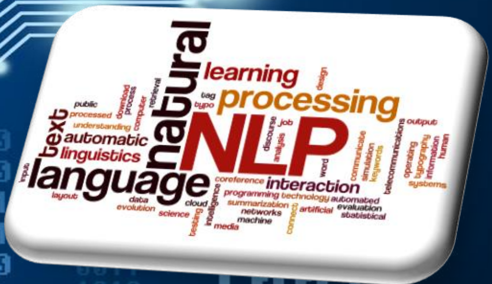
Cairo University
Faculty of Engineering
Computer Engineering Department

Natural Language Processing

Dr. Sandra Wahid



Motivation

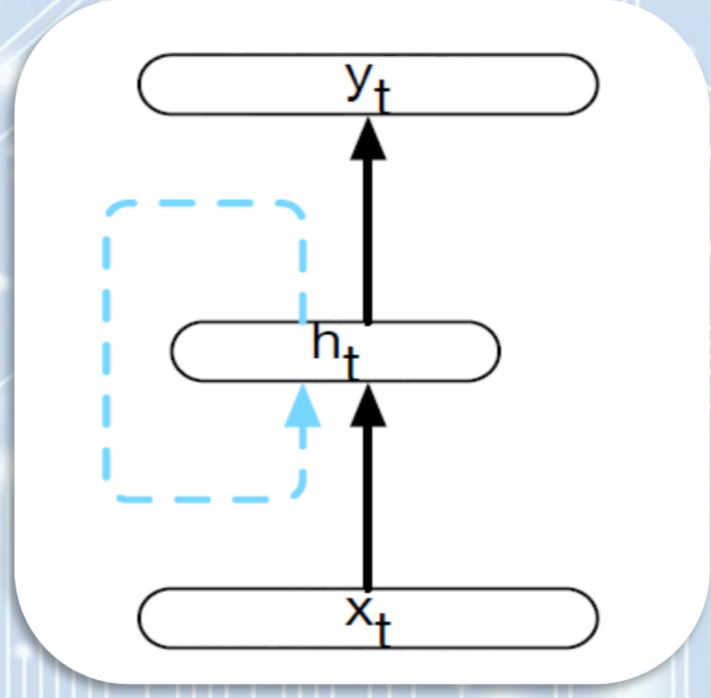
- Language is an inherently **temporal** phenomenon.
  - Normally processed sequentially.

- The primary weakness of our earlier Markov N-gram approaches is 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 an issue since there are many tasks that require access to information that can be arbitrarily **distant** from the point at which processing is happening.

Recurrent Neural Networks(RNNs): have mechanisms to deal directly with the sequential nature of language that allow them to handle variable length inputs without the use of arbitrary fixed sized windows, and to capture and exploit the temporal nature of language.

Recurrent Neural Networks

- A recurrent neural network (RNN) is any network that contains a **cycle** within its network connections.
  - That is, any network where the value of a unit is directly, or indirectly, dependent on its own earlier outputs as an input.



- x_t :an **input vector** representing the current input.
- x_t is multiplied by a **weight matrix** and then passed through a **non-linear activation function** to compute the values for a **layer of hidden units**.
- This hidden layer is then used to calculate a corresponding **output: y_t**

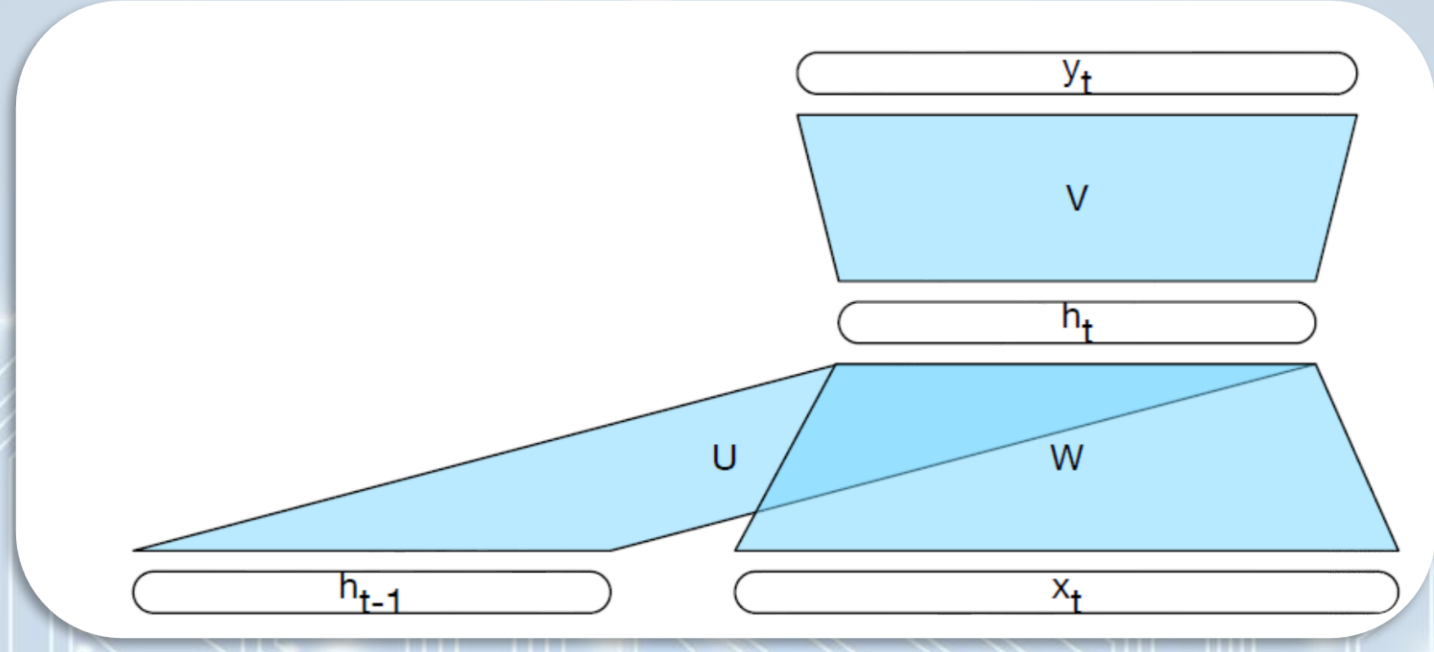
Recurrent Neural Networks

- Sequences are processed by presenting **one item at a time** to the network.

- The key difference from a feedforward network lies in the **recurrent link** shown in the figure with the dashed line →**temporal dimension**
- This link augments the **input** to the computation at the hidden layer with the **value of the hidden layer from the preceding point in time**.

- The hidden layer from the previous time step provides a **form of memory**, or context, that encodes earlier processing.
- Critically, this approach does **not impose a fixed-length limit** on this prior context
  - the context embodied in the previous hidden layer includes information extending back to the **beginning of the sequence**.

Recurrent Neural Networks



- The most significant change lies in the **new set of weights: U** that connect the hidden layer from the previous time step to the current hidden layer.

- These weights determine how the network makes use of past context in calculating the output for the current input.

- As with the other weights in the network, these connections are trained via **backpropagation**.

Inference in RNNs

- 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 multiply **the input x_t with the weight matrix W**, and 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 .

$$h_t = g(\boxed{U h_{t-1}} + \boxed{W x_t})$$

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

$$y_t = f(V h_t)$$

- In the commonly encountered case of soft classification, f is a **softmax** function that provides a probability distribution over the possible output classes.

$$y_t = \mathrm{softmax}(V h_t)$$

Inference in RNNs

It's worthwhile here to be careful about specifying the dimensions of the input, hidden and output layers, as well as the weight matrices to make sure these calculations are correct. Let's refer to the input, hidden and output layer dimensions as d_{in}, d_h, and d_{out} respectively. Given this, our three parameter matrices are: $\boxed{W \in \mathbb{R}^{d_h \times d_{in}}}$, $\boxed{U \in \mathbb{R}^{d_h \times d_h}}$, and $\boxed{V \in \mathbb{R}^{d_{out} \times d_h}}$.

- 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.

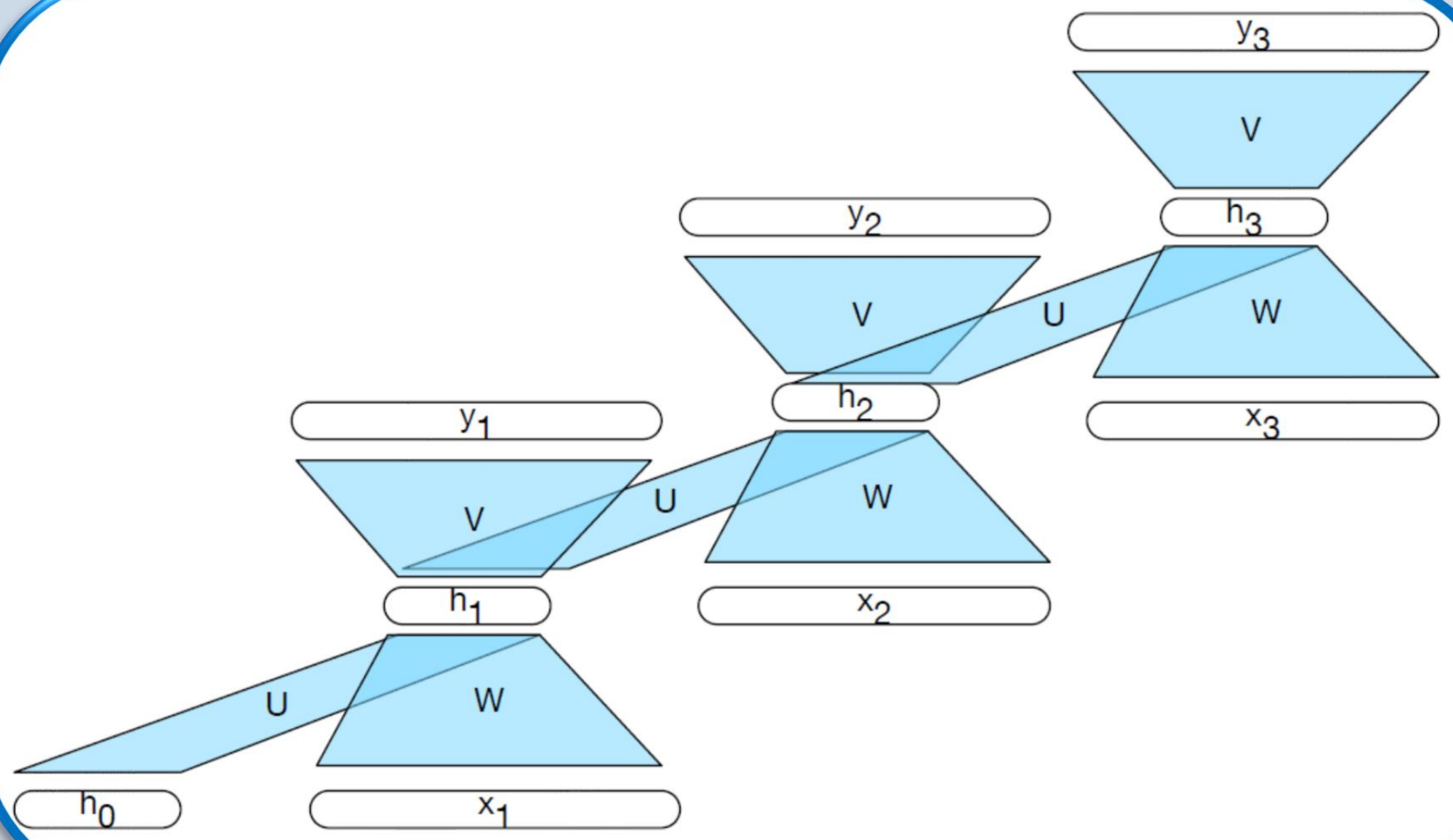
function FORWARDRNN(x, $network$) **returns** output sequence y

$h_0 \leftarrow 0$
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

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

Inference in RNNs

- The sequential nature of simple recurrent networks can also be seen by **unrolling** the network in time:



Training

- A neural network is an instance of **supervised** machine learning.

→We know the **correct output *y*** for each **observation *x***.

→The system generates **output** $\hat{y}$.

- The goal of the training procedure is to learn parameters such as *W* that makes $\hat{y}$ as close as possible to the true *y*.

→First, we'll need a **loss function** that models the distance between the system output and the gold output.

→Second, to find the parameters that minimize this loss function, we'll use the **gradient descent optimization** algorithm.

Forward propagation: Computing the error.
Backward propagation: Computing the partial derivatives and updating the parameters.

Training

- In RNN, there are 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.
  3. **V**, the weights from the hidden layer to the output layer.

- RNNs are trained using **Backpropagation Through Time (BPTT):**
  - BPTT begins by unrolling a recurrent neural network in time.
  - Propagates the error backward over the entire input sequence, one timestep at a time.
  - Equations to calculate the error gradients:
    - **V**: depends only on the **current timestamp**.
    - **W and U**: depend on **current timestamp and previous** ones.
    - Using Chain rule at each timestep:

$$\frac{\partial Et}{\partial V} = \frac{\partial Et}{\partial y_t}\frac{\partial y_t}{\partial V} \qquad \frac{\partial Et}{\partial W} = \sum_{k=1}^{t} \frac{\partial Et}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W} \qquad \frac{\partial Et}{\partial U} = \sum_{k=1}^{t} \frac{\partial Et}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial U}$$

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

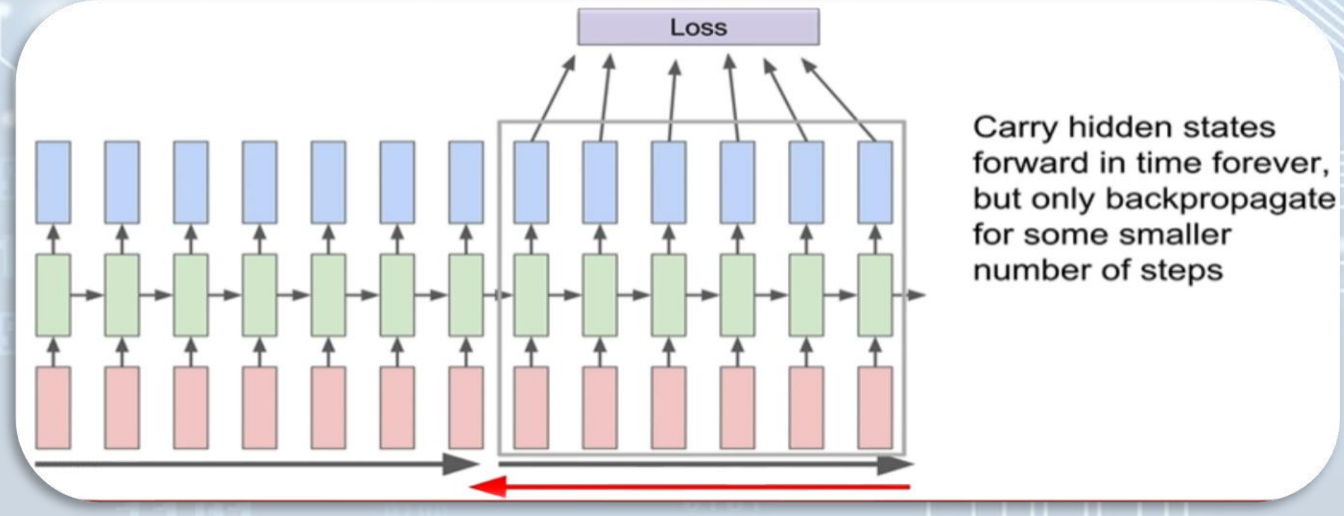
Ex: at timestep3: $\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial W} + \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W} + \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W}$

    - Total Error: is the sum of errors for all timesteps $\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial Et}{\partial W}$

Training

- For applications that involve **long input sequences**, such as speech recognition, character-level processing, or streaming of continuous inputs, unrolling an entire input sequence may not be feasible and BPTT becomes very expensive and problematic (*check BPTT disadvantages with long sequences*).

- One solution is to use **Truncated Backpropagation Through Time.**
  - Choose a number of timesteps to use as input →split up the long input sequences into subsequences.
  - A modification of BPTT to limit the number of timesteps used on the backward pass and in effect estimate the gradient used to update the weights rather than calculate it fully.



RNNs as Language Models

- 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 **don't have the limited context** problem that n-gram models have, since the hidden state can in principle represent information about all of the preceding words **all the way back to the beginning of the sequence**.

- The input sequence: $\mathbf{X} = [\mathbf{x}_1; ...; \mathbf{x}_t; ...; \mathbf{x}_N]$ consists of a series of word embeddings each represented as a **one-hot vector** of size **|V|x1.**
- The output prediction: **y** is a vector representing a probability distribution over the vocabulary.
- **E** the word embedding matrix: used to retrieve the embedding for the current word.

RNNs as Language Models

- At time t:

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

- The vector resulting from **Vh** can be thought of as a **set of scores** over the vocabulary given the evidence provided in h.

- Passing these scores through the softmax normalizes the scores into a **probability distribution**.

- The probability that a particular word **i** in the vocabulary is the next word is represented by $\mathbf{y_t[i]}$→the **ith** component of $\mathbf{y_t}$:

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

RNNs as Language Models

- The probability of an entire sequence is just the product of the probabilities of each item in the sequence.

- $\mathbf{y_i[w_i]}$ means the probability of the true word $\mathbf{w_i}$ at time step **i**

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

- Training an RNN as a language model:
  - we use a **corpus of text** as training material
  - the model **predict the next word** at each time step t.
  - we **minimize the error** in predicting the true next word in the training sequence, using **cross-entropy** as the loss function.

Cross Entropy Loss

- Measures the difference between a predicted probability distribution and the correct distribution.
- For example, in the case of Binary Classification, cross-entropy is given by:

$$l = -(\, y\, log(p) \; + \; (1-y)\, log(1-p)\,)$$

  - *p* is the predicted probability
  - *y* is the indicator (0 or 1 in the case of binary classification)
  - For example, if the correct indicator is *y=1*

$$l = -(\, 1 \times log(p) + (1-1)\; log(1-p)\;)$$
$$l = -(\, 1 \times log(p)\;)$$

  - Therefore, our loss function will reward the model for giving a correct prediction (high value of p) with a low loss. However, if the probability is lower, the value of the error will be high and therefore it penalizes the model for a wrong outcome. → **best when loss=0** (the probability of the correct class=1)
- Extension for a Multi-Classification (N classes):

$$-\sum_{c=1}^{N} y_c log(p_c)$$

RNNs as Language Models

- The cross-entropy loss for language modeling is determined by the probability the model assigns to the **correct next word**.
- So at time **t** the **CE** loss is the negative log probability the model assigns to the next word in the training sequence.

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

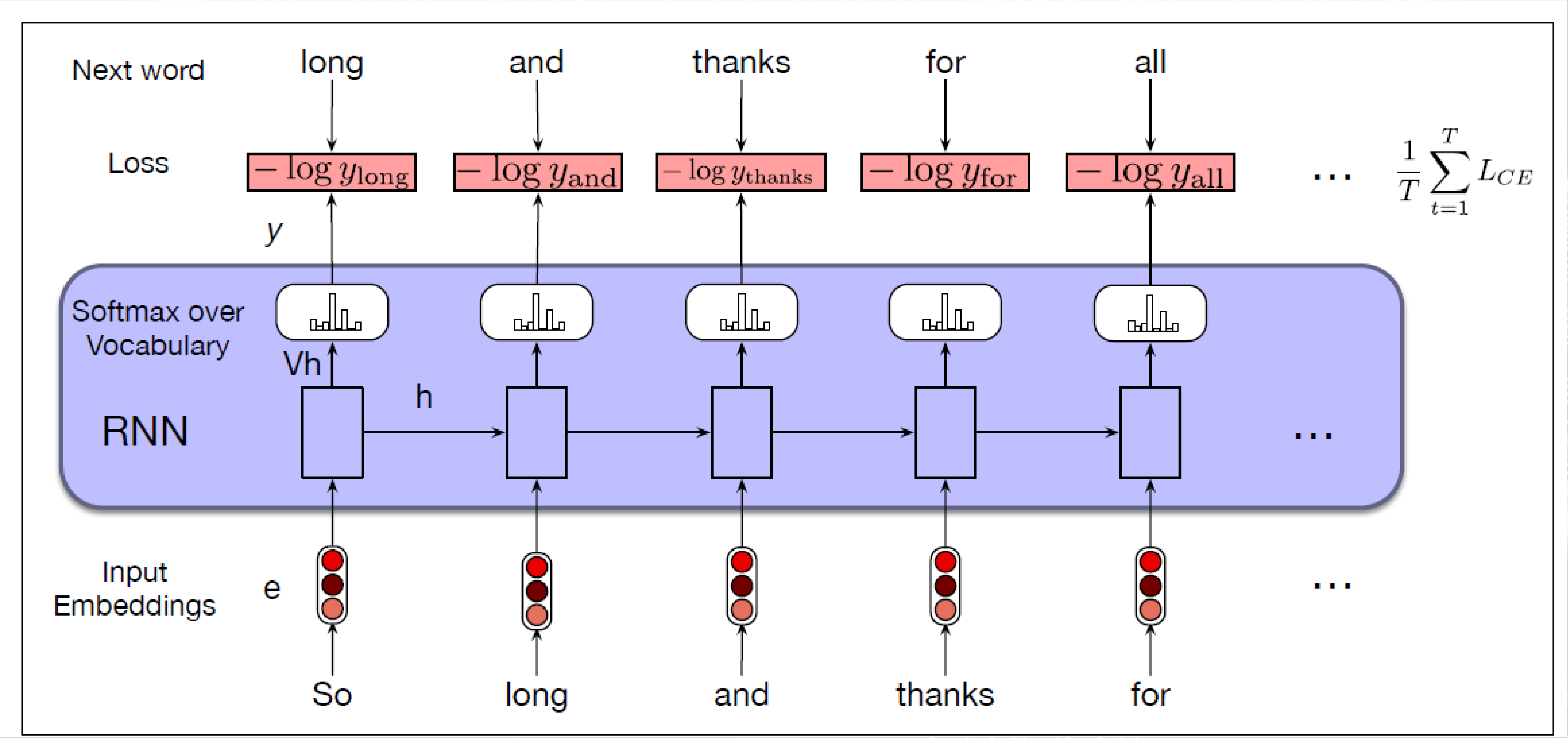
- **Teacher Forcing**

  At each word position *t* of the input, the model takes as input the **correct sequence of tokens $\mathbf{w}_{1:t}$** and uses them to compute a probability distribution over possible next words and computes the model's loss for the next token w_{t+1}. Then we move to the next word, we ignore what the model predicted in the previous time step and instead use the correct sequence of tokens $w_{1:t+1}$ to estimate the probability of token w_{t+2}.

The idea is that we always give the model the correct history sequence to predict the next word (rather than feeding the model its best case from the previous time step)

RNNs as Language Models

- The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent. The following figure illustrates the training process.



RNNs as Language Models

- **Weight tying:** is a method that dispenses redundancy and simply uses a single set of embeddings at the input and softmax layers.
  - We dispense with V and use E in both the start and end of the computation.

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

Dimension of the hidden layer *dh* = Dimension of embeddings

- This provides **improved model perplexity**, and significantly **reduces the number of parameters** required for the model.


Thank You