Language Models

Forward Inference

- At time t:

$$\begin{aligned} \mathbf{e}_t &= \mathbf{E}\mathbf{x}_t \\ \mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \\ \mathbf{y}_t &= \text{softmax}(\mathbf{V}\mathbf{h}_t) \end{aligned}$$

- The probability that a particular word k in the vocabulary is the next word at time t is represented by $y_t[k]$, the kth component of y_t :

$$P(w_{t+1} = k | w_1, \ldots, w_t) = \mathbf{y}_t[k]$$

- The probability of an entire sequence is just the product of the probabilities of each item in the sequence, where $y_i[w_i]$ is the probability of the true word w_i at time step i.

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{1:i-1}) = \prod_{i=1}^{n} \mathbf{y}_i[w_i]$$

RNNs as Language Models

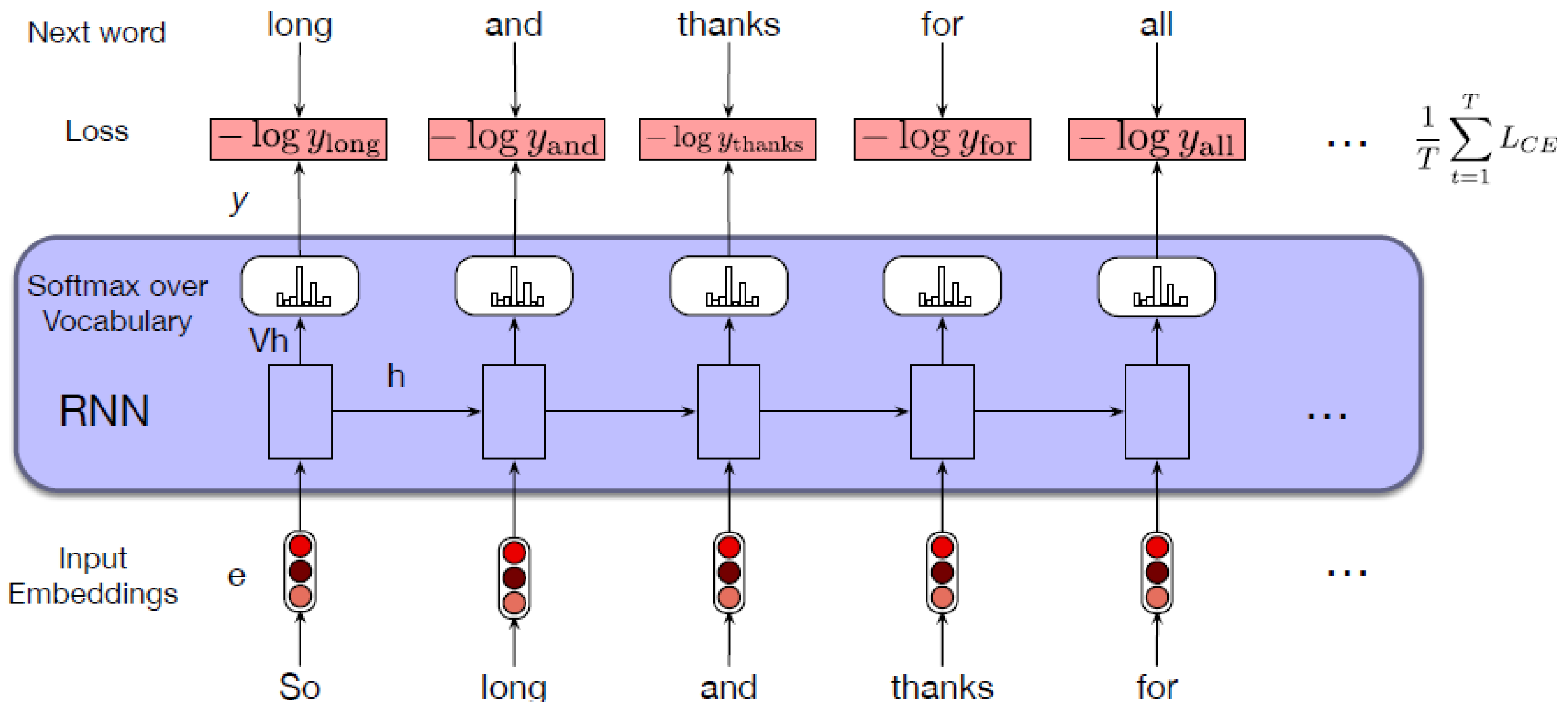
Training an RNN language model

- **self-supervision**: To train an RNN as a language model, we take a corpus of text as training material and at each time step t ask the model to predict the next word.
- We train the model to minimize the error in predicting the true next word in the training sequence, using cross-entropy as the loss function.
  - Recall that the cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution.
  - The correct distribution y_t comes from knowing the next word.
  - At time t the CE loss is the negative log probability the model assigns to the next word in the training sequence.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}]$$

RNNs as Language Models

Training an RNN language model



The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent.

- At each word position t, the model takes the correct sequence of tokens $w_{1:t}$, and computes a probability distribution over possible next words to compute model's loss for next token w_{t+1}.
  - Then we ignore what the model predicted for the next word and instead use the correct sequence of tokens $w_{1:t+1}$ to estimate the probability of token w_{t+2}.
  - The idea that the model use the correct history sequence to predict the next word (rather than feeding the model its best case from the previous time step) is **called teacher forcing**.

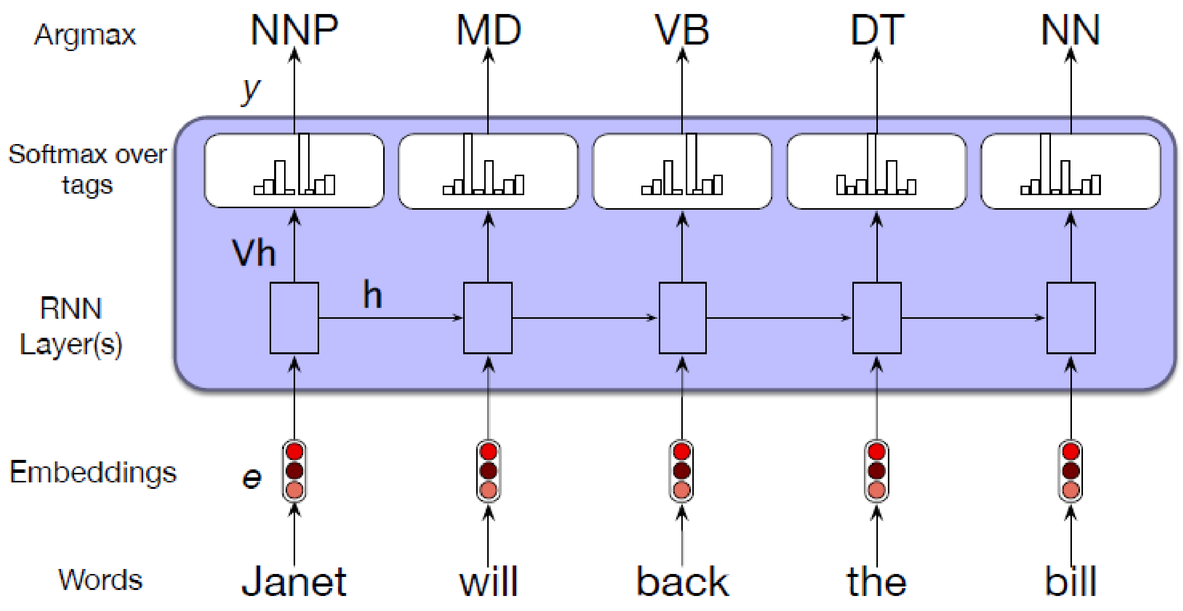
Recurrent Neural Networks (RNNs)

- ***RNNs for other NLP tasks***

RNNs for Other NLP Tasks

Sequence Labeling

- In **sequence labeling**, the network's task is to assign a label chosen from a small fixed set of labels to each element of a sequence, like the ***part-of-speech tagging.***
- ***Part-of-speech tagging as sequence labeling with a simple RNN***.
  - Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.



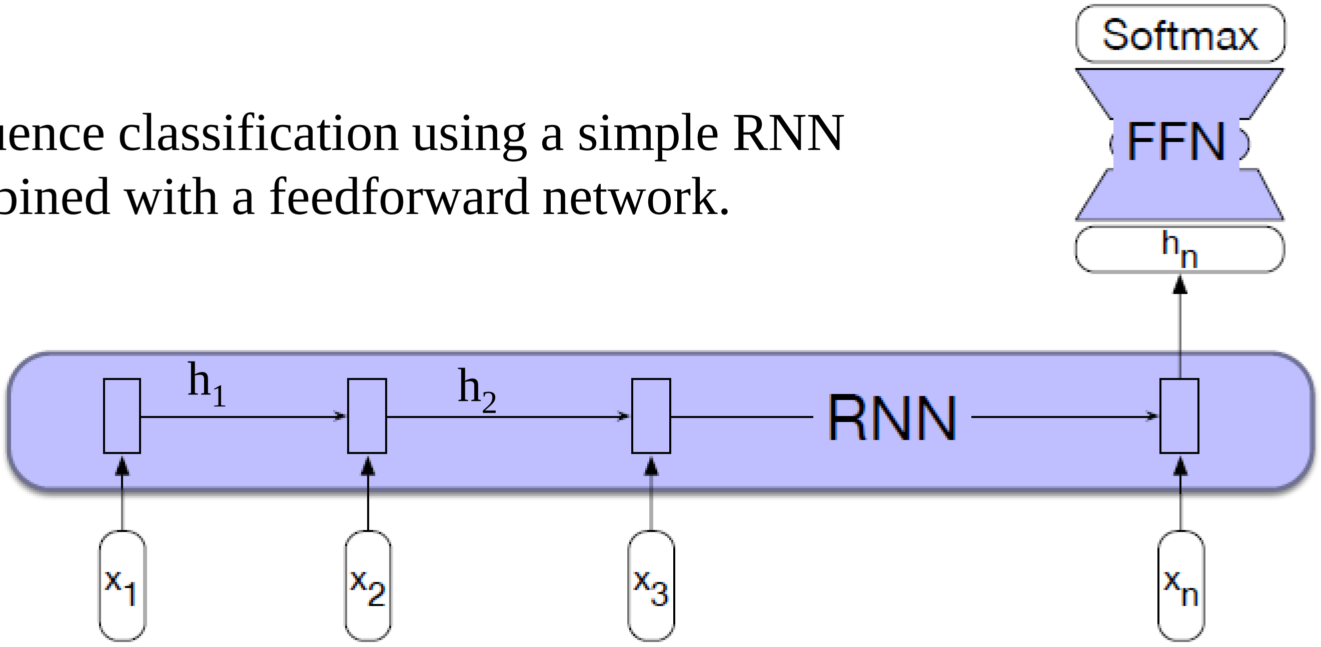
This RNN represents an **unrolled simple recurrent network** consisting of an input layer, hidden layer, and output layer at each time step, as well as the shared U, V and W weight matrices that comprise the network.

RNNs for Other NLP Tasks

RNNs for Sequence Classification – Text Classification

- RNNs can **classify entire sequences** rather than the tokens within them.
  - The text to be classified is passed through the RNN a word at a time generating a new hidden layer at each time step.
  - The hidden layer for the last token of the text, h_n, to constitute a compressed representation of the entire sequence.
  - h_n is given to a feedforward network that chooses a class via a softmax function.

Sequence classification using a simple RNN combined with a feedforward network.



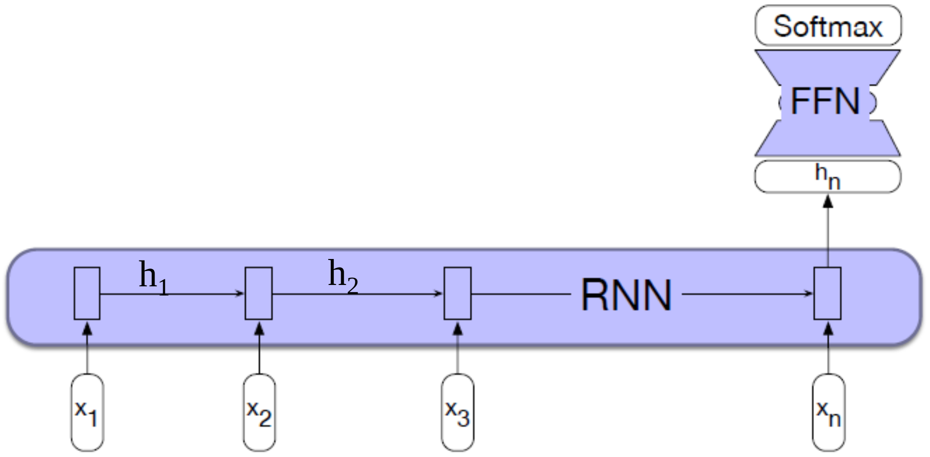
RNNs for Other NLP Tasks

RNNs for Sequence Classification – Text Classification

- *No need intermediate outputs for the words* in the sequence preceding the last element.
  - Therefore, there are no loss terms associated with those elements.
  - Instead, the loss function used to train the weights in the network is based entirely on the final text classification task.
  - The output from the softmax output from the feedforward classifier together with a cross-entropy loss drives the training.
  - The error signal from the classification is backpropagated all the way through the weights in the feedforward classifier through, and then through to the three sets of weights in the RNN

- Instead of using just last token h_n to represent the whole pooling sequence, we can use a ***pooling function*** of all hidden states h_i.
- A representation that pools all the n hidden states by taking their element-wise mean:

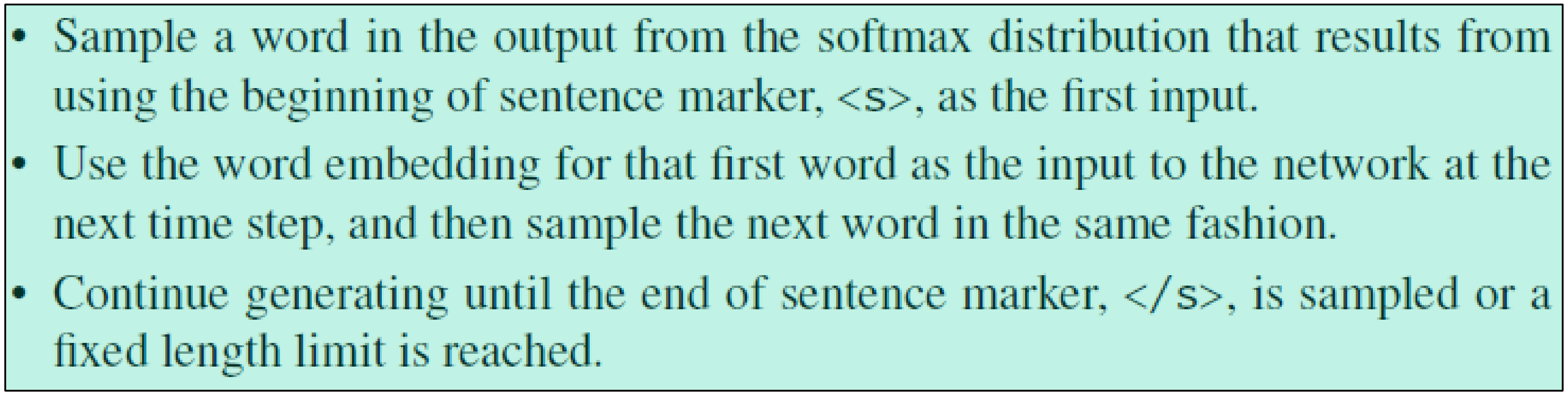
$$\mathbf{h}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{h}_i$$



RNNs for Other NLP Tasks

Generation with RNN-Based Language Models

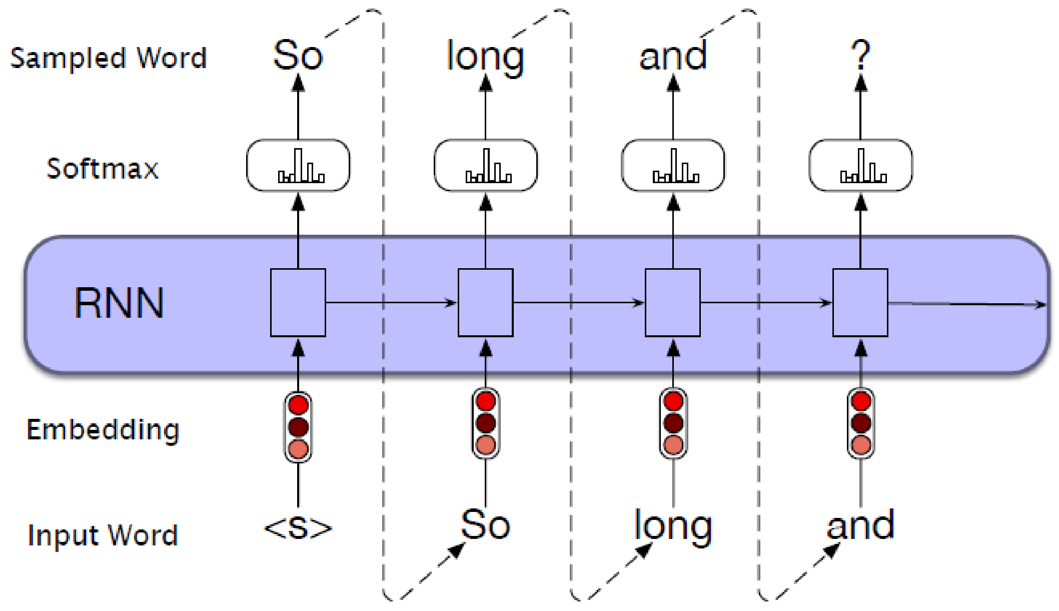
- ***RNN-based language models can also be used to generate text*.**
  - Text generation is of enormous practical importance any task where a system needs to produce text such as question answering, machine translation, text summarization.
  - Text generation constitute a new area of AI that is often called **generative AI**.
- Using a language model to incrementally generate words by repeatedly sampling the next word conditioned on our previous choices is called **autoregressive generation**.



- Sample a word in the output from the softmax distribution that results from using the beginning of sentence marker, `<s>`, as the first input.
- Use the word embedding for that first word as the input to the network at the next time step, and then sample the next word in the same fashion.
- Continue generating until the end of sentence marker, `</s>`, is sampled or a fixed length limit is reached.

RNNs for Other NLP Tasks

Generation with RNN-Based Language Models



Autoregressive generation with an RNN-based neural language model.

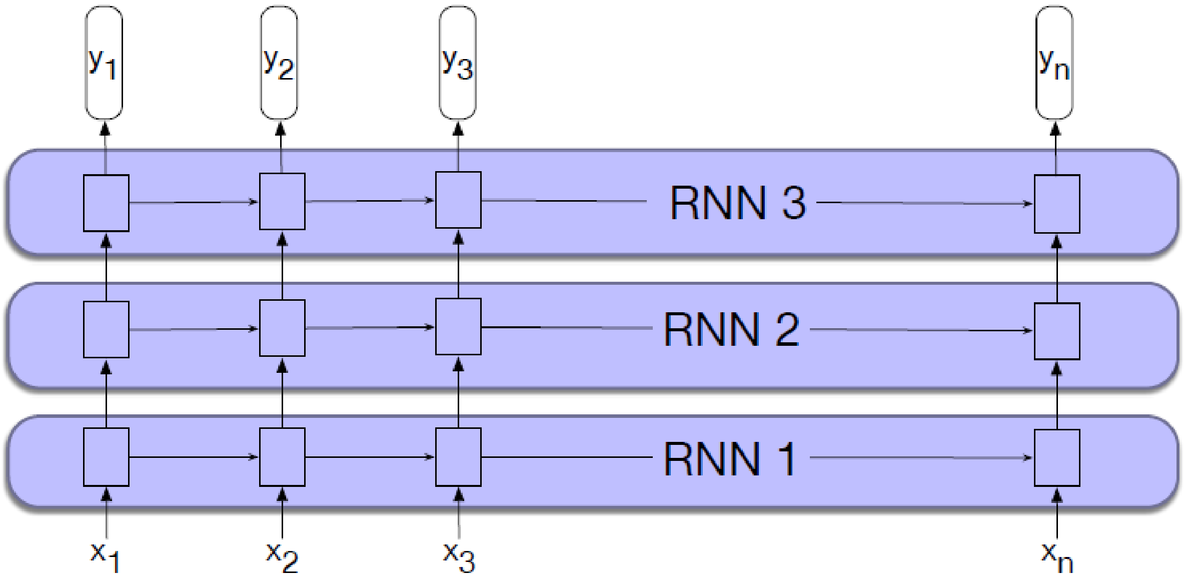
- Autoregressive generation architecture is applicable to many NLp applications such as machine translation, summarization, and question answering.
  - The key is to prime the generation component with an appropriate context.
  - Instead of simply using <s> to get things started we can provide a richer task-appropriate context; for translation the context is the sentence in the source language; for summarization it’s the long text we want to summarize.

Recurrent Neural Networks (RNNs)

- ***Stacked and Bidirectional RNN architectures***

Stacked RNNs

- Recurrent networks are quite flexible.
  - The entire sequence of outputs from one RNN as an input sequence to another one.
- **Stacked RNNs consist of multiple networks where the output of one layer serves as the input to a subsequent layer.**

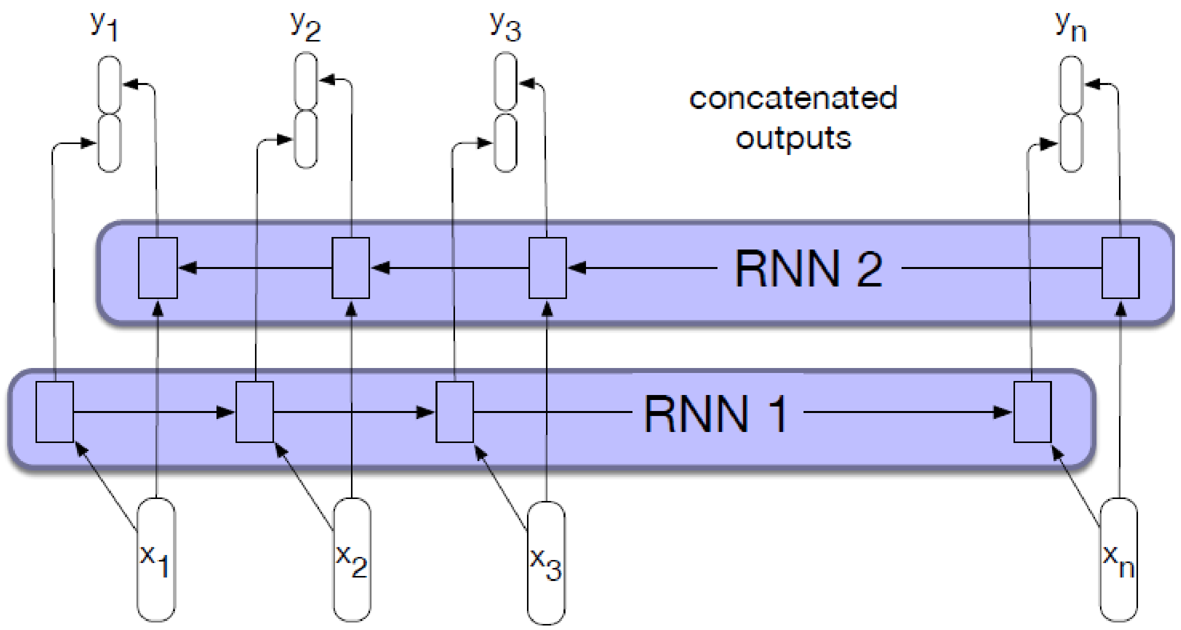


Stacked recurrent networks.

- The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.
- Stacked RNNs generally outperform single-layer networks.
- As the number of stacks is increased the training costs rise quickly.

Bidirectional RNNs

- The RNN uses information from the **left context** to make its predictions at time t.
- But many NLP applications may require **right context** in addition to left context.
- **A bidirectional RNN combines two independent RNNs, one where the input is processed from the start to the end, and the other from the end to the start.**
  - We then concatenate the two representations computed by the networks into a single vector that captures both the left and right contexts of an input at each point in time.



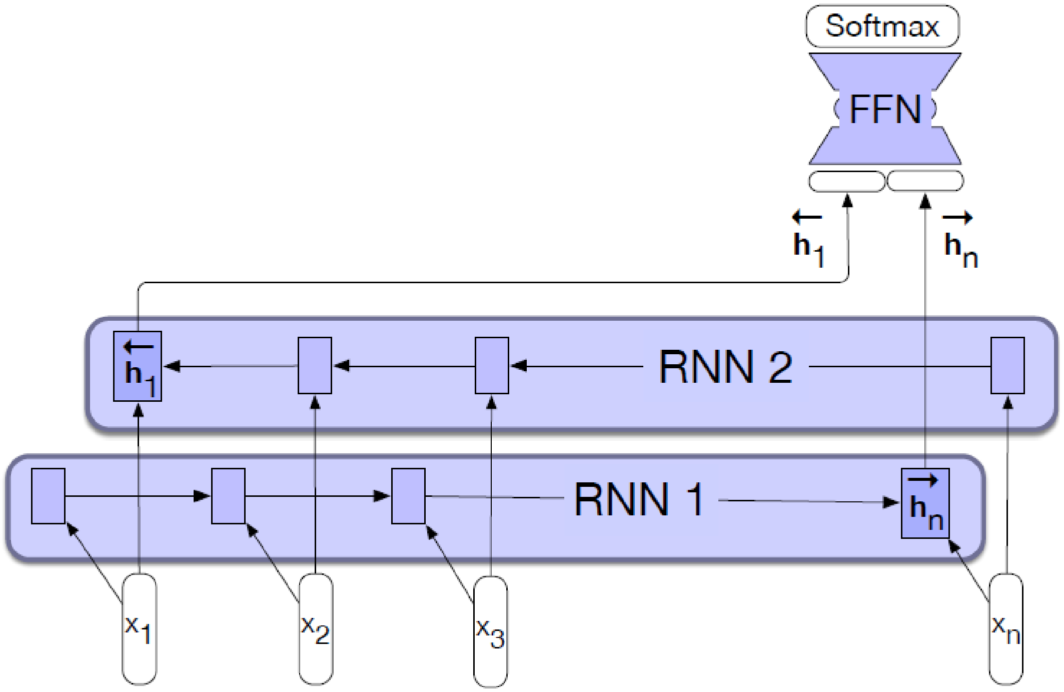
A bidirectional RNN

- Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

Bidirectional RNNs

A bidirectional 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.



LSTM (long short-term memory) Network

LSTM

shortcomings of RNNs – distant information

- ***Distant information*** is critical to many language applications.
- To train RNNs is difficult to make *use of information distant from the current point of processing*.
  - Alhough RNNs access to the entire preceding sequence, *the information encoded in hidden states tends to be fairly loca*l, more relevant to the most recent parts of the input.

Example: The flights the airline was canceling were full.

- Assigning a high probability to **was** following airline is straightforward since airline provides a strong **local context** for the singular agreement.
- However, assigning an appropriate probability to **were** is quite difficult, not only because the plural flights is quite **distant**, but also because the singular noun airline is closer in the intervening context.
- Ideally, ***a network should be able to retain the distant information about plural flights until it is needed***, while still processing the intermediate parts of the sequence correctly.

LSTM

shortcomings of RNNs – distant information

Reasons for the inability of RNNs to carry forward critical information

1. The weights in the hidden layer are being asked to perform two tasks simultaneously:
   - provide information useful for the current decision, and
   - updating and carrying forward information required for future decisions.
2. Training RNNs needs to backpropagate the error signal back through time.
   - The hidden layer at time t contributes to the loss at the next time step since it takes part in that calculation.
   - During the ***backward pass of training***, the hidden layers are subject to repeated multiplications, as determined by the length of the sequence.
   - A frequent result of this process is that the gradients are eventually driven to zero, a situation called the **vanishing gradients problem**.

- More complex network architectures are designed to *explicitly manage the task of maintaining relevant context over time*,
  - by enabling the network to learn to ***forget information that is no longer needed*** and
  - to ***remember information required for decisions still to come***.

  ➔ **LSTM**

LSTM

- The **long short-term memory (LSTM)** networks which are *extensions of RNNs* are designed ***to solve context management problem***.

- LSTMs divide the *context management problem* into two subproblems:
  - ***removing information no longer needed from the context***, and
  - ***adding information likely to be needed for later decision making***.

- LSTMs accomplish the context management problem
  - Adding an **explicit context layer** to the architecture (in addition to hidden layer), and
  - Use of specialized neural units that make use of **gates to 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 weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

LSTM
gates

The ***gates in an LSTM share a common design pattern***:

- Each consists of a **feedforward layer**, followed by a **sigmoid activation function**, followed by a **pointwise multiplication with the layer being gated**.
- The sigmoid activation function is preferred because it pushes its outputs to 0 or 1.
- A pointwise multiplication of the sigmoid function output and the layer being gated has an effect similar to that of a *binary mask of the layer*.
- Values in the layer being gated that align with values near 1 in the mask are passed through nearly unchanged; values corresponding to lower values are essentially erased.

LSTM
Gates – forget gate

- The purpose of **forget gate** is to delete information from the context that is no longer needed.
  - The forget gate computes a weighted sum of the previous state’s hidden layer and the current input and passes that through a sigmoid.
  - This mask is then multiplied element-wise ($\odot$) by the context vector to remove the information from context that is no longer required.

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t)$$
$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$$

- U_f and W_f : weights for the forget gate
- c_{t-1} : the previous context
- f_t : a binary mask for forget gate

LSTM

Gates – add gate

- The next task is to compute the actual information we need to extract from the previous hidden state and current inputs.

$$\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t)$$

- U_g and W_g : weights for possible context items
- g_t : possible context items to be added
  in the current state

- We generate the mask for the **add gate** to select the information to add to the current context. We add this to the modified context vector to get our new context vector.

$$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t)$$
$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t$$
$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$

- U_i and W_i : weights for the add gate
- i_t : a binary mask for add gate
- j_t : the new context items to be added
- c_t : the new context

LSTM

Gates – output gate

- The **output gate** is used to decide what information is required for the current hidden state.

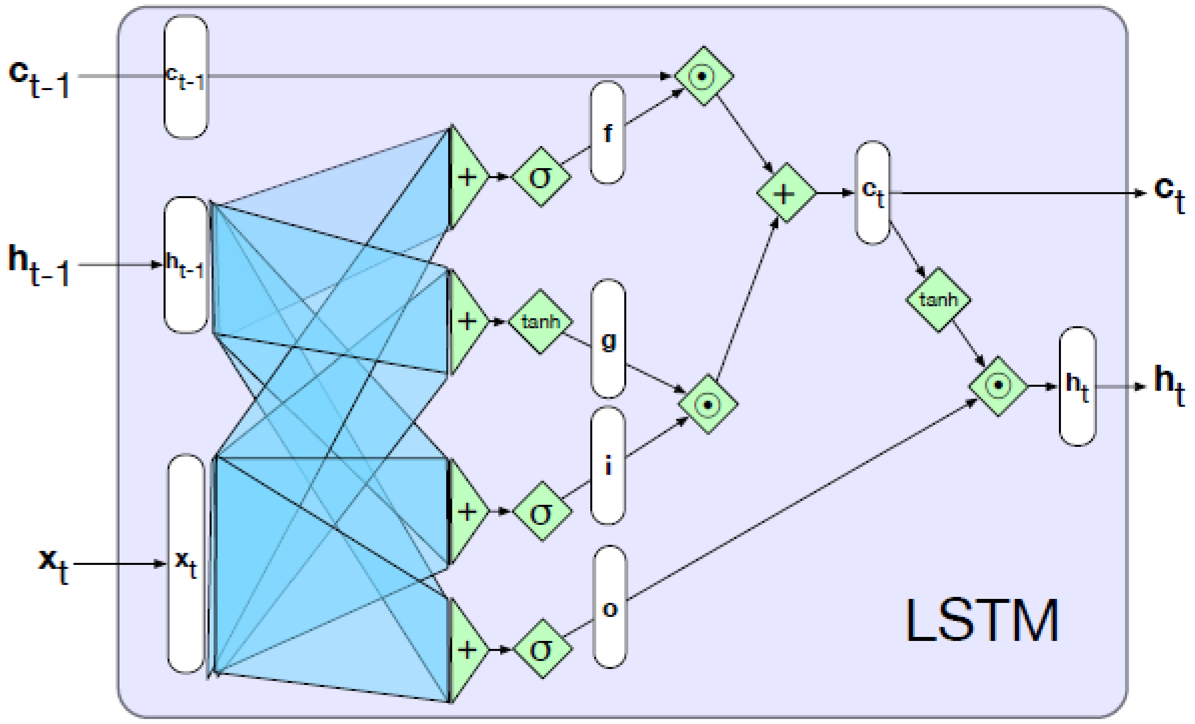
$$\mathbf{o}_t = \sigma(\mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{W}_o\mathbf{x}_t)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

- U_o and W_o : weights for output gate
- o_t : binary mask for output gate
- h_t : hidden layer at time t

LSTM

A single LSTM unit

- **A single LSTM unit displayed as a computation graph:**
  - The inputs to each unit consists of ***current input x_t, previous hidden state h_{t-1}, and previous context c_{t-1}.***
  - The outputs are a ***new hidden state h_t*** and an ***updated context c_t*** .



forget gate

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t)$$
$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$$

add gate

$$\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t)$$
$$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t)$$
$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t$$
$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$

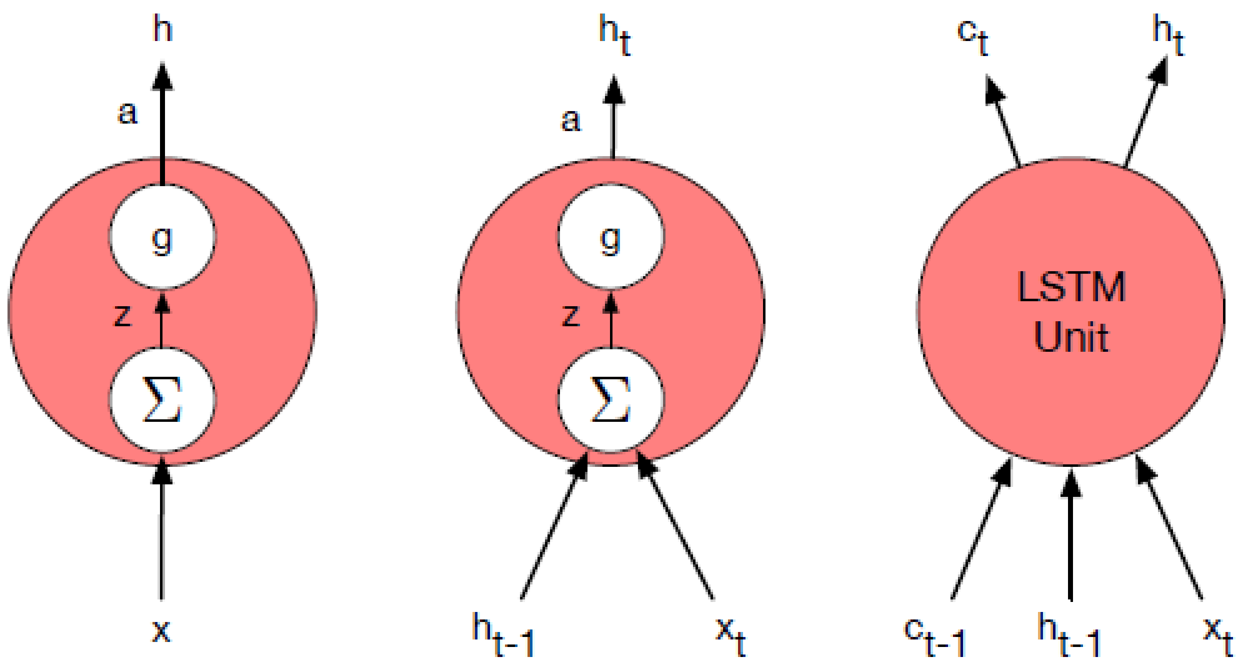
output gate

$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

LSTM

LSTM units

The **neural units used in LSTMs** are obviously much more complex than those used in basic feedforward networks, simple RNNs.

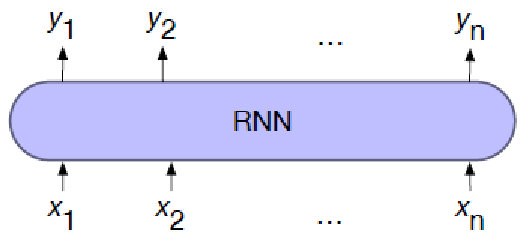


Feedforward Unit **RNN Unit** **LSTM Unit**

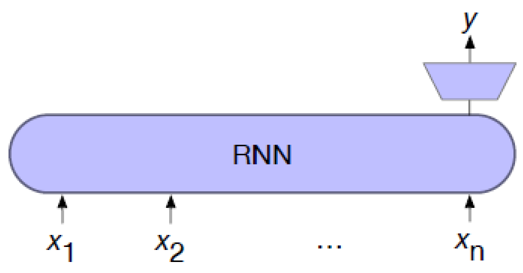
Encoder-Decoder Model with RNNs

Encoder-Decoder Model with RNNs

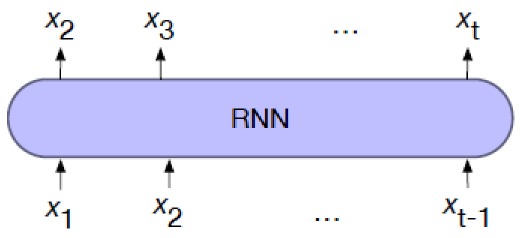
Four architectures for NLP tasks



In **sequence labeling** (POS tagging) we map each input token x_i to an output token y_i.



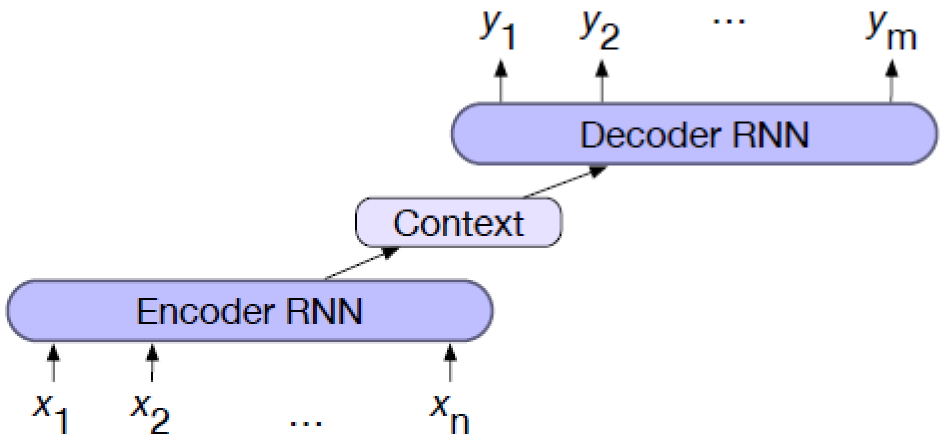
In **sequence classification** we map the entire input sequence to a single class.



In **language modeling** we output next token conditioned on previous tokens.

In **encoder-decoder** model we have two separate RNN models,

- First one maps input sequence to intermediate representation called as context, and
- Second one maps from context to output sequence.



Encoder-Decoder Model with RNNs

- The **encoder-decoder model** takes an input sequence and translates it to an output sequence that is of a different length than the input.
  - **Encoder-decoder models** are used especially for tasks like machine translation, where the input sequence and output sequence can have different lengths and the mapping between a token in the input and a token in the output can be very indirect.

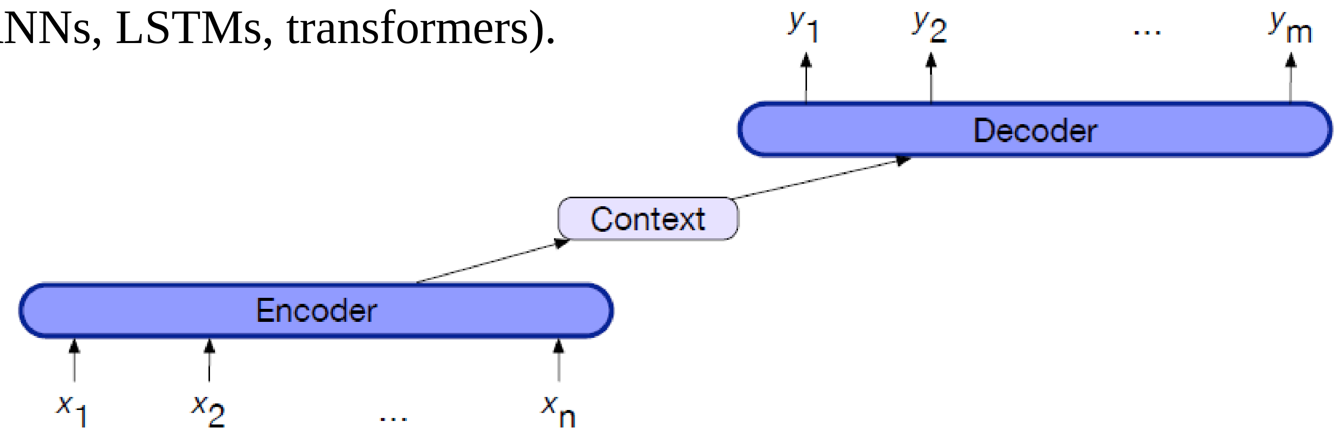
- The key idea underlying these networks is the use of an encoder network that takes an input sequence and creates a contextualized representation of it, often called the **context**.
  - This context representation is then passed to a decoder which generates a task specific output sequence.

Encoder-Decoder Model with RNNs

The encoder-decoder architecture

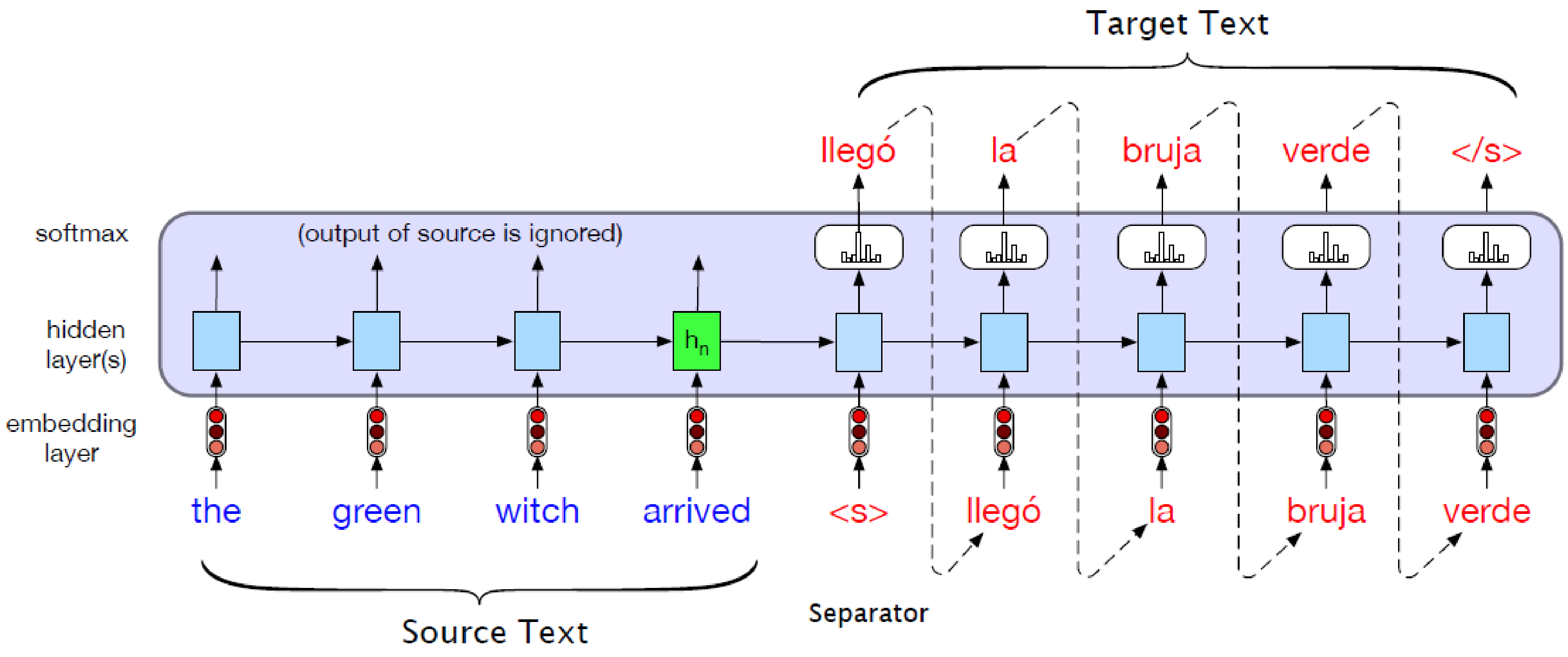
Encoder-decoder networks consist of ***three conceptual components***:

1. An **encoder** that accepts an input sequence, $x_{1:n}$, and generates a corresponding sequence of contextualized representations, $h_{1:n}$.
   - LSTMs, and transformers can all be employed as encoders.
2. A **context vector** c, which is a function of $h_{1:n}$, and conveys the essence of the input to the decoder.
3. A **decoder**, which accepts c as input and generates an arbitrary length sequence of hidden states $h_{1:m}$, from which a corresponding sequence of output states $y_{1:m}$, can be obtained.
   - Just as with encoders, decoders can be realized by any kind of sequence architecture (RNNs, LSTMs, transformers).



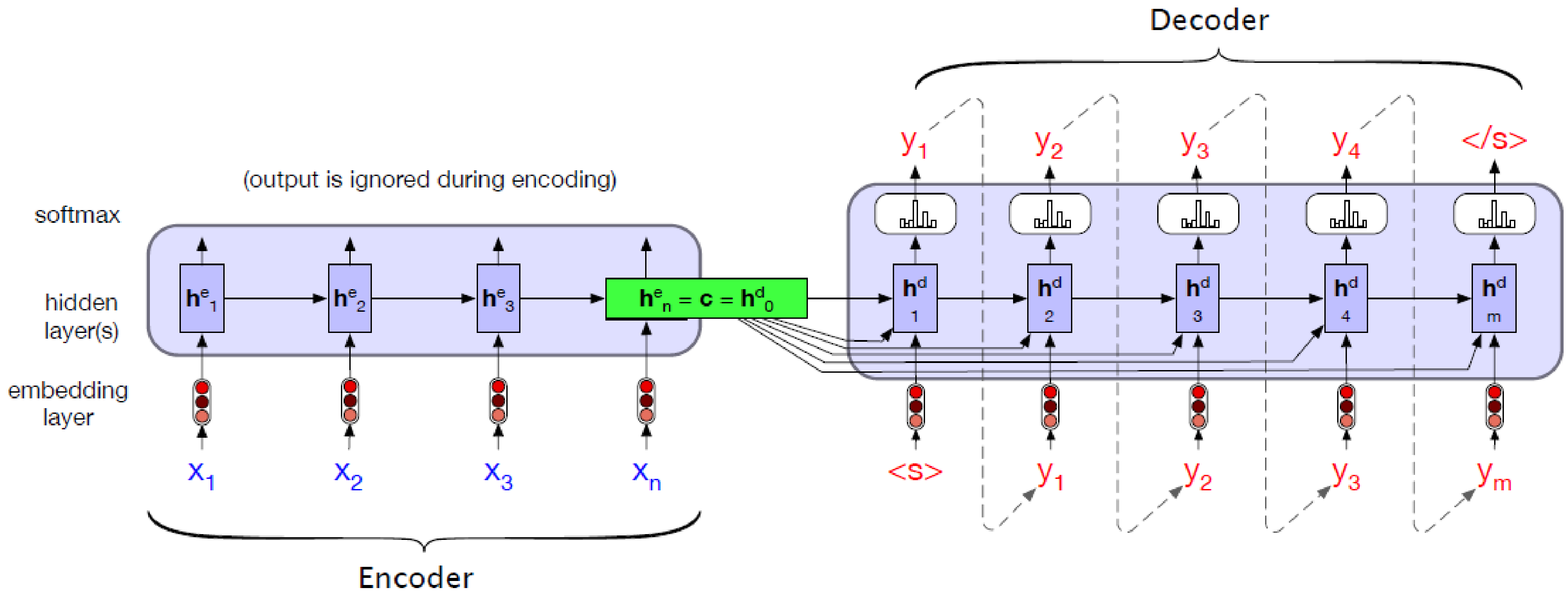
Encoder-Decoder Model with RNNs

- Translating a single sentence in the ***basic RNN version of encoder-decoder approach to machine translation.***
  - Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.



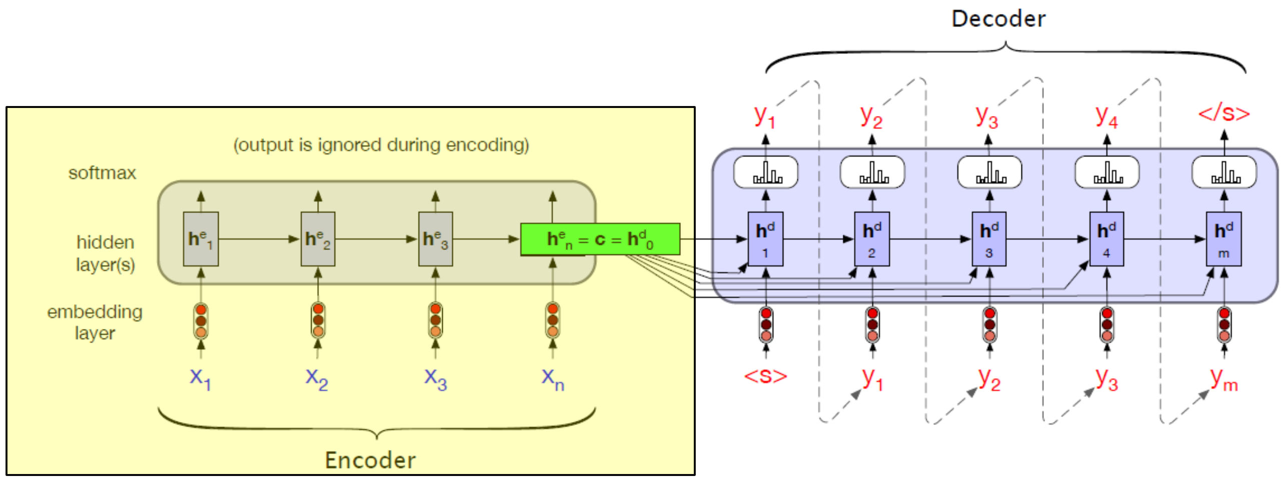
Encoder-Decoder Model with RNNs

- ***A formal version of translating a sentence in the basic RNN-based encoder-decoder architecture:***
  - The **final hidden state** of the encoder RNN $\mathbf{h}_n^e$, serves as the **context c** for the decoder in its role as $\mathbf{h}_0^d$ in the decoder RNN, and
  - The **context** is also made available to each decoder hidden state.



Encoder-Decoder Model with RNNs

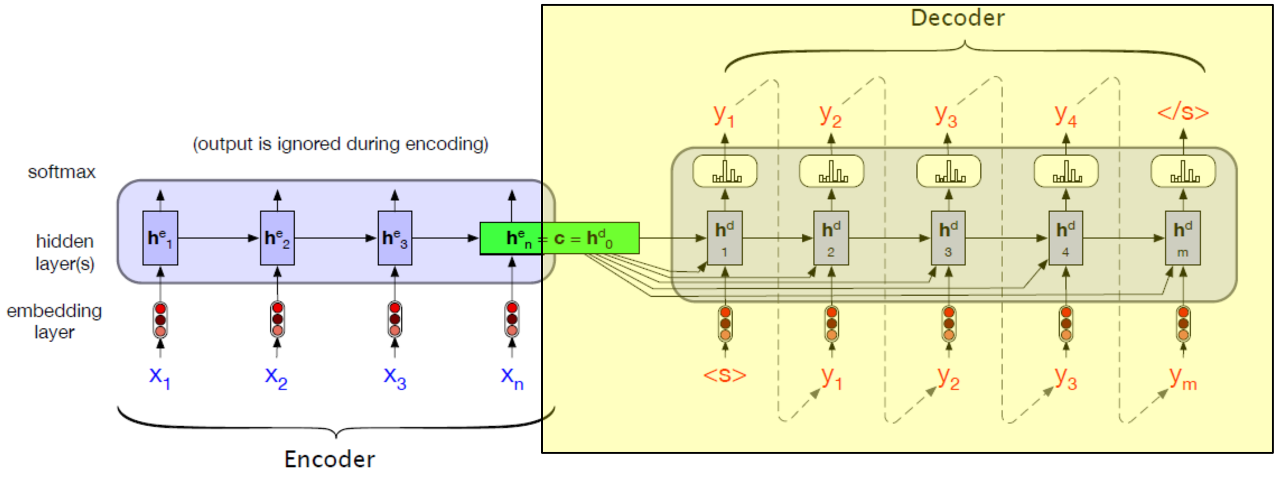
basic RNN-based encoder-decoder architecture



- The purpose of the **encoder** is to generate a contextualized representation of the input.
- This representation is embodied in the ***final hidden state of the encoder*** $\mathbf{h^e_n}$, also called c for **context**, and it is then passed to the decoder.
- Encoder can be a single network layer,
  - Stacked architectures (such as stacked biLSTMs) are widely used to represent encoders.

Encoder-Decoder Model with RNNs

basic RNN-based encoder-decoder architecture



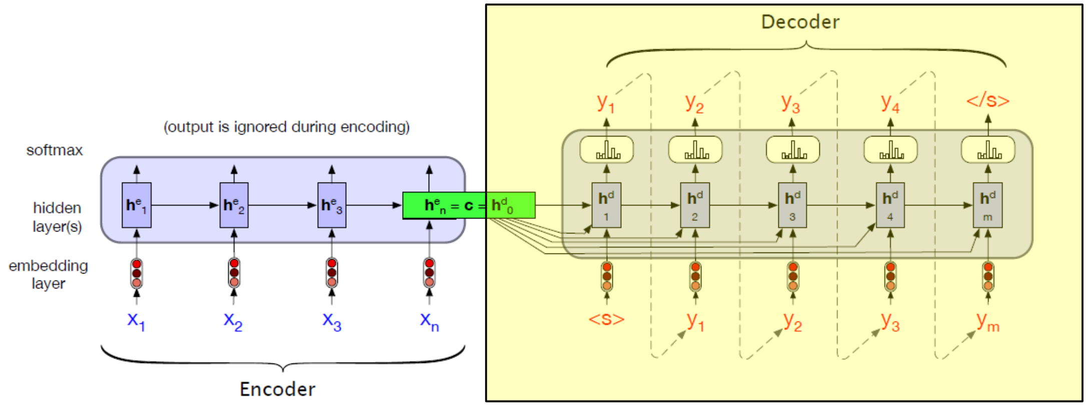
- The **decoder** network takes the context and use it just to initialize the ***first hidden state of the decoder***;
  - The first decoder RNN cell would use c as its prior hidden state $\mathbf{h_0^d}$.
  - The decoder autoregressively generates a sequence of outputs, an element at a time, until an end-of-sequence marker is generated.
  - Each hidden state is conditioned on the previous hidden state and the output generated in the previous state.

Encoder-Decoder Model with RNNs

basic RNN-based encoder-decoder architecture

Equations for the decoder, with context available at each decoding timestep.

$$
\begin{aligned}
\mathbf{c} &= \mathbf{h}_n^e \\
\mathbf{h}_0^d &= \mathbf{c} \\
\mathbf{h}_t^d &= g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \\
\mathbf{z}_t &= f(\mathbf{h}_t^d) \\
y_t &= \text{softmax}(\mathbf{z}_t)
\end{aligned}
$$

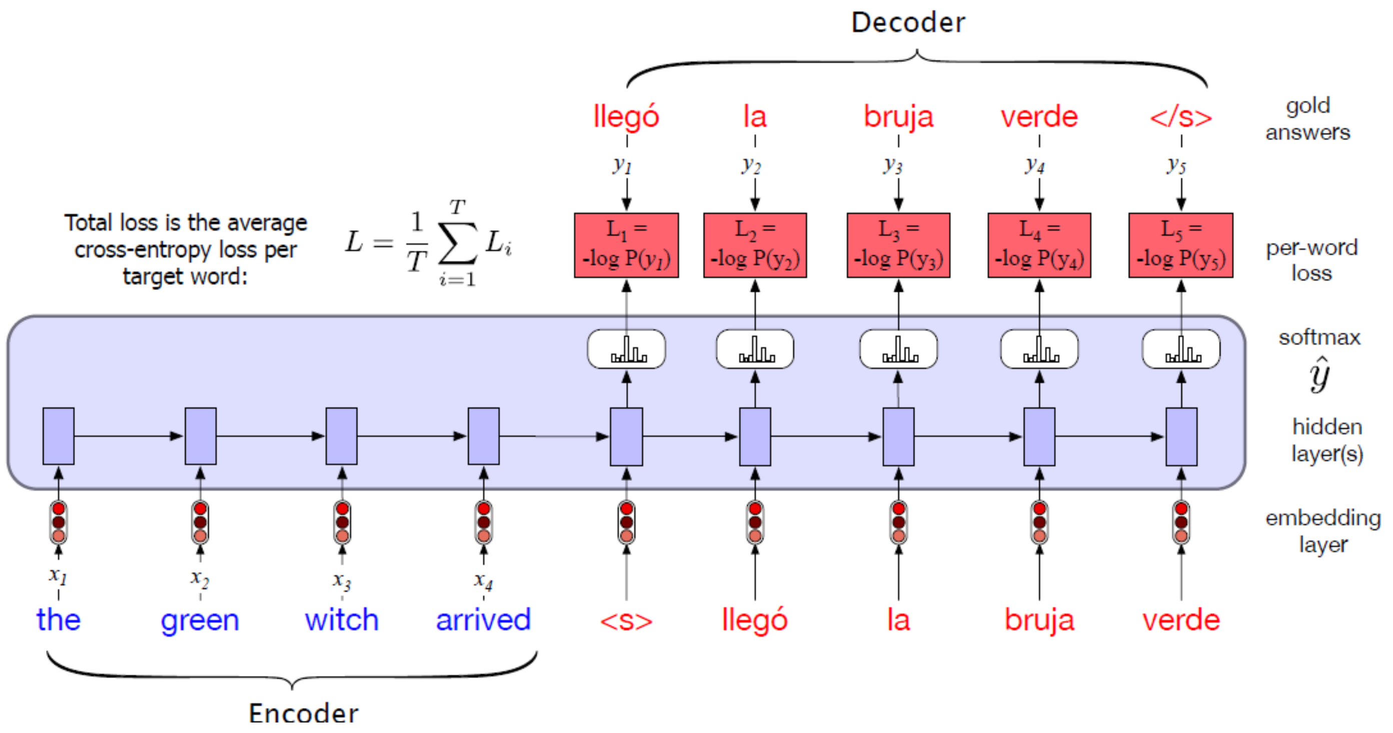


- The output y at each time step consists of a softmax computation over the set of possible outputs (the vocabulary).
- We compute the most likely output at each time step by taking the argmax over the softmax output:

$$\hat{y}_t = \text{argmax}_{w \in V} P(w | y_1 \ldots y_{t-1}, x)$$

Encoder-Decoder Model with RNNs

basic RNN-based encoder-decoder architecture - Training



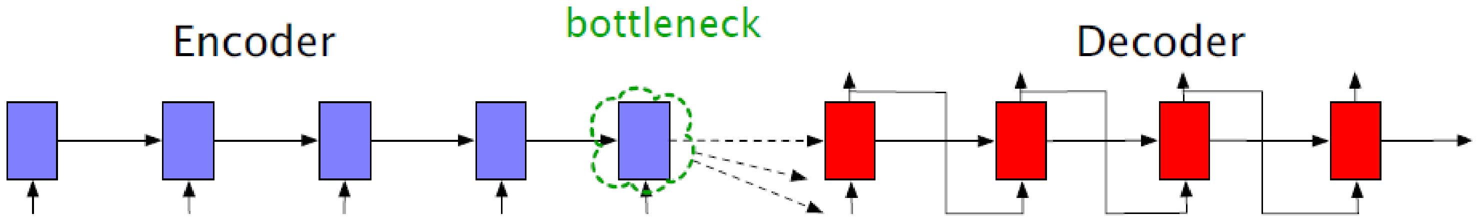
- **Training the basic RNN encoder-decoder approach to machine translation.**
  - Note that in the decoder we usually don't propagate the model's softmax outputs $\hat{\mathbf{y}}_t$, but use **teacher forcing** to force each input to the correct gold value for training.
  - We compute the softmax output distribution over ˆ $\hat{\mathbf{y}}$ in the decoder in order to compute the loss at each token, which can then be averaged to compute a loss for the sentence.

Encoder-Decoder Model with RNNs

- ***Attention***

Encoder-Decoder Model with RNNs - Attention

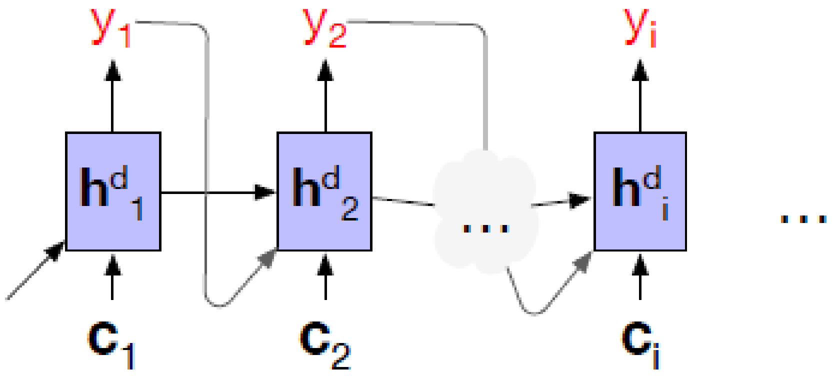
- ***Requiring the context c to be only the encoder's final hidden state*** forces all the information from the entire source sentence to pass through this representational **bottleneck**.



- The **attention mechanism** is ***a solution to the bottleneck problem***,
- The **attention mechanism** allows the decoder to get information from all the hidden states of the encoder, not just the last hidden state.

Encoder-Decoder Model with RNNs - Attention

- ***The idea of attention is to create the single fixed-length vector c by taking a weighted sum of all the encoder hidden states.***
- The weights focus on (‘**attend to**’) a particular part of the source text that is relevant for the token the decoder is currently producing.
- **Attention** thus replaces the static context vector with one that is dynamically derived from the encoder hidden states, different for each token in decoding.



The attention mechanism allows each hidden state of the decoder to see a different, dynamic, context, which is a function of all the encoder hidden states.

Encoder-Decoder Model with RNNs - Attention

- How **relevant** each encoder state is to the decoder state.
  - Similarity of jth encoder hidden state $\mathbf{h}^{\mathbf{e}}_{\mathbf{j}}$ with i-1th decoder hidden state $\mathbf{h}^{\mathbf{d}}_{\mathbf{i-1}}$: **score(**$\mathbf{h}^{\mathbf{d}}_{\mathbf{i-1}}$**,**$\mathbf{h}^{\mathbf{e}}_{\mathbf{j}}$
- Simple score function is **dot-product attention:**

$$\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j) = \mathbf{h}^d_{i-1} \cdot \mathbf{h}^e_j$$

- To use othese scores, we create **a vector of weights $\boldsymbol{\alpha}_{\mathbf{ij}}$**, that tells us the proportional relevance of each encoder hidden state j to the prior hidden decoder state, $\mathbf{h}^{\mathbf{d}}_{\mathbf{i-1}}$.

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j)) = \frac{\exp(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_j))}{\sum_k \exp(\text{score}(\mathbf{h}^d_{i-1}, \mathbf{h}^e_k))}$$

- We can compute a **fixed-length context vector for the current decoder state** by taking a weighted average over all the encoder hidden states.

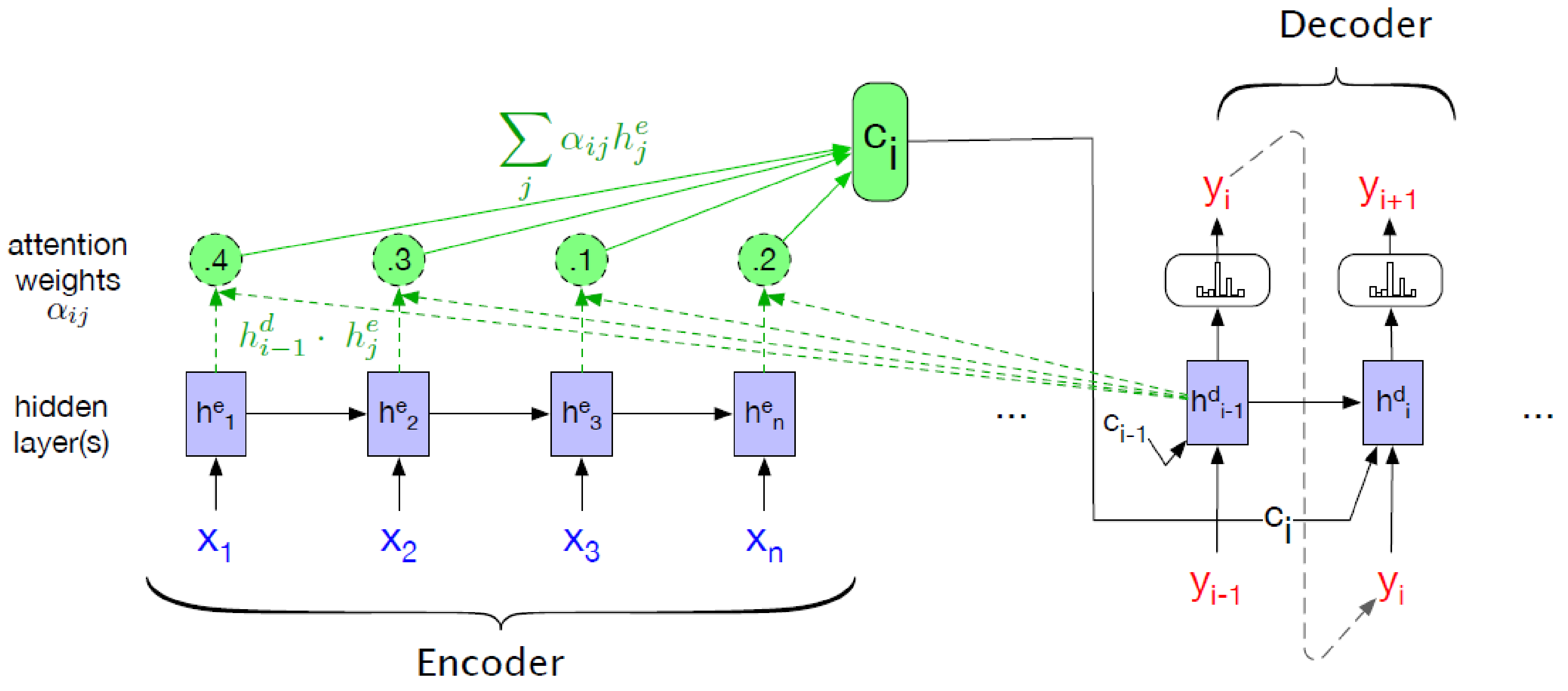
$$\mathbf{c}_i = \sum_j \alpha_{ij}\, \mathbf{h}^e_j$$

Encoder-Decoder Model with RNNs - Attention

The encoder-decoder network with attention, focusing on the computation of c_i:

– The context value c_i is one of the inputs to the computation of $\mathbf{h_i^d}$

– It is computed by taking the weighted sum of all the encoder hidden states, each weighted by their dot product with the prior decoder hidden state $\mathbf{h_{i-1}^d}$.



Recurrent Neural Networks (RNNs) - Summary

Recurrent Neural Networks (RNNs) - Summary

- In simple Recurrent Neural Networks sequences are processed one element at a time, with the output of each neural unit at time t based both on the current input at t and the hidden layer from time t-1.
- RNNs can be trained with a straightforward extension of the backpropagation algorithm, known as **backpropagation through time (BPTT).**
- Simple recurrent networks fail on long inputs because of problems like **vanishing gradients**; modern systems use more complex gated architectures such as **LSTMs** that explicitly decide what to remember and forget in their hidden and context layers.

Recurrent Neural Networks (RNNs) - Summary

Common language-based applications for RNNs include:

- **Probabilistic language modeling**: assigning a probability to a sequence, or to the next element of a sequence given the preceding words.
- **Auto-regressive generation** using a trained language model.
- **Sequence labeling** like part-of-speech tagging, where each element of a sequence is assigned a label.
- **Sequence classification**, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification.
- **Encoder-decoder architectures**, where an input is mapped to an output of different length and alignment.