Natural Language Processing

Lecture : Neural Language Models

Master Degree in Computer Engineering
University of Padua
Lecturer : Giorgio Satta

Neural language models

Omid Armin on Unsplash

Neural language models

N-gram language models have been largely supplanted by **neural language models** (NLM).

Main advantages of NLM

- can incorporate arbitrarily distant contextual information, while remaining computationally and statistically tractable
- can generalize better over contexts of similar words, and are more accurate at word-prediction

On the other hand, as compared with *N*-gram language models, NLM are much more complex, are slower, need more time to train, and are less interpretable.

For many (especially smaller) tasks *N*-gram language models is still the right tool.

Neural language models

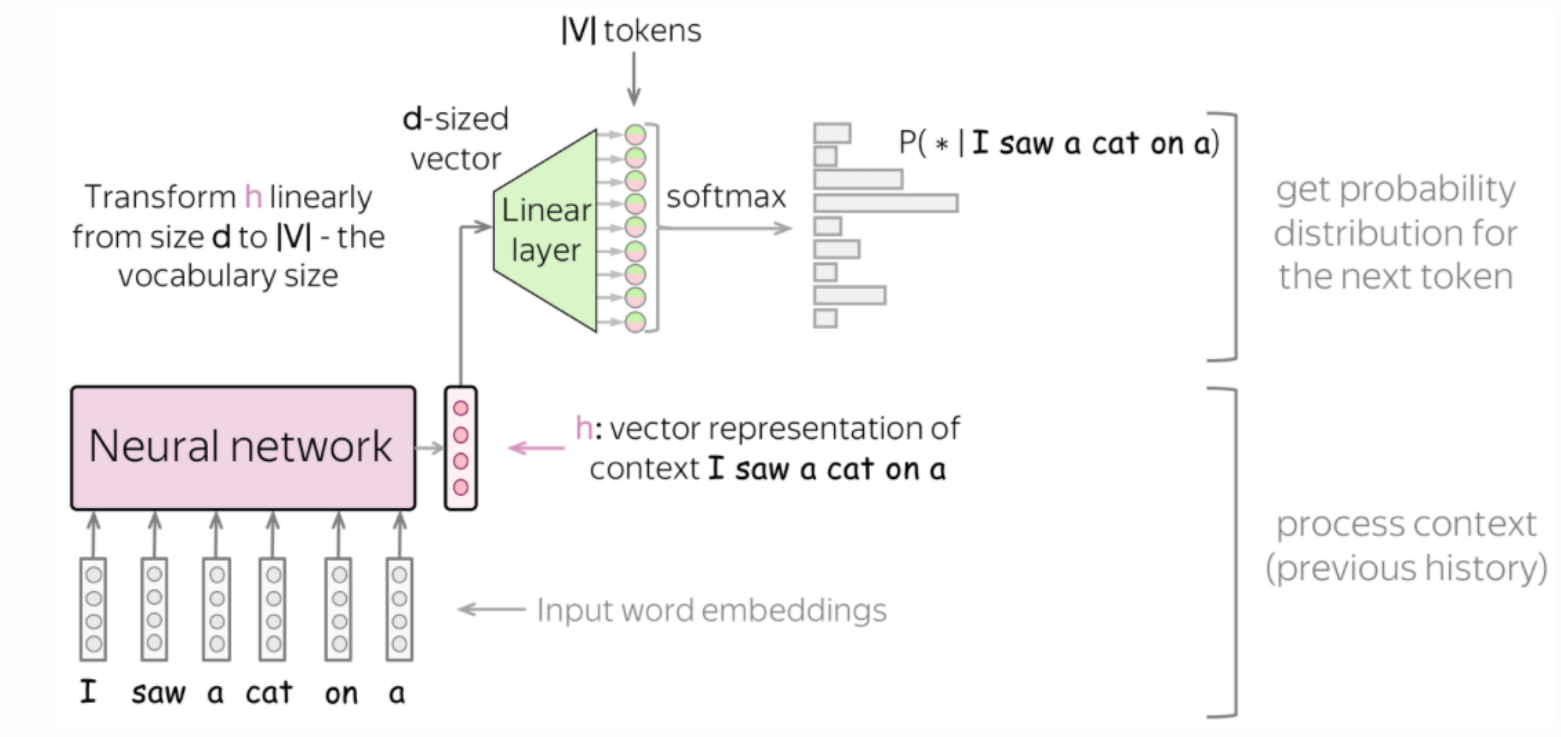
Idea :

- get a vector representation for the previous context
- generate a probability distribution for the next token

First bullet depends on NN architecture; second bullet is model-agnostic.

Most natural choice for NN architecture is recurrent neural network (RNN) but feedforward neural network (FNN) and convolutional neural network (CNN) have also been exploited.

General architecture for NLM



Elena Voita
https://lena-voita.github.io/nlp_course.html

Feedforward NLM: inference

See Jurafsky & Martin §6.3 and §6.6 for definition and training of feedforward neural networks.

Like the N-gram language model, the feedforward NLM uses the following **approximation**

$$P(w_t \mid w_{1:t-1}) \approx P(w_t \mid w_{t-N+1:t-1})$$

and a moving window that can see $N-1$ words into the past.

For $w \in V$, let $ind(w) \in [1..|V|]$ be the **index** associated with w.

We represent each input word w_t as a **one-hot vector** $\mathbf{x}_t$ of size $|V|$, defined as follows

- element $\mathbf{x}_t[ind(w_t)]$ is set to one
- all the other elements of $\mathbf{x}_t$ are set to zero

Feedforward NLM: inference

At the first layer

- we convert one-hot vectors for the words in the $N-1$ window into word embeddings of size d
- we concatenate the $N-1$ embeddings

The first hidden layer equation is (assuming $N = 4$)

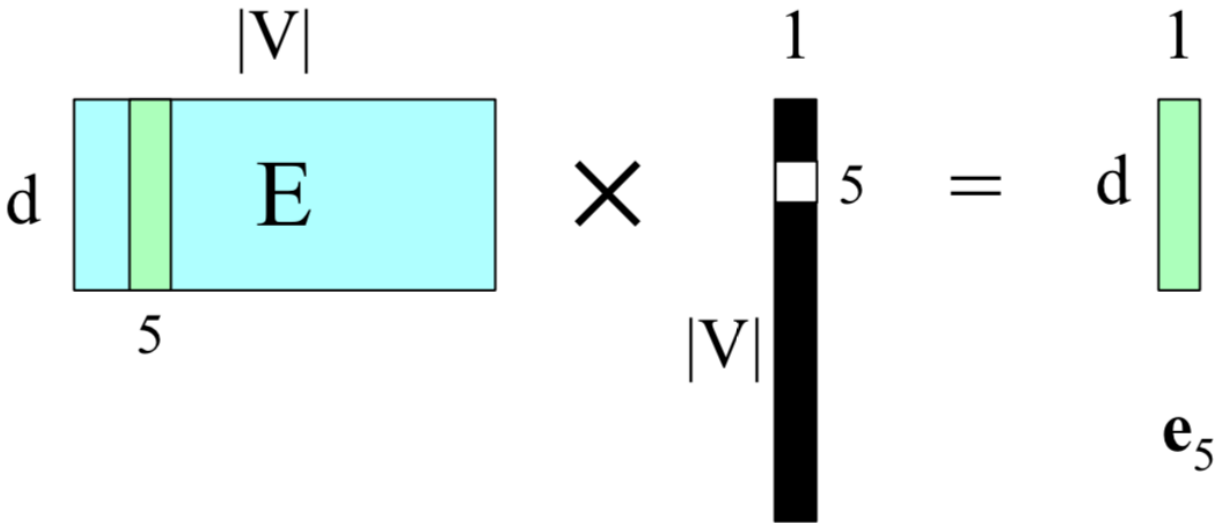
$$\mathbf{e}_t = [\mathbf{E}\mathbf{x}_{t-3}; \mathbf{E}\mathbf{x}_{t-2}; \mathbf{E}\mathbf{x}_{t-1}]$$

where

- $\mathbf{E} : d \times |V|$ is a learnable matrix with the word embeddings
- $\mathbf{x}_{t-i} : |V| \times 1$ are 1-hot representation of word w_{t-i}
- $\mathbf{e}_t : 3d \times 1$ is the concatenation of the embeddings of the $N-1$ previous words

Feedforward NLM: inference

Conversion from 1-hot word representation into word embedding.



Feedforward NLM: inference

The model equations for the remaining layers

$$\begin{aligned} \mathbf{h}_t &= g(\mathbf{W}\mathbf{e}_t + \mathbf{b}) \\ \mathbf{z}_t &= \mathbf{U}\mathbf{h}_t \\ \hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{z}_t) \end{aligned}$$

where

- $\mathbf{W} : d_h \times 3d$ is a learnable matrix, d_h the size of the second hidden vector representation
- $\mathbf{b} : d_h \times 1$ is a learnable vector
- $\mathbf{h}_t : d_h \times 1$ is obtained through some activation function g
- $\mathbf{U} : |V| \times d_h$ is a learnable matrix
- $\mathbf{z}_t$, $\hat{\mathbf{y}}_t : |V| \times 1$ scores and distribution (see next slide)

Minor changes to indices wrt the textbook.

Feedforward NLM: inference

The vector $\mathbf{z}_t$ can be thought of as a set of scores over V, also called **logits**: raw (non-normalized) predictions that a classification model generates.

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

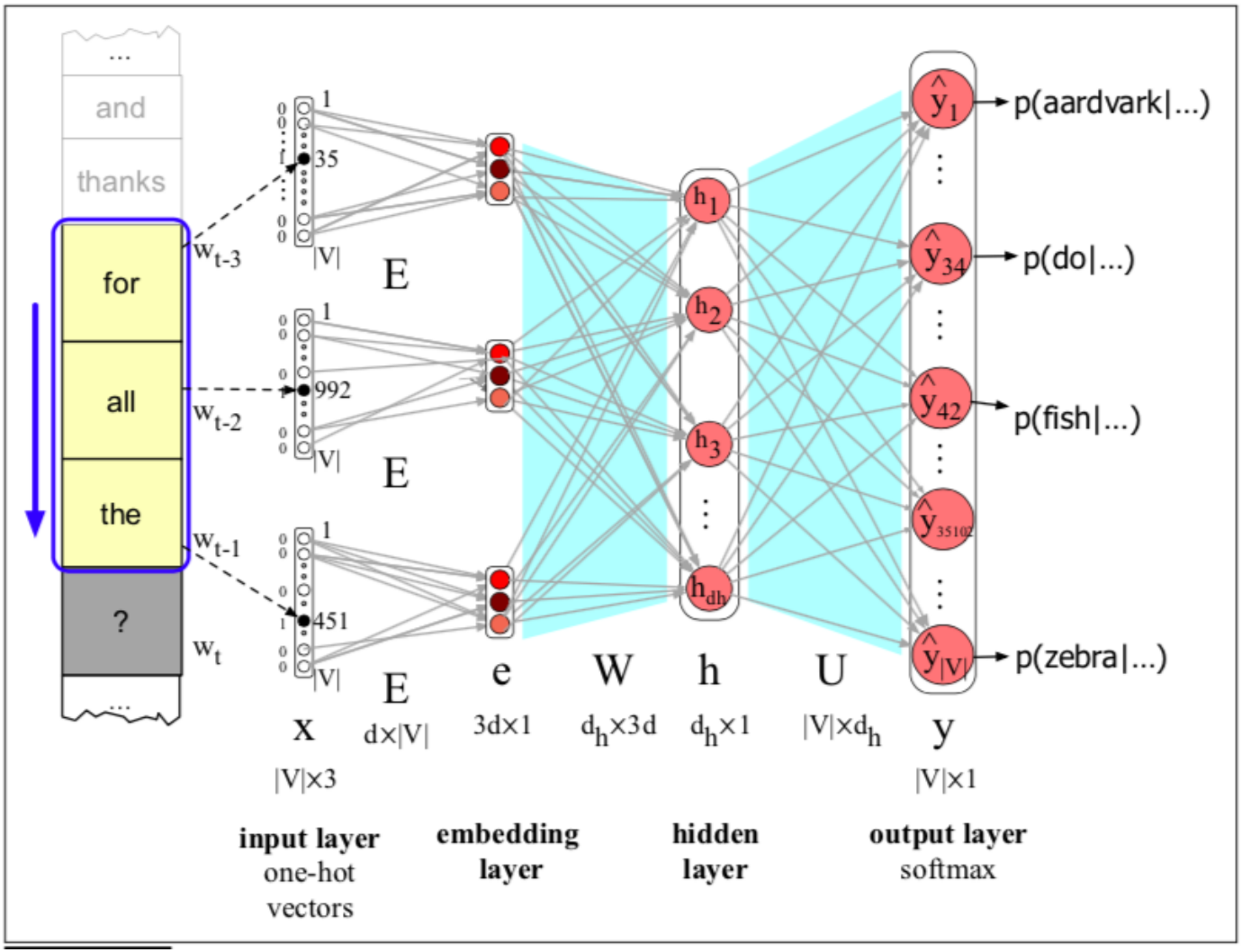
For each $w \in V$, the element of $\hat{\mathbf{y}}_t$ with index $ind(w)$ is the probability that the word at position t is w:

$$\forall w \in V, \quad \hat{\mathbf{y}}_t[ind(w)] \quad = \quad P(w_t = w \mid w_{t-3:t-1})$$

Recall we are assuming $N = 4$.

Feedforward NLM: inference

Feedforward NLM architecture



Feedforward NLM: training

The **parameters** of the model are $\mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$.

The number of parameters is $\mathcal{O}(|V|)$, since d and d_h are constants.

Let w_t be the word at position t in the training data. Then the **true distribution** $\mathbf{y}_t$ for the word at position t is a 1-hot vector of size $|V|$ with

- $\mathbf{y}_t[ind(w_t)] = 1$
- $\mathbf{y}_t[k] = 0$ everywhere else

We use the cross-entropy loss for training the model:

$$\begin{aligned} L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) &= -\sum_{k=1}^{|V|} \mathbf{y}_t[k] \log \hat{\mathbf{y}}_t[k] \\ &= -\log \hat{\mathbf{y}}_t[ind(w_t)] \end{aligned}$$

This is the expected information of $\hat{\mathbf{y}}_t$ computed w.r.t. $\mathbf{y}_t$.

Feedforward NLM: training

Recall the equation for the estimated distribution (general N)

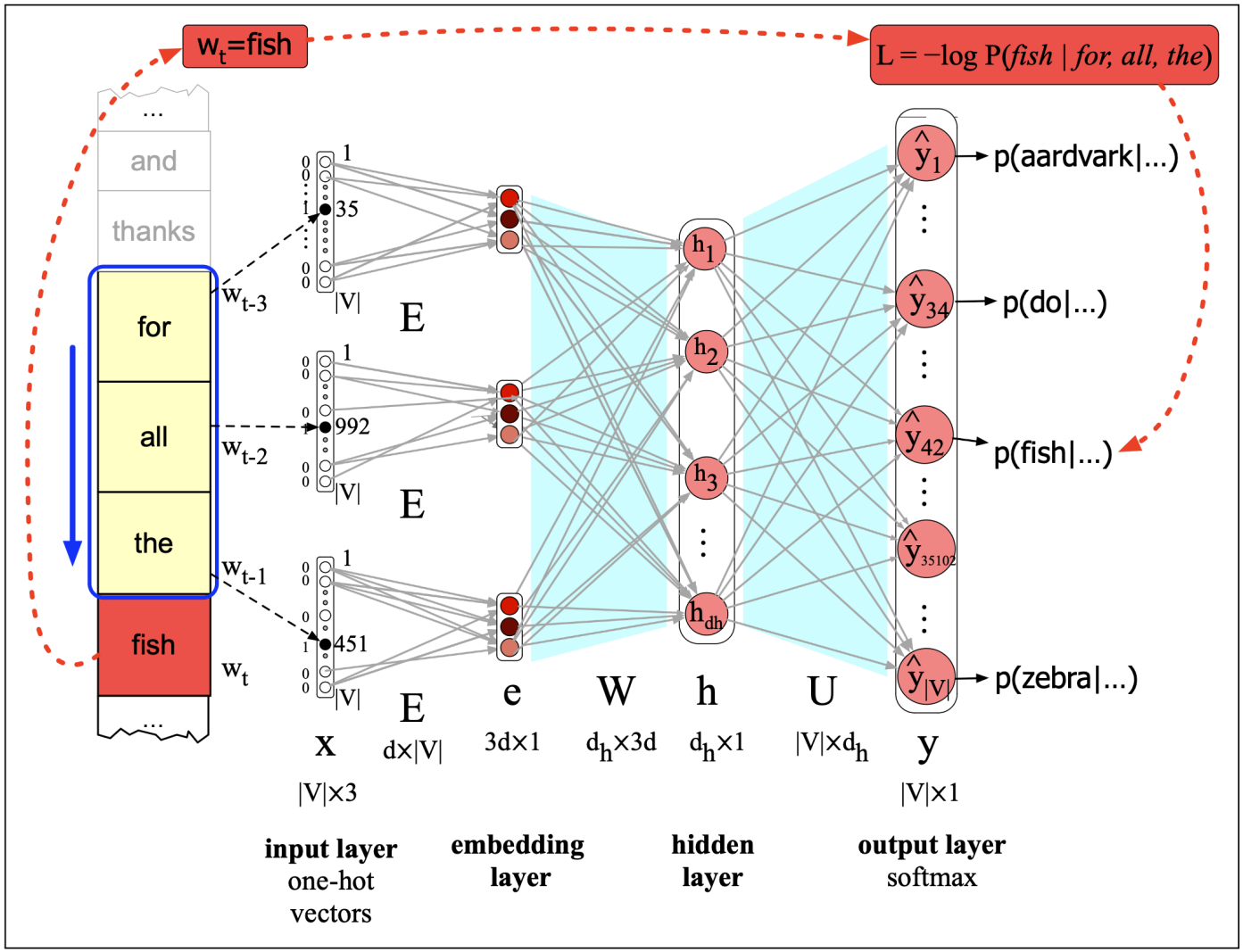
$$\hat{\mathbf{y}}_t[ind(w_t)] \quad = \quad P(w_t \mid w_{t-N+1:t-1})$$

Replacing in the cross-entropy loss equation, we obtain

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) \quad = \quad -\log P(w_t \mid w_{t-N+1:t-1})$$

Observe that the cross-entropy loss equals the **negative log likelihood** of the training data (see later slides).

Feedforward NLM training



Recurrent NLM: inference

See Jurafsky & Martin §13.1 for definition and training of recurrent neural networks.

RNN language models process the input one word at a time, predicting the next word from the current **hidden state**.

The current hidden state is computed from

- the current word
- the previous hidden state

RNNs can model probability distribution $P(w_t \mid w_{1:t-1})$ without the $N-1$ window approximation of feedforward NLM.

The hidden state of the RNN model can (in principle) represent information about all of the preceding words.

Recurrent NLM: inference

The model equations are (for some $\mathbf{h}_0$)

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

where (d_h is the size of the hidden vectors)

- $\mathbf{x}_t : |V| \times 1$ is a 1-hot representation of word w_t
- $\mathbf{E} : d_h \times |V|$ is a learnable matrix with the word embeddings
- $\mathbf{U}, \mathbf{W} : d_h \times d_h$ are learnable matrices
- $\mathbf{h}_t : d_h \times 1$ is the hidden vector at step t
- $\mathbf{V} : |V| \times d_h$ is a learnable matrix
- $\hat{\mathbf{y}}_t : |V| \times 1$ is a probability distribution

Minor changes to notation wrt the textbook.

Recurrent NLM: inference

The vector resulting from $\mathbf{V}\mathbf{h}_t$ records the **logits** (unormalized scores) over the vocabulary V, given the evidence provided by $\mathbf{h}_t$.

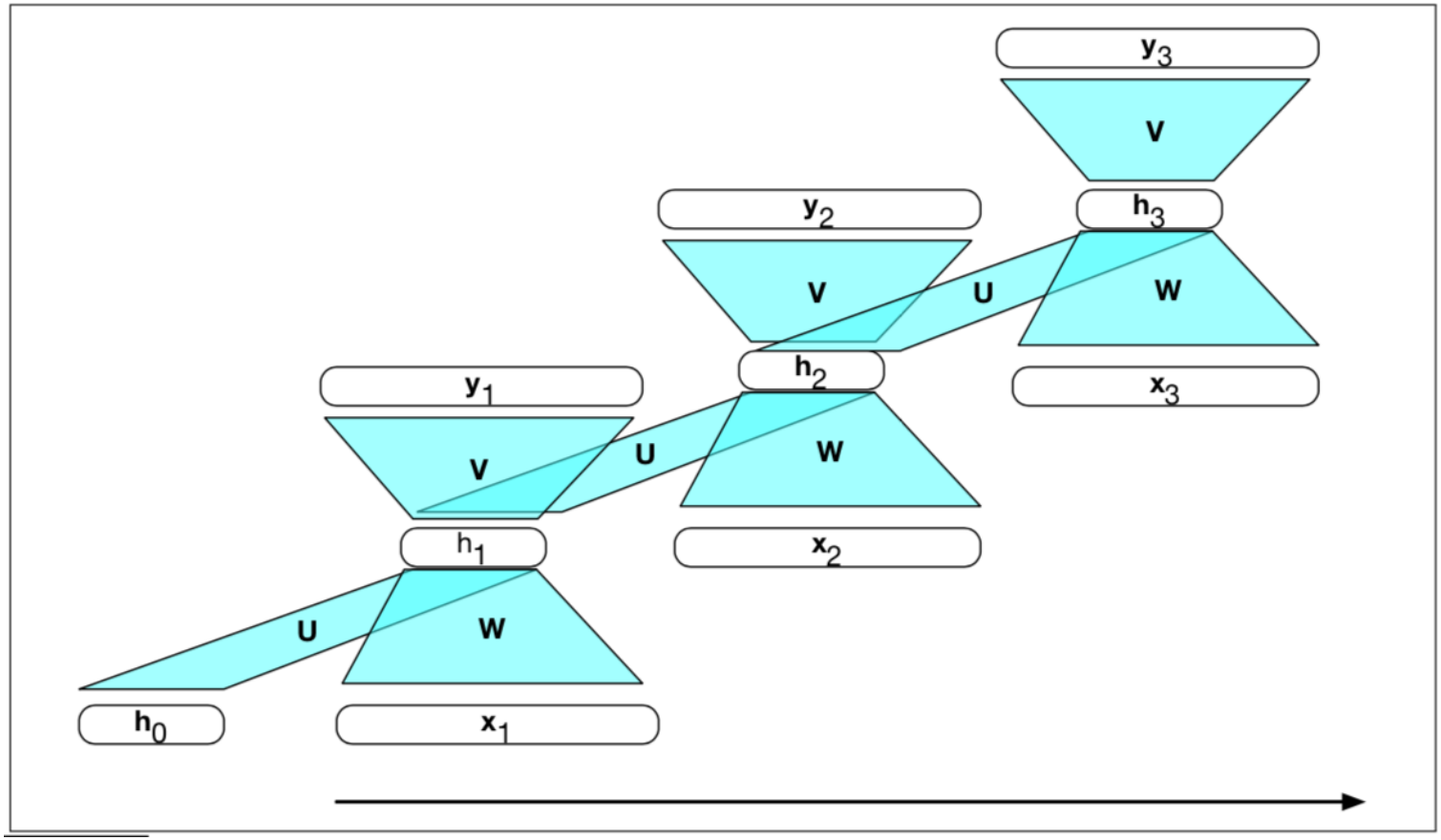
The softmax normalizes the logits, resulting in the **estimated distribution** $\hat{\mathbf{y}}_t$ for the next word.

More precisely, for each word $w \in V$, the element of $\hat{\mathbf{y}}_t$ with index $ind(w)$ estimates the probability that the **next word** is w:

$$\hat{\mathbf{y}}_t[ind(w)] \quad = \quad P(w_{t+1} = w \mid w_{1:t})$$

Recurrent NLM: inference

The recurrent NLM unrolled in time



Recurrent NLM: training

The number of parameters of the model is $\mathcal{O}(|V|)$, since d_h is a constant.

Let $\mathbf{y}_t$ be the **true distribution** on the next word, computed at step t. This is a 1-hot vector over V, obtained from the training set.

We train the model to **minimize** the error in predicting the true next word w_{t+1} in the training sequence, using cross-entropy as the loss function:

$$\begin{aligned} L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) &= -\sum_{w \in V} \mathbf{y}_t[ind(w)] \log \hat{\mathbf{y}}_t[ind(w)] \\ &= -\log \hat{\mathbf{y}}_t[ind(w_{t+1})] \end{aligned}$$

Again, the cross-entropy loss equals the negative log likelihood of training set.

Recurrent NLM: training

At each step t during training

- prediction is based on the correct sequence of tokens $w_{1:t}$
- we ignore what the model predicted at previous steps

The idea that we always give the model the correct history sequence to predict the next word is called **teacher forcing**.

Teacher forcing has some **disadvantages**: the model is never exposed to prediction mistakes; therefore at inference time the model is not able to recover from errors.

Character-level NLM

Character-level NLM

- improves modeling of uncommon and unknown words
- reduces training parameters due to the small softmax

Performance usually worse than the word-level NLMs, since longer history is needed to predict the next word correctly.

A variety of solutions that combine character- and word- level information have been proposed, called **character-aware LM**.

Practical issues

Practical issues

Both the feedforward NLM and the recurrent NLM learn word embeddings $\mathbf{E}$ simultaneously with training the network.

Alternatively, one can resort to **freezing**:

- use pretrained word embeddings, for instance word2vec
- hold $\mathbf{E}$ constant while training, and modify the remaining parameters in θ

Practical issues

In the recurrent NLM

- the columns of $\mathbf{E}$ provide the learned word embeddings
- the rows of $\mathbf{V}$ provide a second set of learned word embeddings, that capture relevant aspects of word meaning and function

Weight tying, also known as parameter sharing, means that we impose $\mathbf{V} = \mathbf{E}^\top$.

Weight tying can significantly reduce model size, and has an effect similar to **regularization**, preventing overfitting of the NLM.

Practical issues

RNNs suffer from the **vanishing gradient** problem: past events have weights that decrease exponentially with the distance from actual word w_t.

Gated recurrent units (GRU) and long-short term memory (LSTM) neural networks are **much better** in capturing long distance relations.

Practical issues

The last step in NLMs, involving softmax over the entire vocabulary, dominates the computation both at training and at test time.

Vocabulary usually contains 10K to 100K words.

An effective alternative is **hierarchical softmax**, based on word clustering.

Adaptive softmax is a simple variant of hierarchical softmax, based on Zipf's law, especially tailored for GPUs.

This provides giant speedup with only minimal costs in accuracy.

The text generated by sampling our NLM should be

- **coherent**: text has to make sense
- **diverse**: the model has to be able to produce very different samples

There is a trade-off between the two.

A very popular method for balancing the coherence / diversity trade-off is to change the softmax **temperature** τ

$$\frac{\exp(\frac{\mathbf{V}_i \mathbf{h}_t}{\tau})}{\sum_j \exp(\frac{\mathbf{V}_j \mathbf{h}_t}{\tau})}$$

where $\mathbf{V}_i$ denotes the i-th row of $\mathbf{V}$.

Low τ produces peaky distribution (high coherence); large τ produces flat distribution (high diversity).

Contrastive evaluation is used to test specific linguistic constructions in NLM.

Example : Subj/Obj agreement, compare $P(\text{is} \mid w_{1:t-1})$ to $P(\text{are} \mid w_{1:t-1})$

- The **roses** ___
- The **roses** in the vase ___
- The **roses** in the vase by the door ___