Neurális hálók és neurális nyelvmodellek

A számítógépes nyelvészet alapjai – 2022/23 tavasz
9-10. óra

Simon Eszter

2023. május 15.

Tartalom

Bevezetés

Gépi tanulás

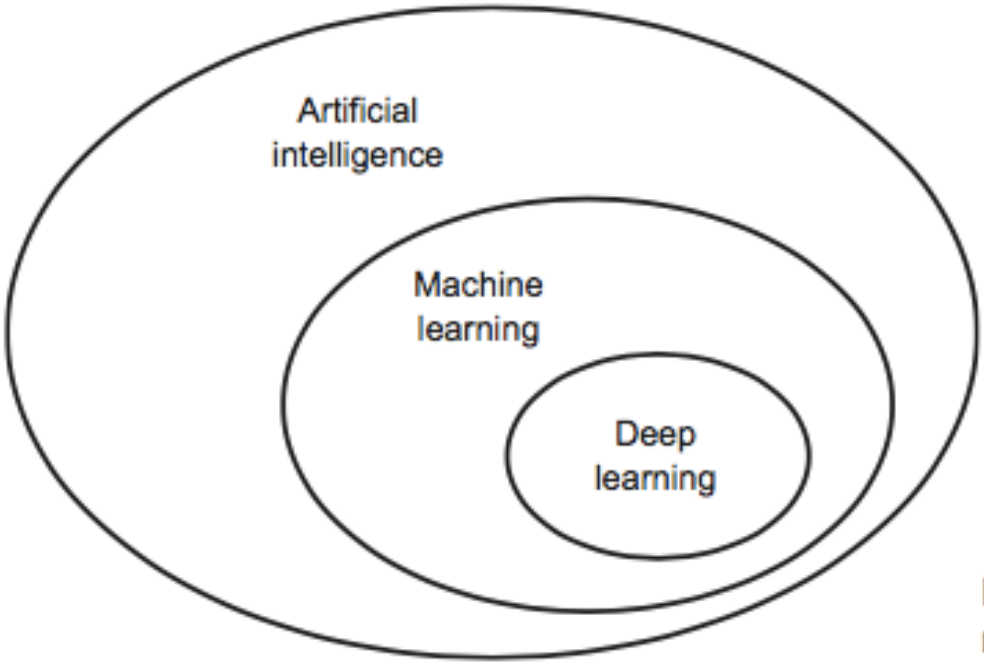


Figure 1.1 Artificial intelligence, machine learning, and deep learning

The ‘deep’ in deep learning

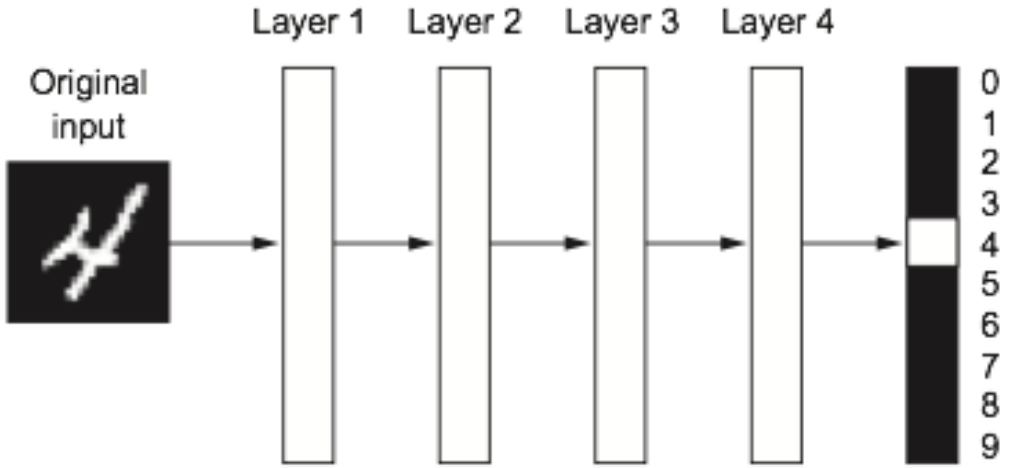


Figure 1.5 A deep neural network for digit classification

Understanding how deep learning works 1.

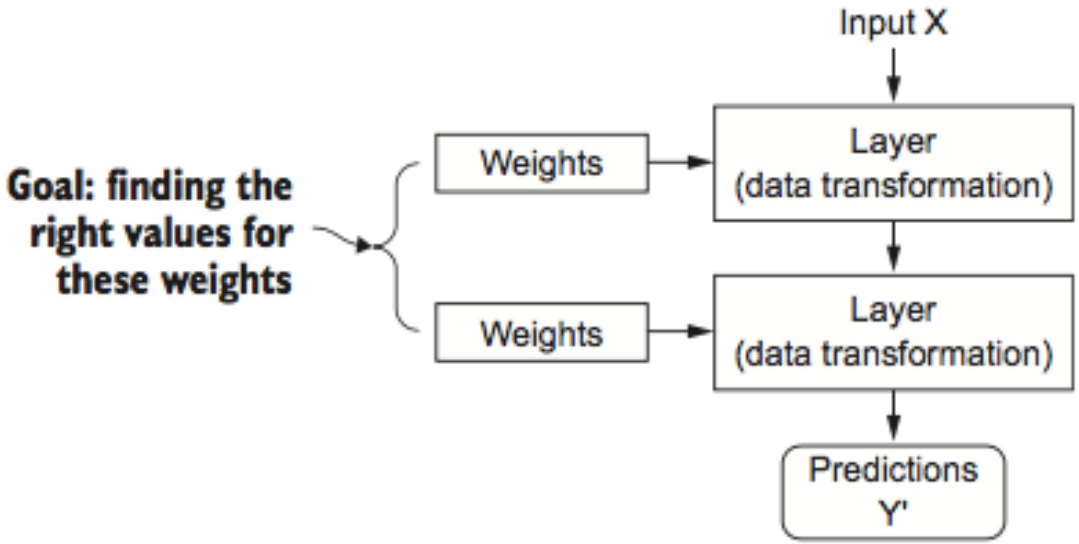


Figure 1.7 A neural network is parameterized by its weights.

Understanding how deep learning works 2.

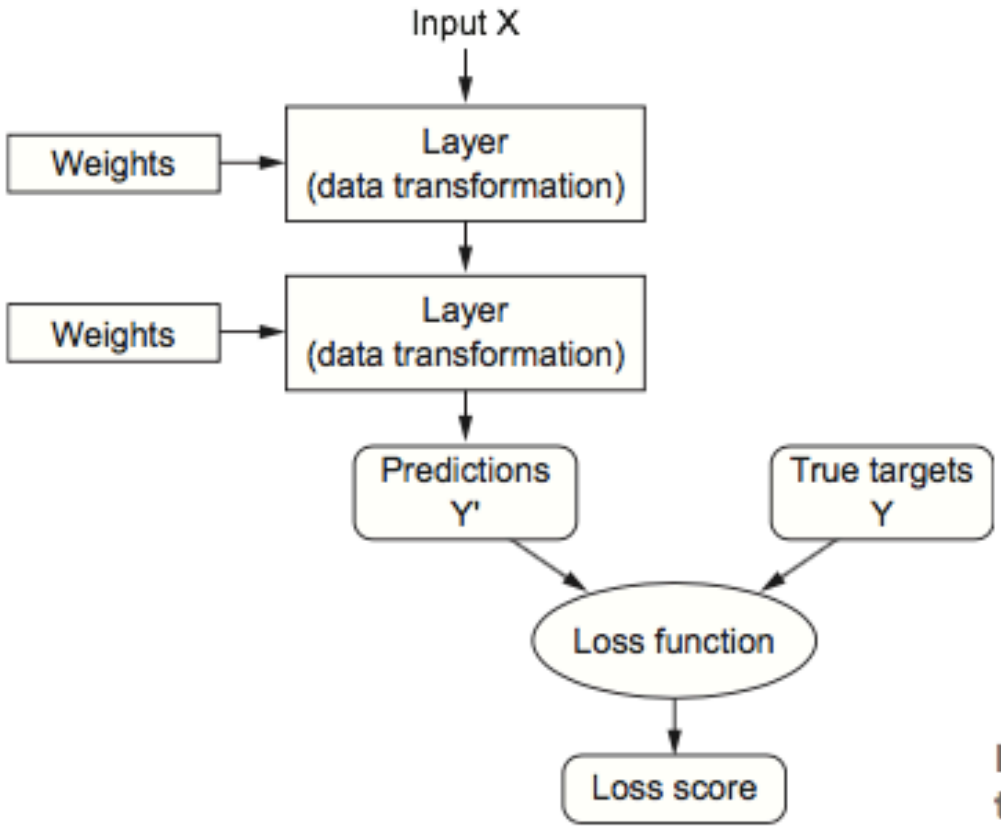


Figure 1.8 A loss function measures the quality of the network's output.

Understanding how deep learning works 3.

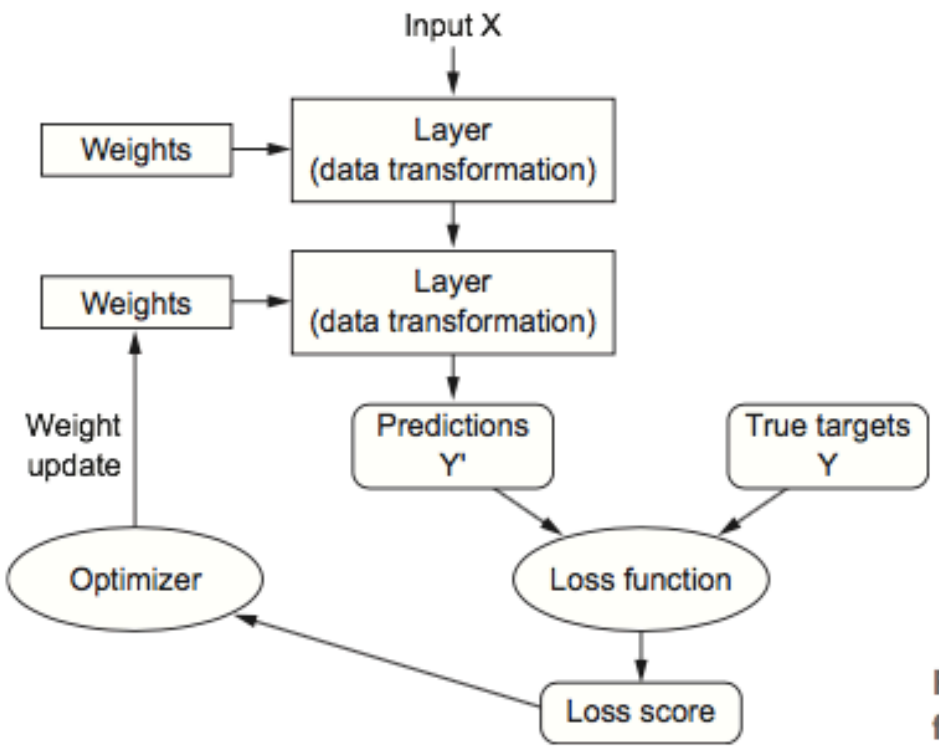


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

the fundamental trick is to use the loss score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score

Training loop

- kezdetben a súlyok random értékek $\rightarrow$ az output távol van az ideálistól, a loss score nagyon magas
- a súlyok minden egyes tanulási kör során egy kicsit módosulnak $\rightarrow$ a loss score kisebb lesz
- ha ezt a tanulási kört elégszer iteráljuk, akkor elérjük a loss score minimumát
- a minimális loss score-ral rendelkező rendszer kimenete lesz a legközelebb a gold standardhez

Történeti áttekintés

The promise of AI

mottó:

“Don’t believe in the short-term hype, but do believe in the long-term vision.”

a deep learning sok mindenre jó, de nem mindenre a legjobb eszköz:

- kevés az adat
- más algoritmus jobban használható az adott feladatra

AI winters

AI winter: high expectations for the short term → technology fails to deliver → research investment dries up, slowing progress for a long time

1. 1960s: symbolic AI
   Marvin Minsky 1967: “Within a generation … the problem of creating artificial intelligence will substantially be solved.”
   1969-70: first AI winter
2. 1980s: expert systems
   a few initial success stories → expensive to maintain, difficult to scale, and limited in scope
   early 1990s: second AI winter

Elméleti alapok

- 1940s: McCulloch–Pitts neuron: a simplified model of the human neuron as a kind of computing element
- 1950/60s: perceptron (Rosenblatt, 1958), bias (Widrow and Hoff, 1960), XOR (Minsky and Papert, 1969)
- 1980s: backpropagation (Rumelhart et al., 1986), handwriting recognition with backpropagation and convolutional neural networks (LeCun et al., 1989)
- 1990s: recurrent networks (Elman, 1990), Long Short-Term Memory (1997)
- 2010s: Geoffrey Hinton et al., Yoshua Bengio et al.

Why now?

Hardware

- Graphical Processing Unit (GPU): developed for gaming
- 2007: NVIDIA launched CUDA, a programming interface for its line of GPUs
- a small number of GPUs can replace massive clusters of CPUs
- parallelizable matrix multiplications
- 2016: Tensor Processing Unit (TPU) by Google

Data

“if deep learning is the steam engine of this revolution, then data is its coal”

Why now? – cont.

Algorithms

The feedback signal used to train neural networks would fade away as the number of layers increased.

- better activation functions
- better weight-initialization schemes
- better optimization schemes

Only when these improvements began to allow for training models with 10 or more layers did deep learning start to shine.

A new wave of investment

total investment in AI: 2011: $19 million $\rightarrow$ 2014: $394 million

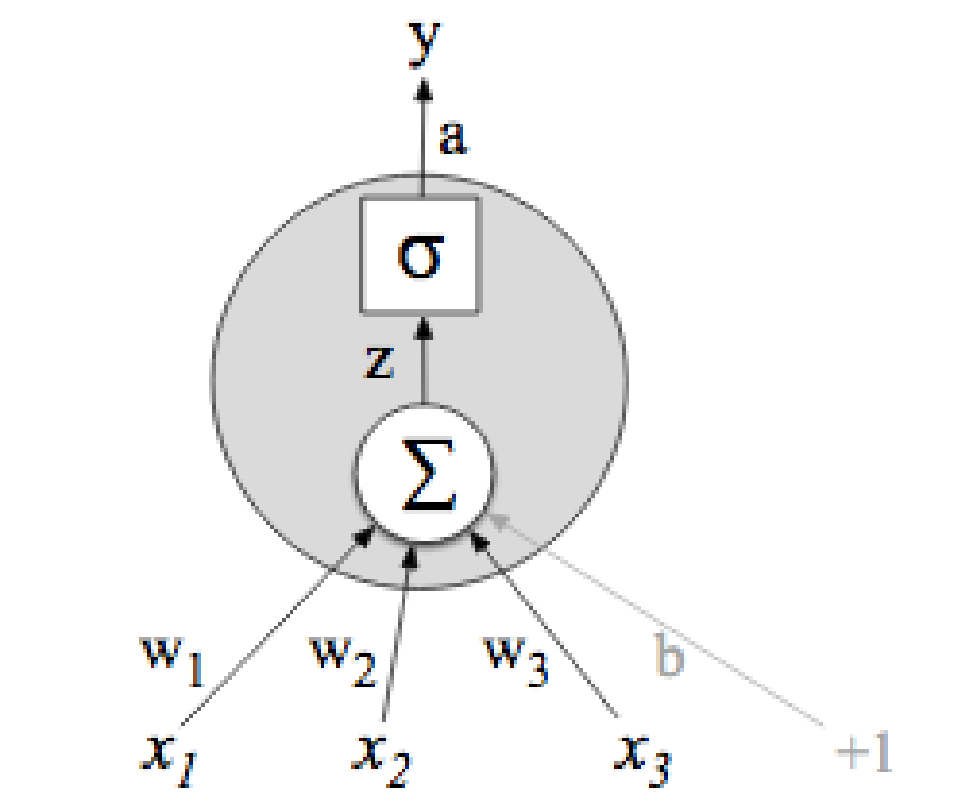
Why now? – cont.

The democratization of deep learning

early days: doing deep learning required significant programming expertise $\rightarrow$ now: basic Python scripting skills are sufficient (PyTorch, TensorFlow, Keras) $\rightarrow$ no feature engineering

Units

A neural unit



The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation on them, and produces an output.

Bias

a neural unit is taking a weighted sum of its inputs, with one additional term in the sum called a bias term

$$z = b + \sum_i w_i x_i$$

expressing this weighted sum using vector notation: replacing the sum with dot product ($z \in \mathbb{R}$):

$$z = w \cdot x + b$$

Activation

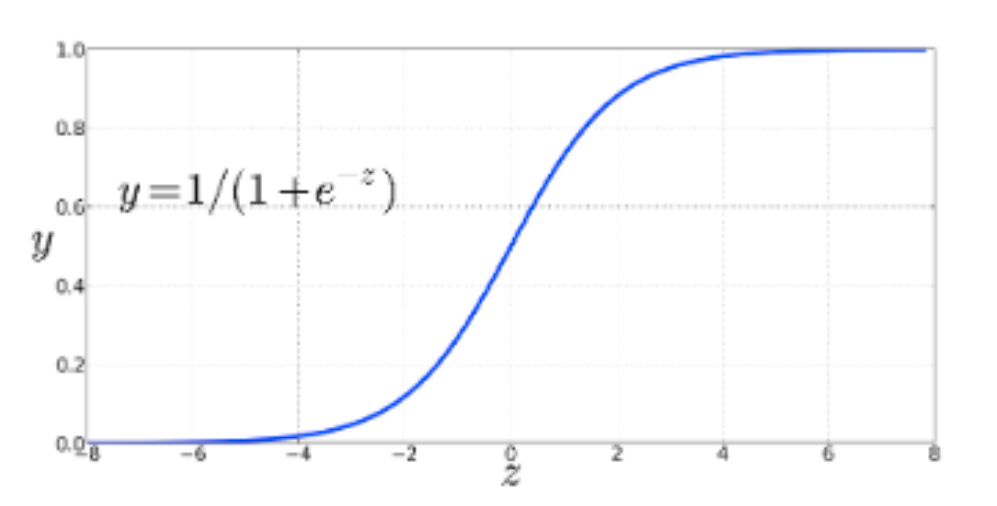
instead of using z, neural units apply a non-linear function f to $z \rightarrow$ the output of this function is the activation value for the unit a

$$y = a = f(z)$$

the final output of the network is y, and since here we have a single unit, y and a are the same

Non-linear functions – sigmoid

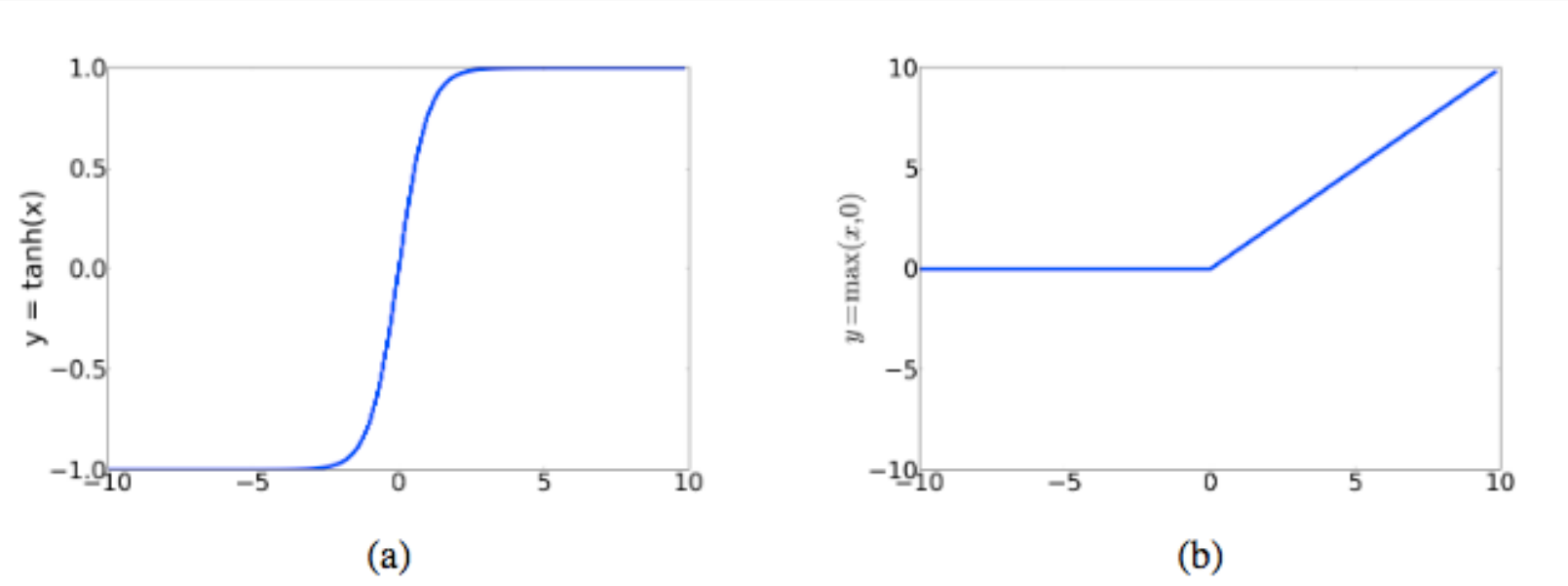
$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$



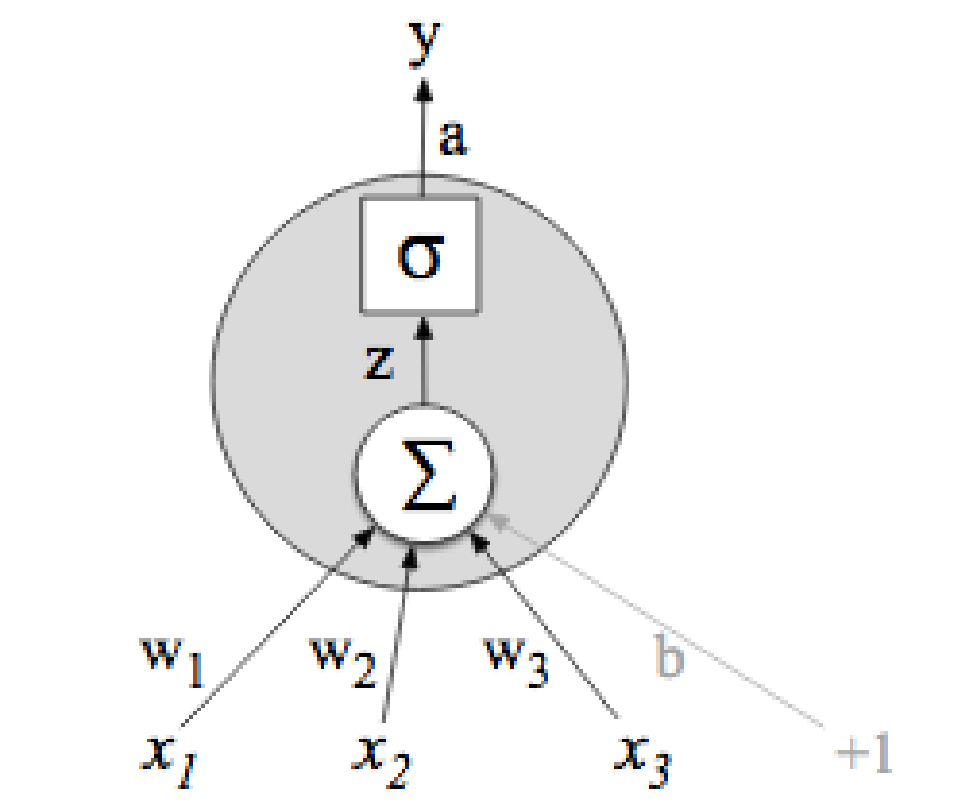
Non-linear functions – tanh and ReLU

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$y = max(x, 0)$$



Summary – a unit



Feedforward Neural Networks

A feedforward network

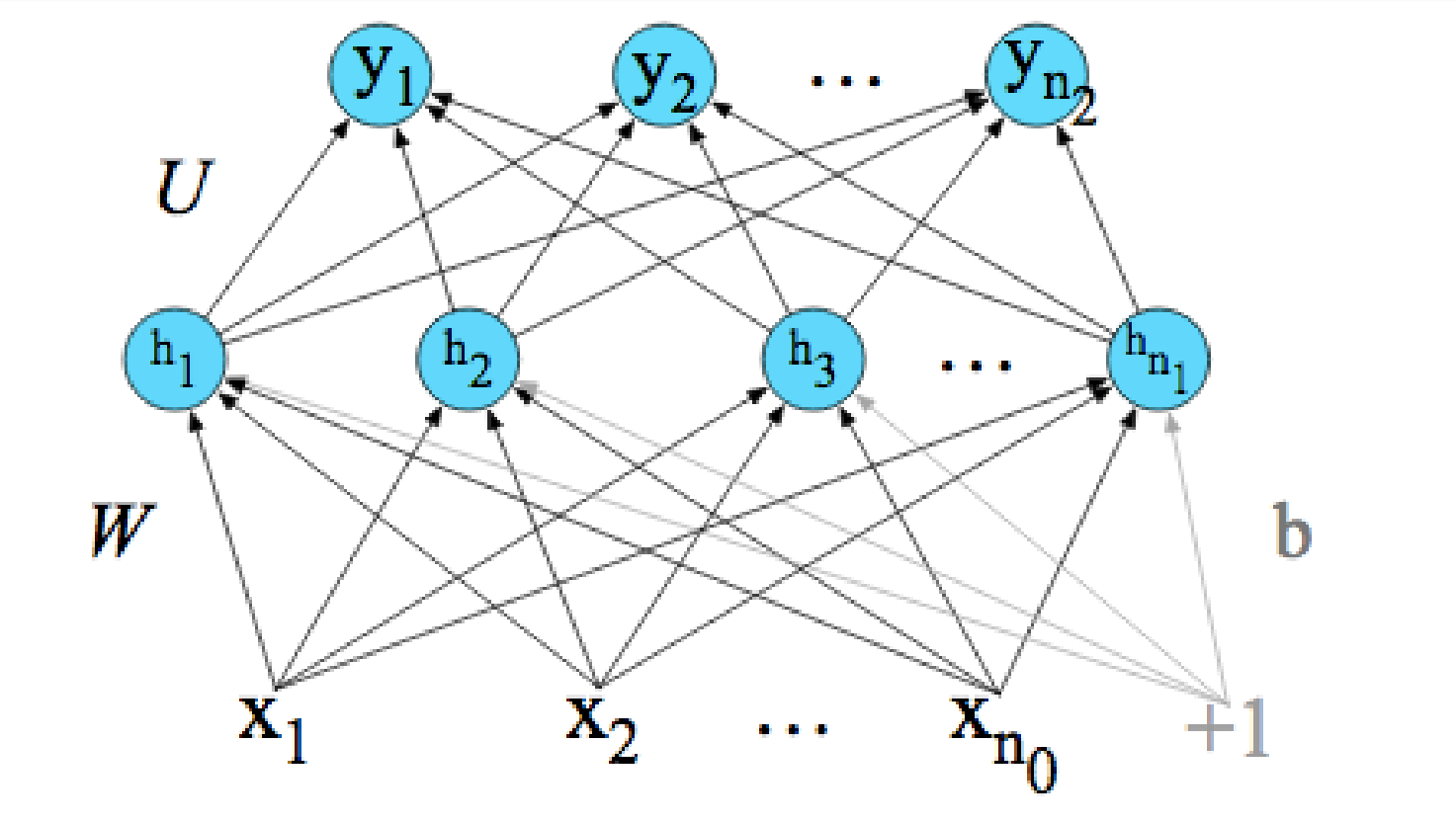
a feedforward network

is a multilayer network

- in which the units are connected with no cycles;
- the outputs from units in each layer are passed to units in the next higher layer, and
- no outputs are passed back to lower layers

(networks with cycles are called recurrent neural networks (RNNs))

Three kinds of nodes



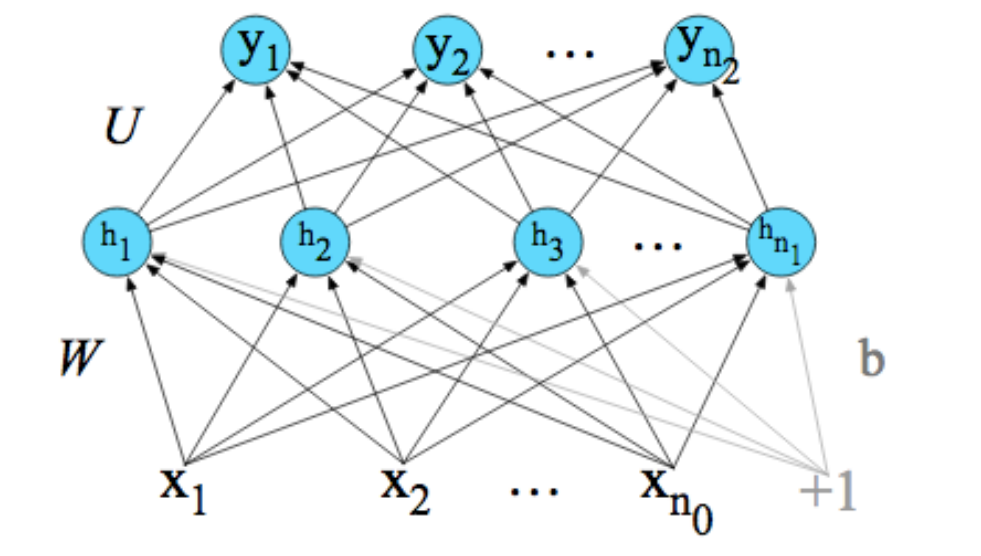
input units, hidden units, and output units

The hidden layer

- the hidden layer is formed of hidden units, each of which is a neural unit, taking a weighted sum of its inputs and then applying a non-linearity
- fully-connected: each hidden unit sums over all the input units

Weight matrix

We represent the parameters for the entire hidden layer by combining the weight vector w_i and bias b_i for each unit i into a single weight matrix W and a single bias vector b for the whole layer. Each element W_{ij} of the weight matrix W represents the weight of the connection from the ith input unit x_i to the jth hidden unit h_j.



The W matrix

	x_1	x_2	x_3
h_1	w_{11}	w_{12}	w_{13}
h_2	w_{21}	w_{22}	w_{23}
h_3	w_{31}	w_{32}	w_{33}
h_4	w_{41}	w_{42}	w_{43}

Matrix operations

3 steps:

1. multiplying the weight matrix by the input vector x
2. adding the bias vector b
3. applying the activation function g

$$h = \sigma(Wx + b)$$

- the number of inputs: n_0
- x is a vector of real numbers of dimension n_0: $x \in \mathbb{R}^{n_0}$
- the hidden layer has dimensionality n_1, so $h \in \mathbb{R}^{n_1}$
- $W \in \mathbb{R}^{n_1 \times n_0}$

The role of the output layer

- the resulting value h forms a representation of the input
- the role of the output layer: to take this representation and compute the final output
- the output can be a real-valued number, but it is rather a probability distribution across the output nodes

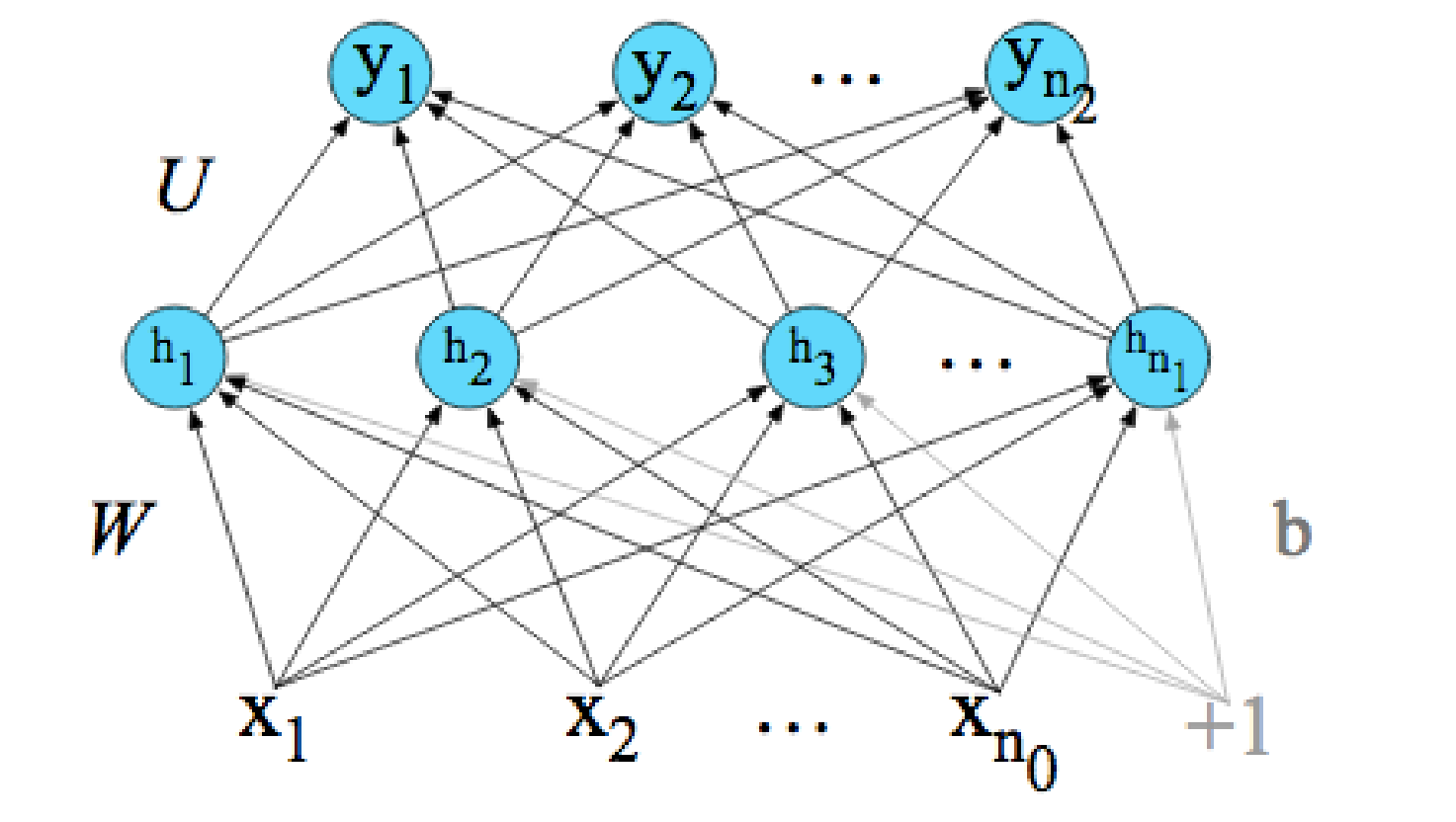
Intermediate output

- the output layer also has a weight matrix (U)
- some models don't include a bias vector b, so here we eliminate it
- the weight matrix U is multiplied by the vector h to produce the intermediate output z:

$$z = Uh$$

- $U \in \mathbb{R}^{n_2 \times n_1}$
- element U_{ij} is the weight from unit j in the hidden layer to unit i in the output layer

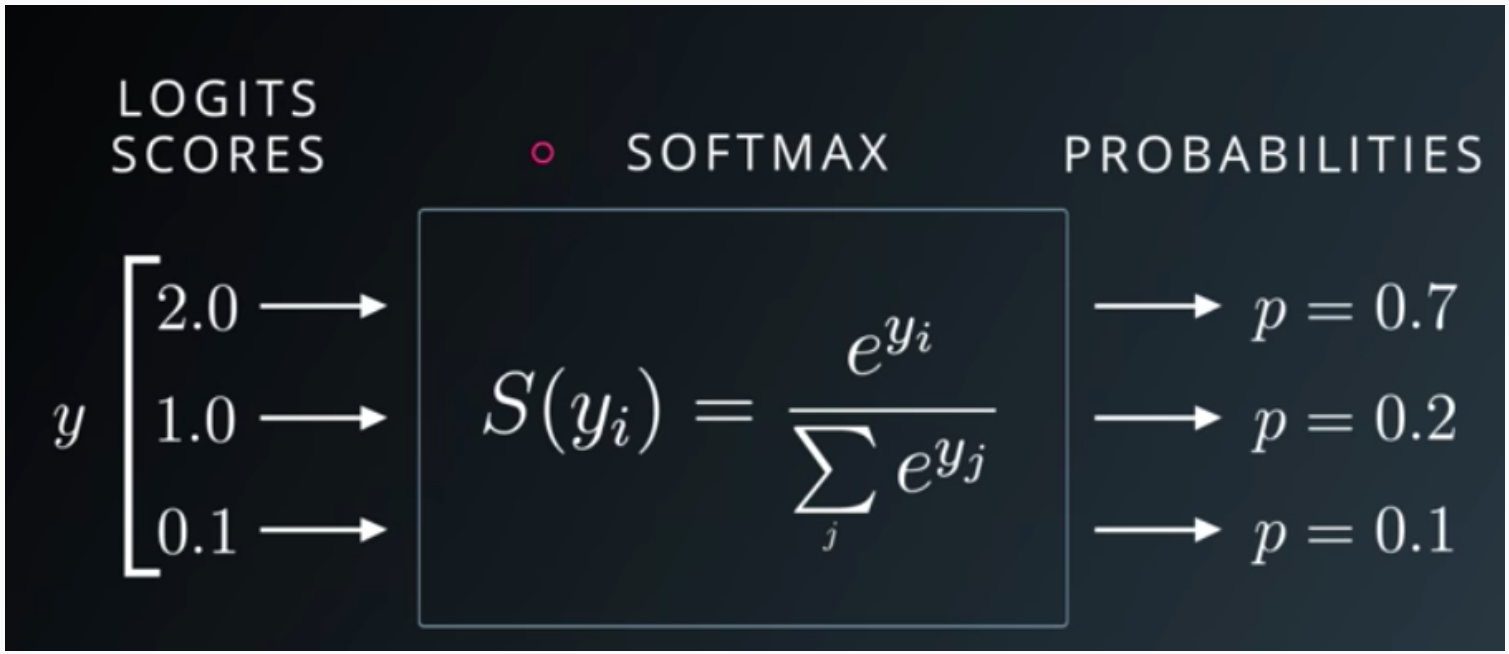
Ismétlés



The softmax function

converting a vector of real-valued numbers to a vector encoding a probability distribution:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} 1 \leq i \leq d$$



Summary – feedforward network

the final equations for a feedforward network with a single hidden layer, which takes an input vector x, outputs a probability distribution y, and is parameterized by weight matrices W and U and a bias vector b:

$$h = \sigma(Wx + b)$$
$$z = Uh$$
$$y = softmax(z)$$

activation functions:

- at the internal layers: ReLU or tanh
- at the final layer:
  - for binary classification: sigmoid
  - for multinomial classification: softmax

Training Neural Nets

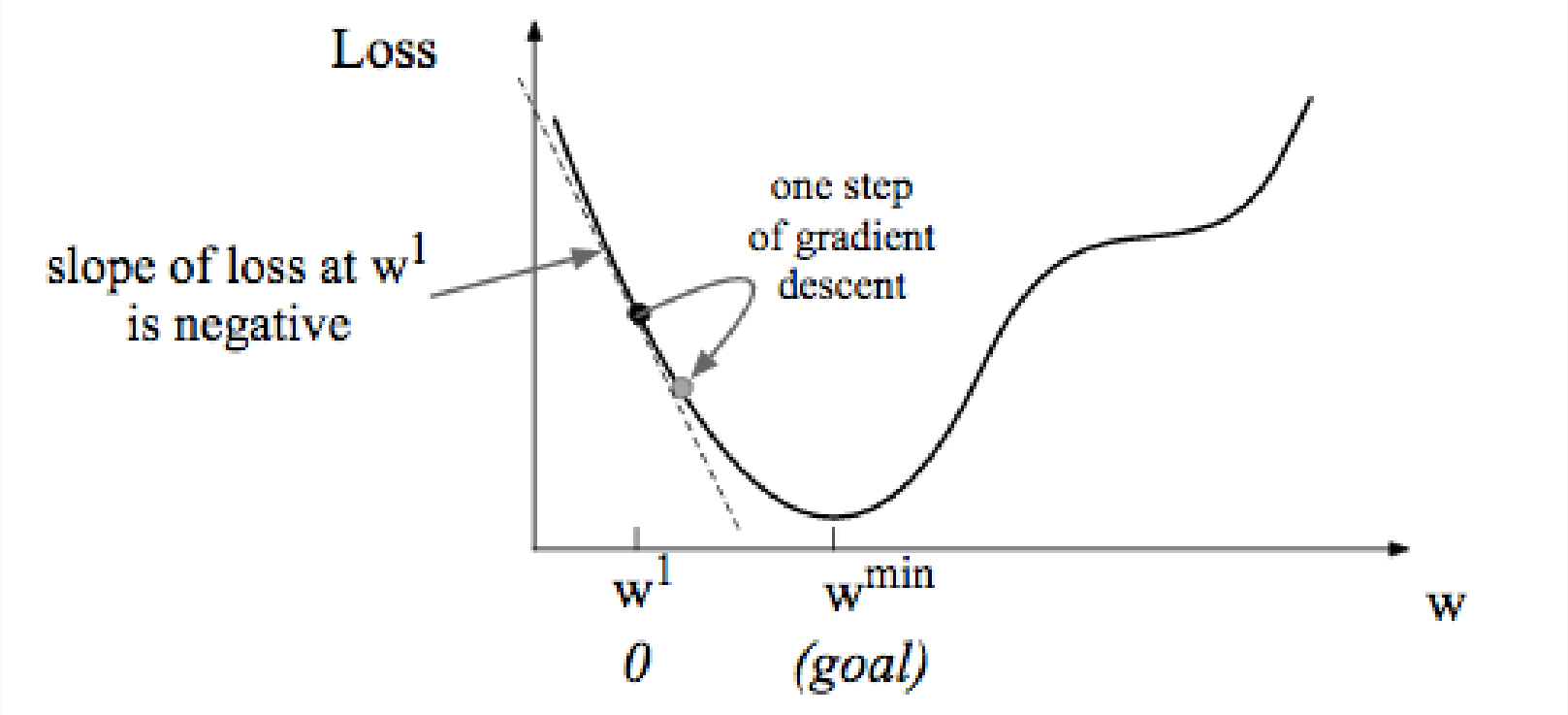
Supervised machine learning

- the correct output: y
- the system's estimate of the true y: $\hat{y}$
- the goal of the training procedure: to learn parameters $W^{[i]}$ and $b^{[i]}$ for each layer i that make $\hat{y}$ as close as possible to the true y

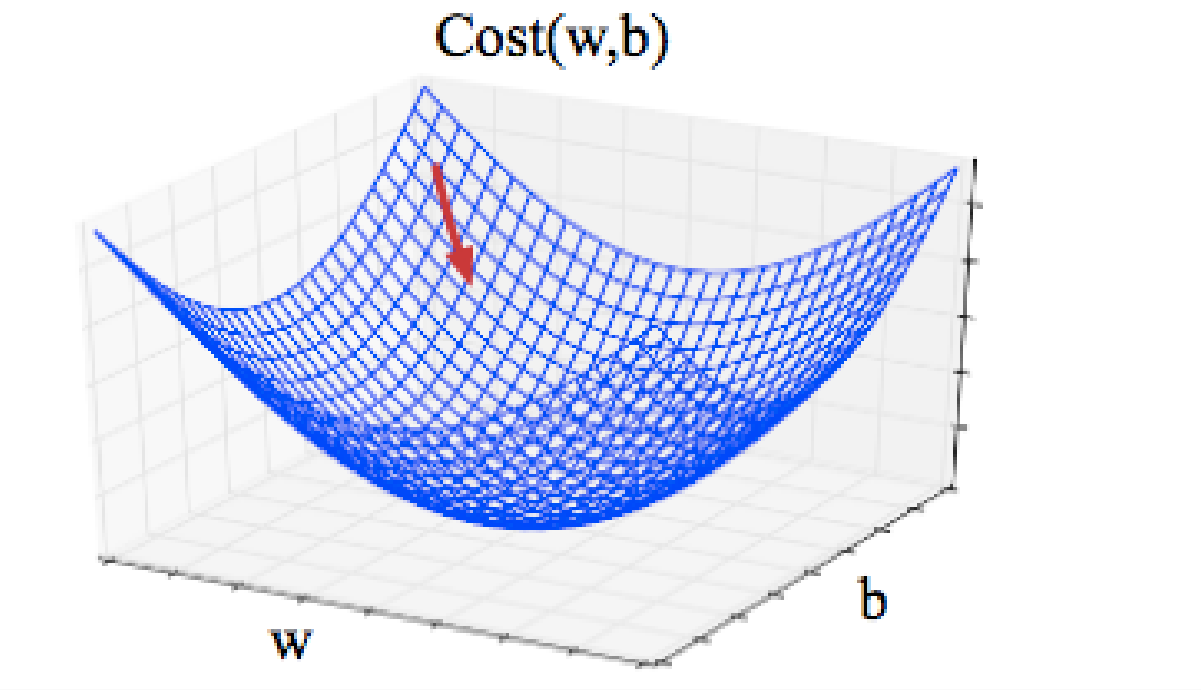
How to do that?

1. we need a loss function that models the distance between $\hat{y}$ and $y \rightarrow$ cross-entropy loss
2. we have to minimize the loss function $\rightarrow$ an optimization algorithm for iteratively updating the weights: gradient descent
3. we have to know the gradient of the loss function $\rightarrow$ error backpropagation

Computing the Gradient – one parameter



Computing the Gradient – two parameters



for more parameters → error backpropagation or backward differentiation → all parameters can be calibrated together

non-convex optimization problem with possible local minima

More details on learning

- to prevent overfitting $\rightarrow$ dropout: randomly dropping some units and their connections from the network during training
- tuning hyperparameters:
  - the number of layers
  - the number of hidden nodes per layer
  - the choice of activation functions
  - ...

Neural Language Models

Language modeling

language modeling:

predicting upcoming words from prior word context

neural language modeling (NLM) has advantages over n-gram language modeling:

- NLM does not need smoothing
- NLM can handle much longer histories
- NLM can generalize over contexts of similar words
- NLM has mush higher predictive accuracy

A feedforward NLM

a feedforward NLM is

a standard feedforward network that takes as input at time t a representation of some number of previous words $w_{t-1}, w_{t-2}, \ldots$, and outputs a probability distribution over possible next words

$$P(w_t|w_1^{t-1}) \approx P(w_t|w_{t-N+1}^{t-1})$$

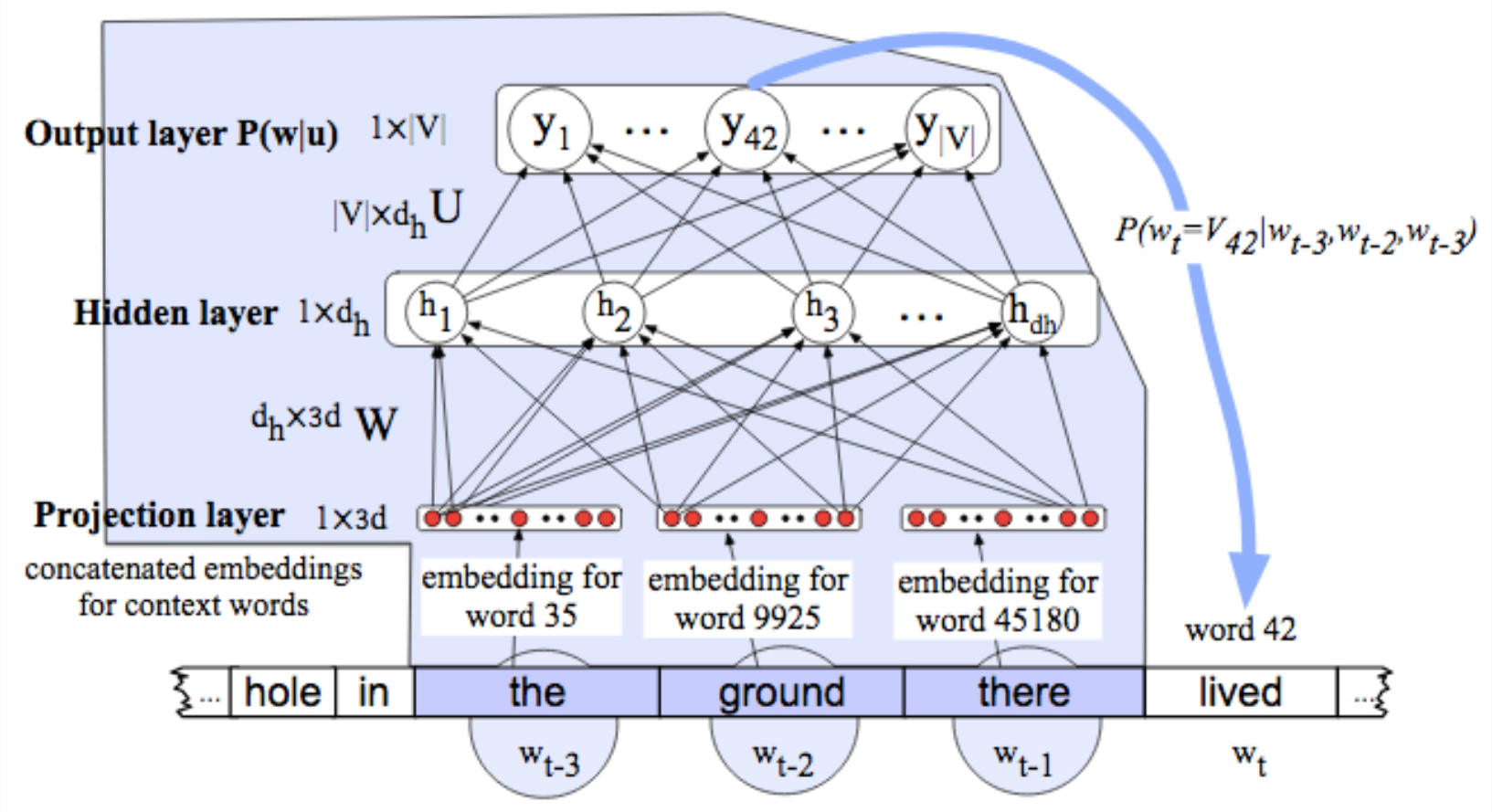
Embeddings

the prior context is represented by embeddings of the previous words $\rightarrow$ allows NLM to generalize to unseen data much better than n-gram models

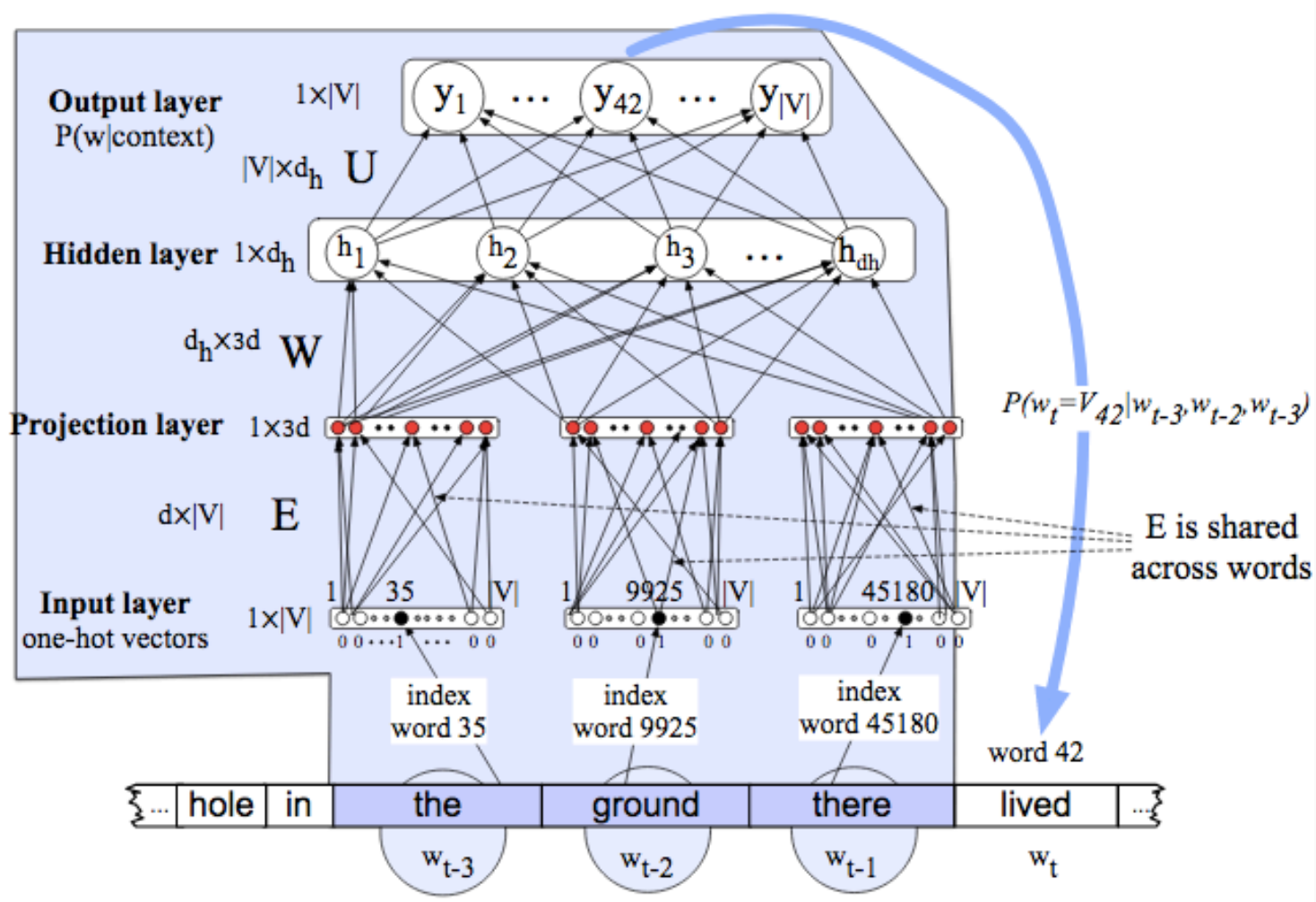
2 ways of using embeddings:

1. pretrained embeddings: we get the embeddings from an embedding dictionary E for each word in our vocabulary V
2. learning embeddings simultaneously with training the network

Using pretrained embeddings



Learning embeddings



Summary – NLM

the final equations for NLM:

$$e = (E_{x_1}, E_{x_2}, ..., E_x)$$
$$h = \sigma(We + b)$$
$$z = Uh$$
$$y = softmax(z)$$

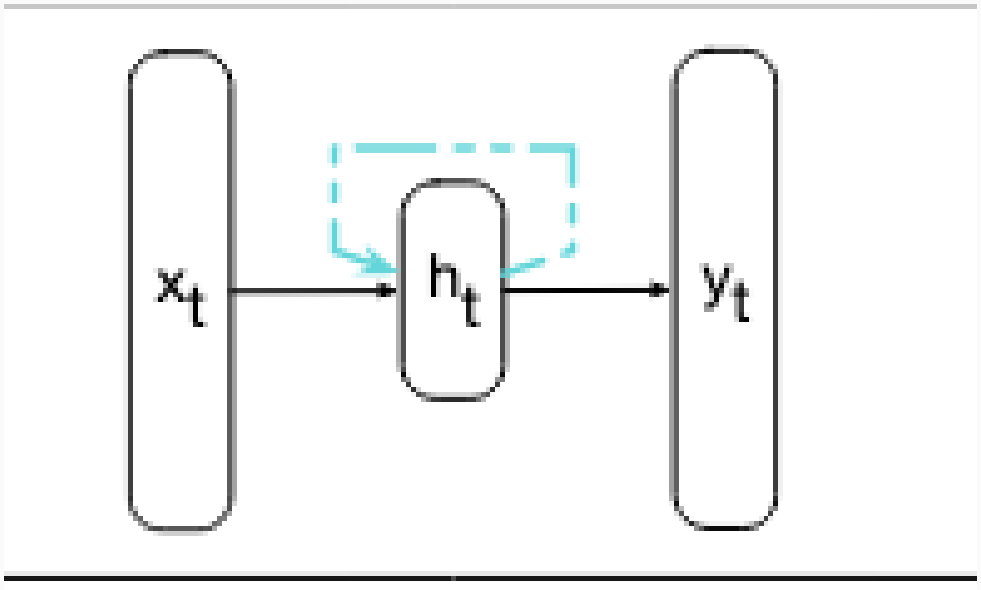
training the parameters to minimze loss will result both in an algorithm for language modeling (a word predictor) and a new set of embeddings

RNNs and LSTMs

Recurrent Neural Networks (RNNs)

- feedforward networks (FFNs) assume simultaneous access to all aspects of their input $\leftrightarrow$ RNNs deals directly with the sequential nature of language, offers a new way to represent the prior context
- a RNN is any network that contains a cycle within its network connections, meaning that the value of some unit is dependent on its own earlier outputs as an input

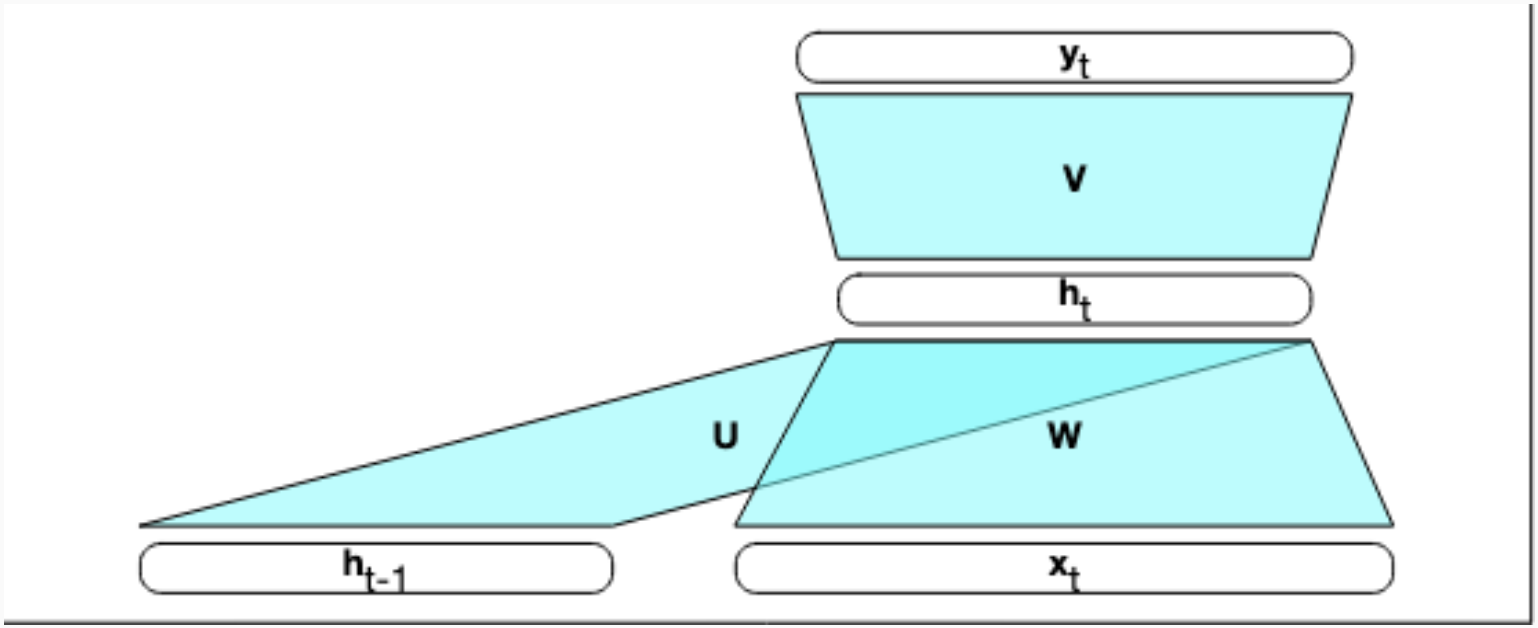
Simple RNN



simple RNN after Elman (1990)

the hidden layer from the previous time step provides a form of memory

Simple RNN illustrated as a FFN



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

$$\mathbf{y}_t = softmax(\mathbf{V}\mathbf{h}_t)$$

Training RNNs

3 sets of weights to update:

- **W**: the weights form the input layer to the hidden layer
- **U**: the weights from the previous hidden layer to the current hidden layer
- **V**: the weights from the hidden layer to the output layer

Backpropagation through time

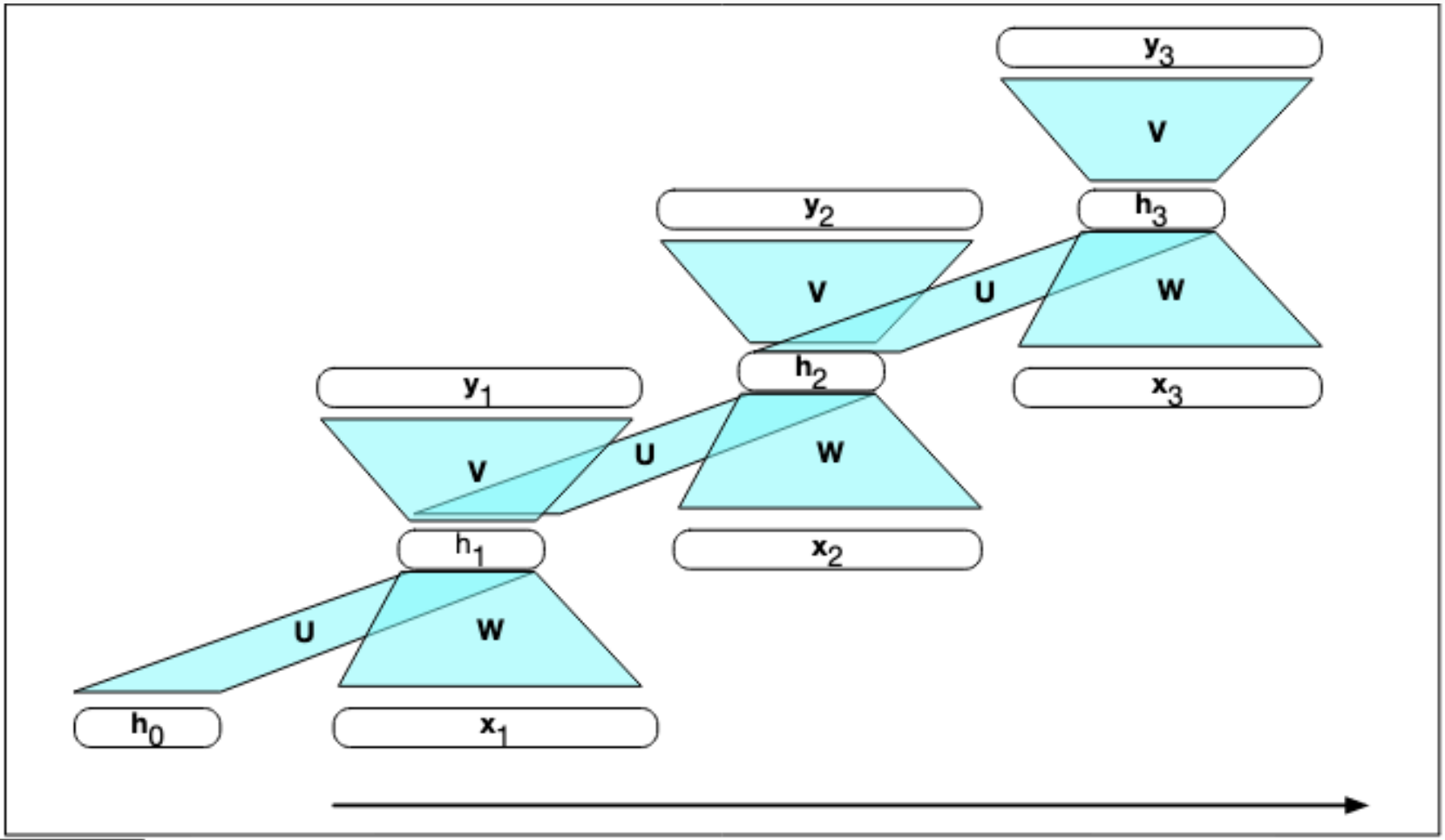


Figure 9.4 A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights **U**, **V** and **W** are shared across all time steps.

RNNs as Language Models

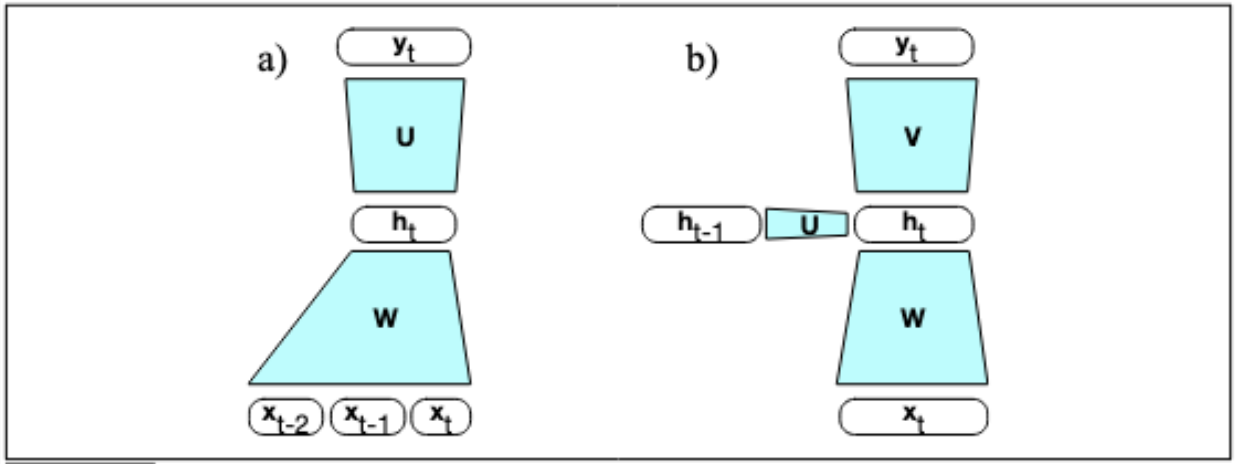


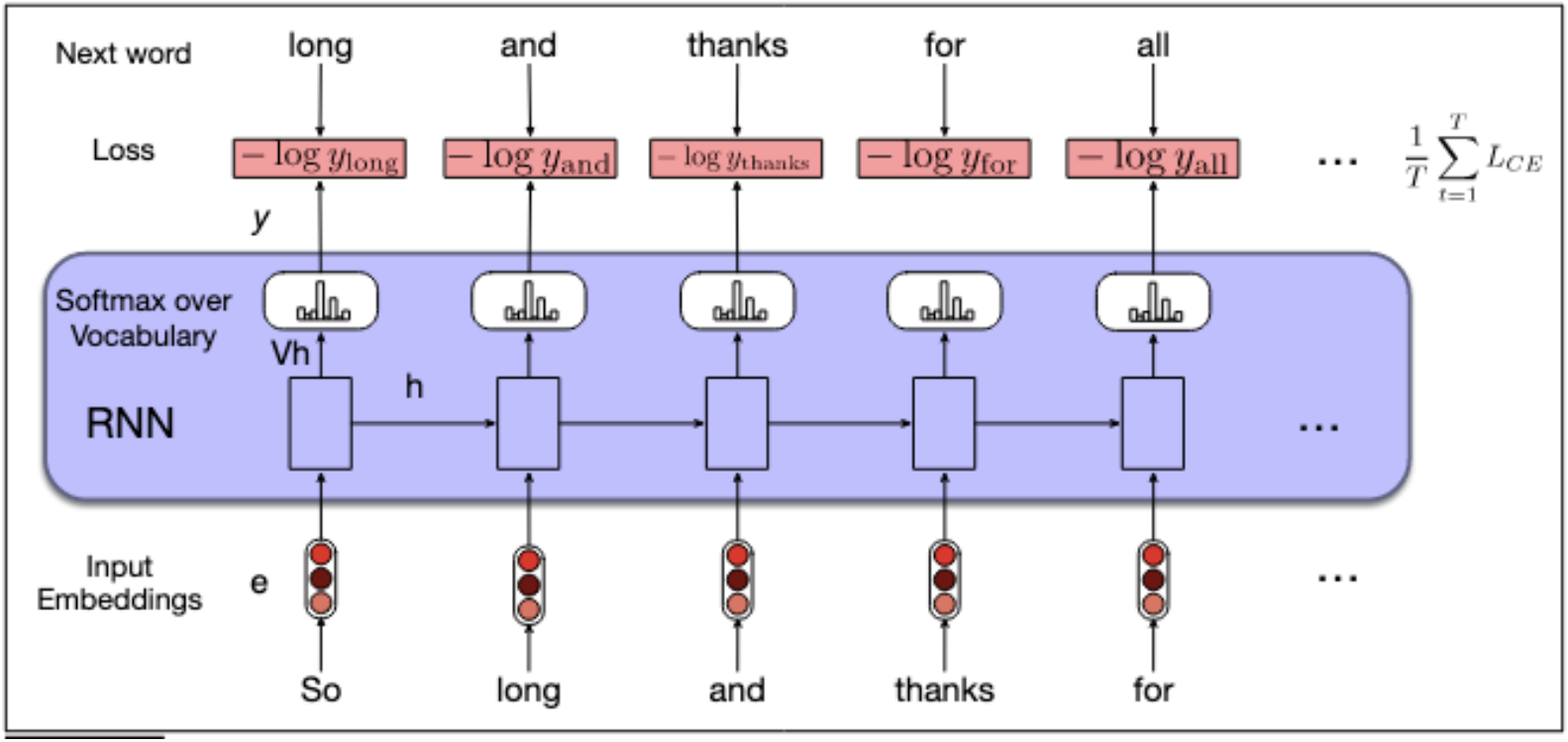
Figure 9.5 Simplified sketch of (a) a feedforward neural language model versus (b) an RNN language model moving through a text.

RNNs don't have the limited context problem, since the hidden state can in principle represent information about all of the prceding words all the way back to the beginning of the sequence

Training an RNN language model

- self-supervision algorithm: we don't need gold standard labels, the natural sequence of words is its own supervision
- **teacher forcing:** At each word position t of the input, the model takes as input the correct sequence of tokens $w_{1:t}$, and uses them to compute a probability distribution over possible next words. We move to the next word, and use the correct sequence of tokens $w_{1:t+1}$.

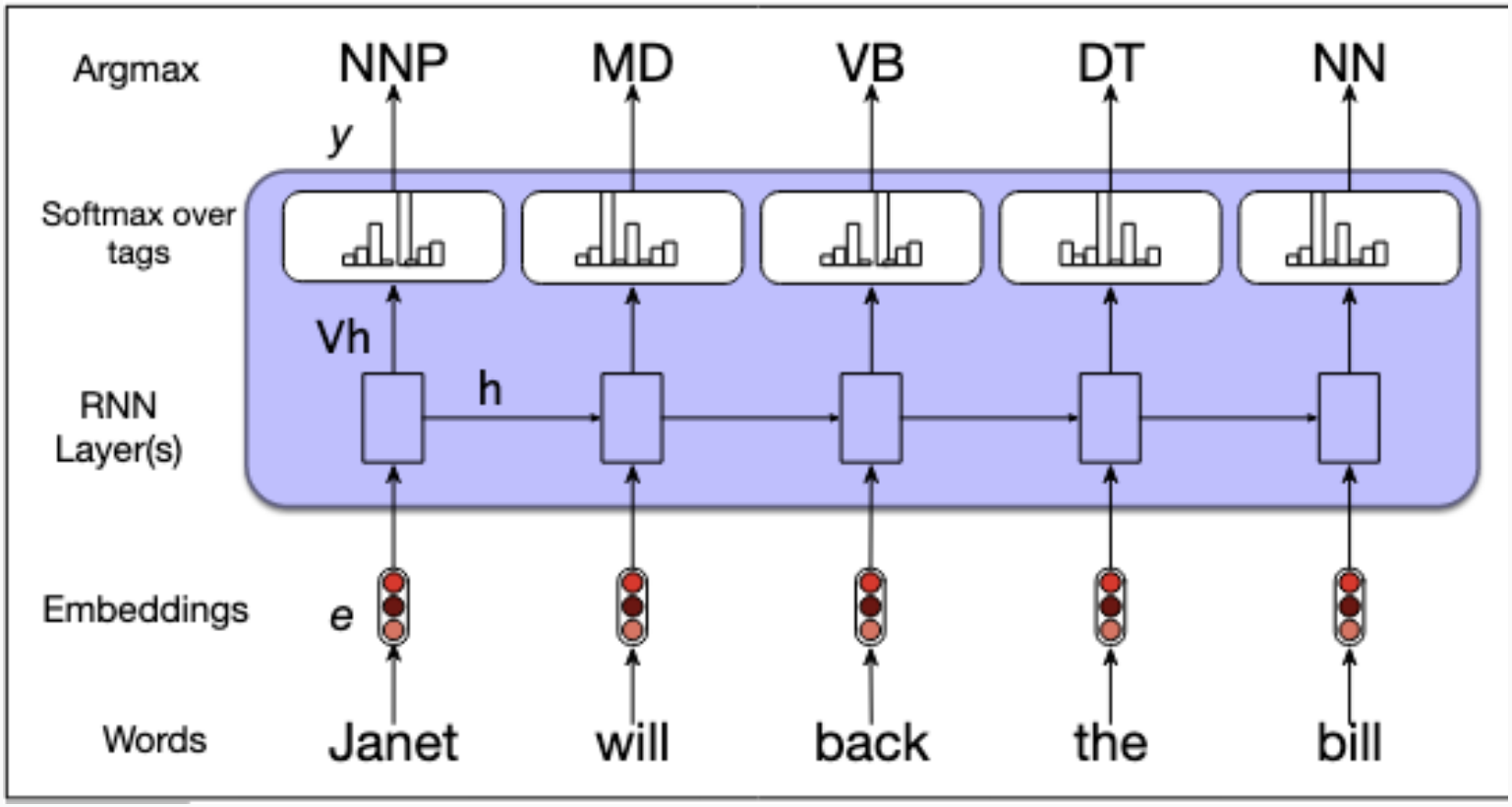
Training RNNs as language models



RNNs for Sequence Labeling I.

- to assign a label chosen from a fixed set of labels to each element of a sequence, e.g. POS tagging or NER
- input: pretrained word embeddings
- output: distribution over the tagset generated by a softmax layer
- forward inference: select the most likely tag from the softmax at each step
- cross-entropy loss

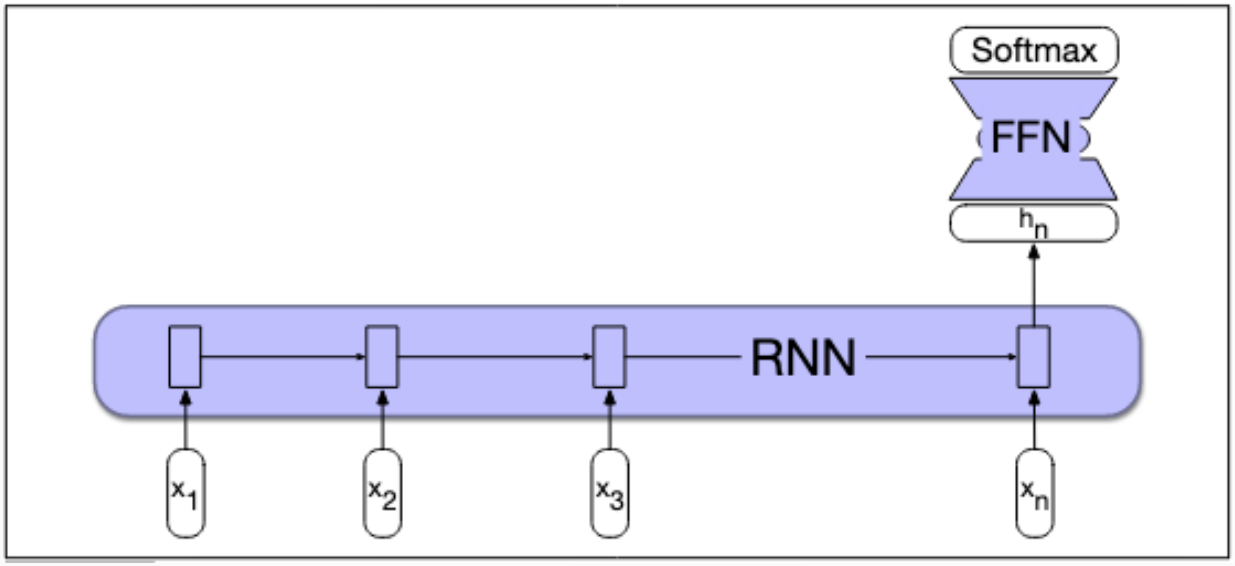
RNNs for Sequence Labeling II.



RNNs for Sequence Classification I.

- to classify entire sequences rather the tokens within them
- text classification, e.g. sentiment analysis, spam detection
- we pass the text through the RNN $\rightarrow$ we take the hidden layer for the last token of the text h_n to constitute a compressed representation of the entire sequence $\rightarrow$ we pass this h_n to a FFN that chooses a class via a softmax over the possible cases

RNNs for Sequence Classification II.



end-to-end training: the loss from the downstream application is backpropagated through the FFN and the RNN weights

Stacked RNNs

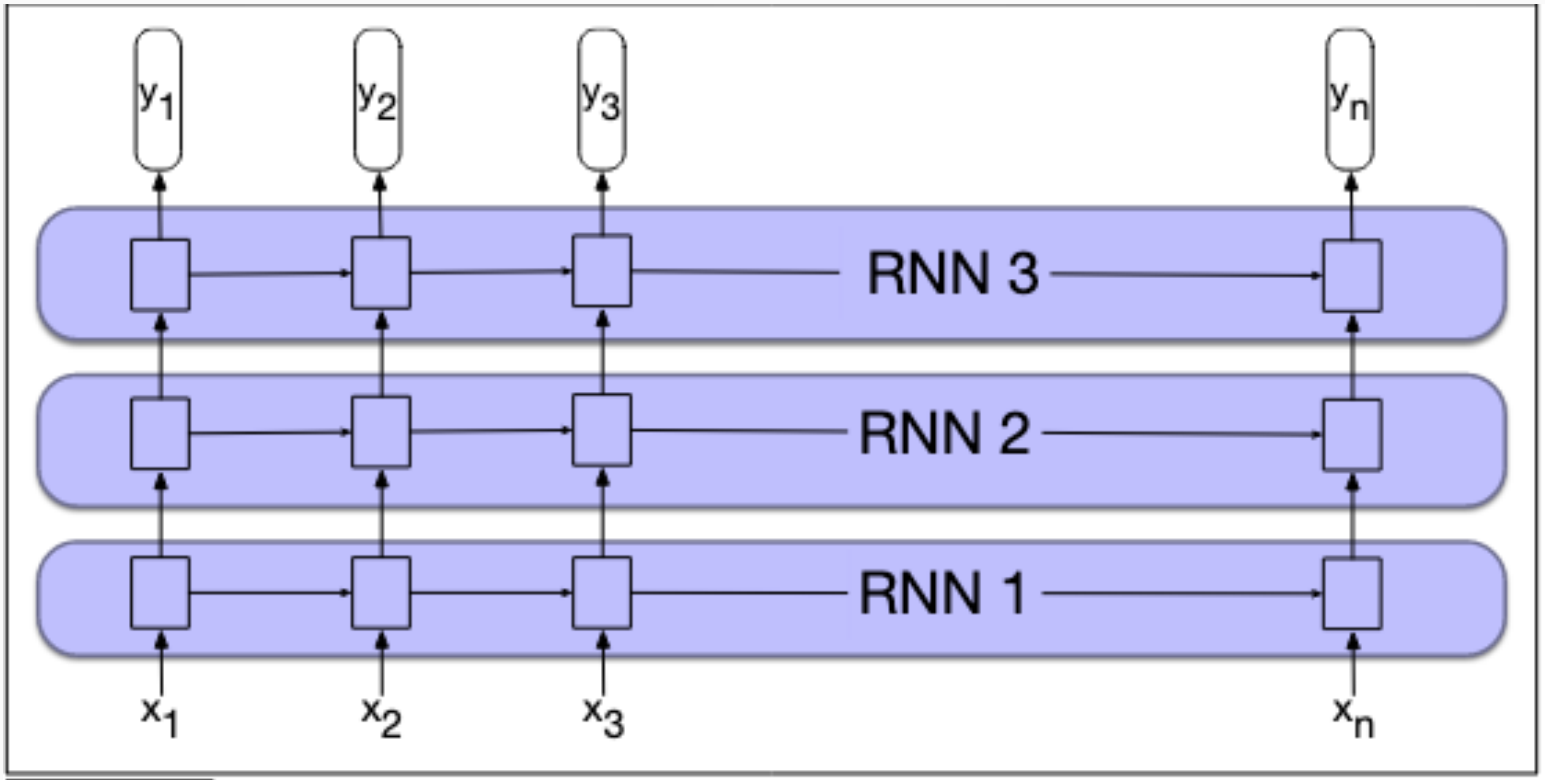


Figure 9.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

Bidirectional RNNs I.

- RNNs only use information from the left context, but in many applications we want to use words from the right context $\rightarrow$ run two RNNs: one left-to-right and one right-to-left, and concatenate their representations $\rightarrow$ bidirectional RNN
- effective for sequence classification

Bidirectional RNNs II.

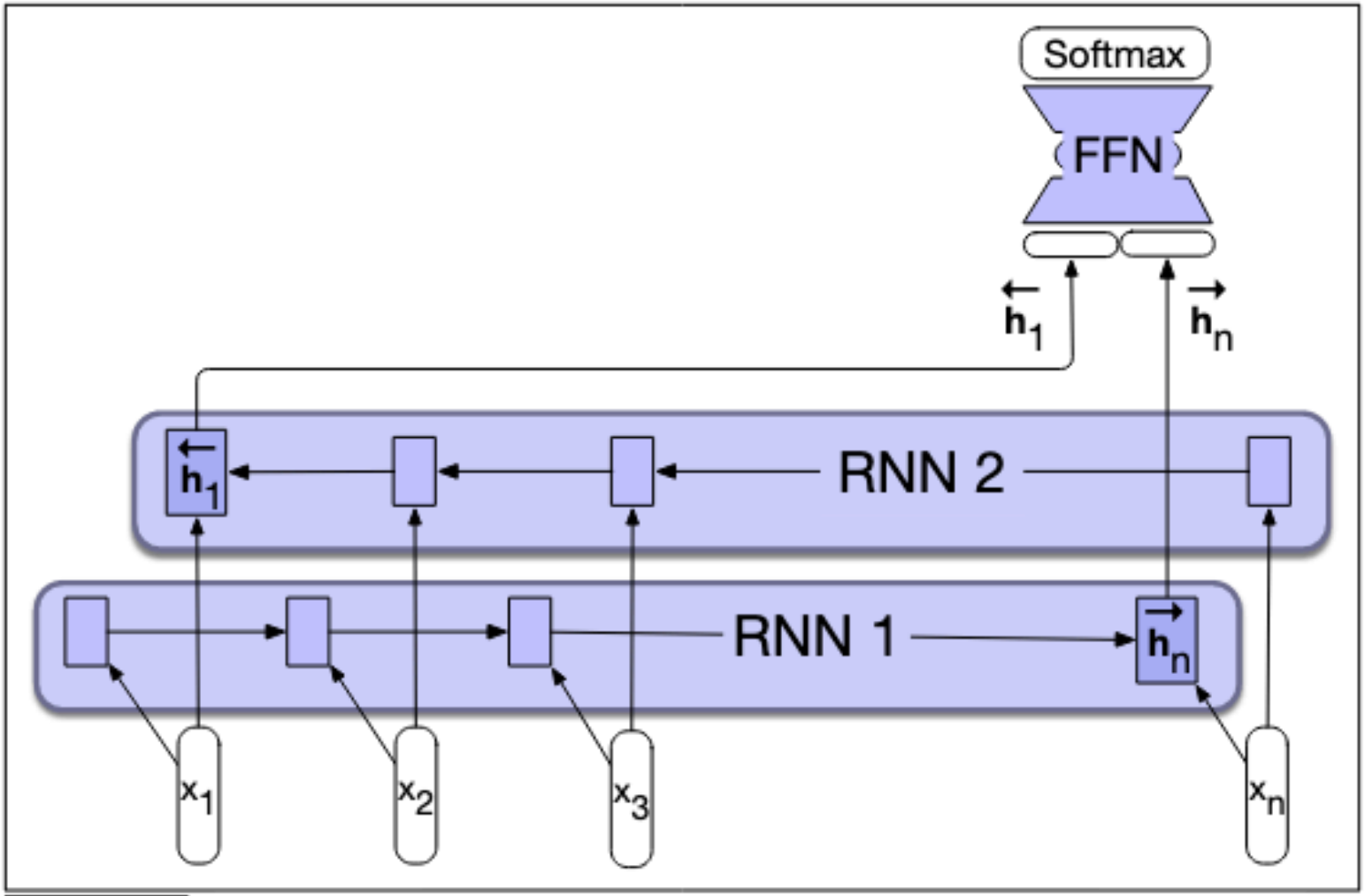


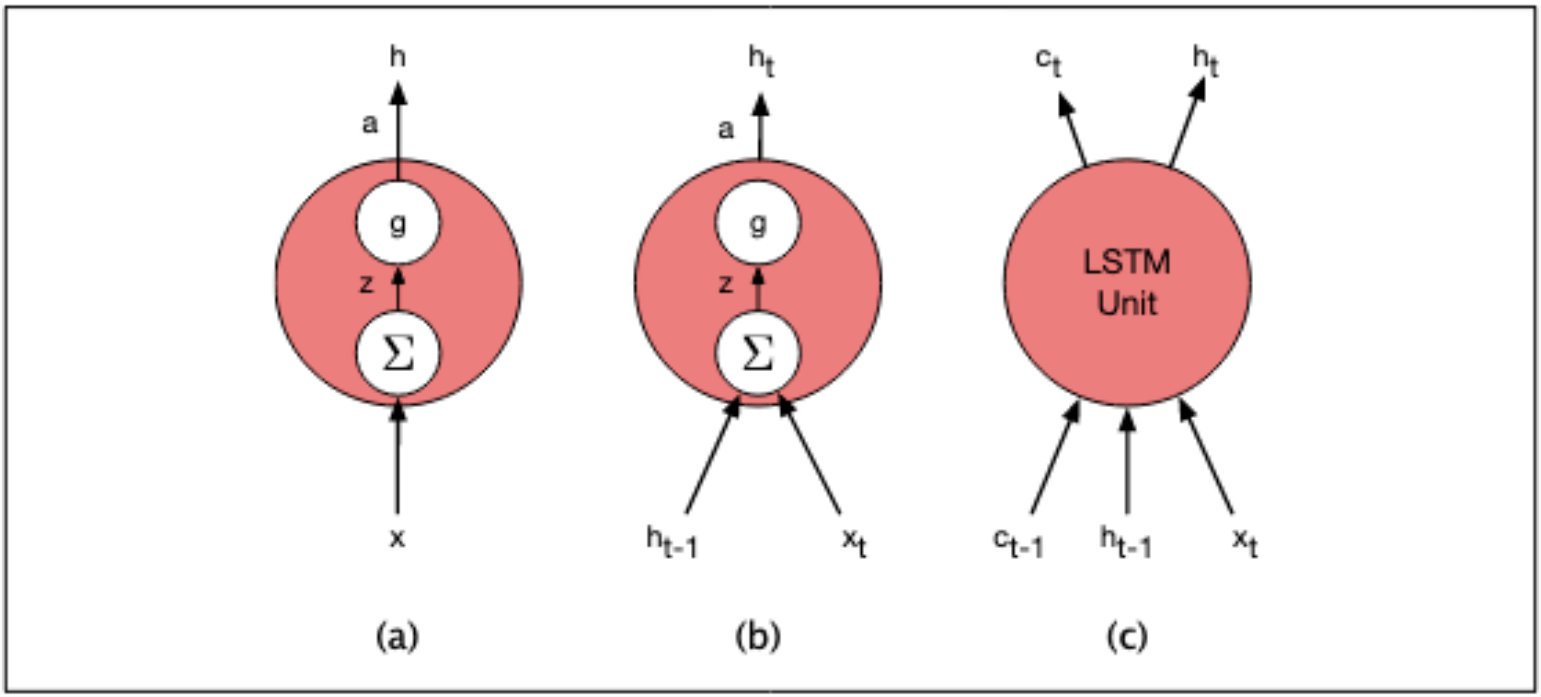
Figure 9.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

The LSTM

- despite having access to the entire preceding sequence, the information encoded in hidden states tned to be fairly local, but distant infomation is also important
- 2 difficulties with RNNs:
  - hidden layers are asked to perform two tasks simultaneously: provide information useful for the current decision, and updating and carrying forward information required for the future decisions
  - vanishing gradients: during the backward pass of training, hidden layers are subject to repeated multiplications $\rightarrow$ gradients are driven to zero

Long Short-term Memory (LSTM): adding a context layer and gated units

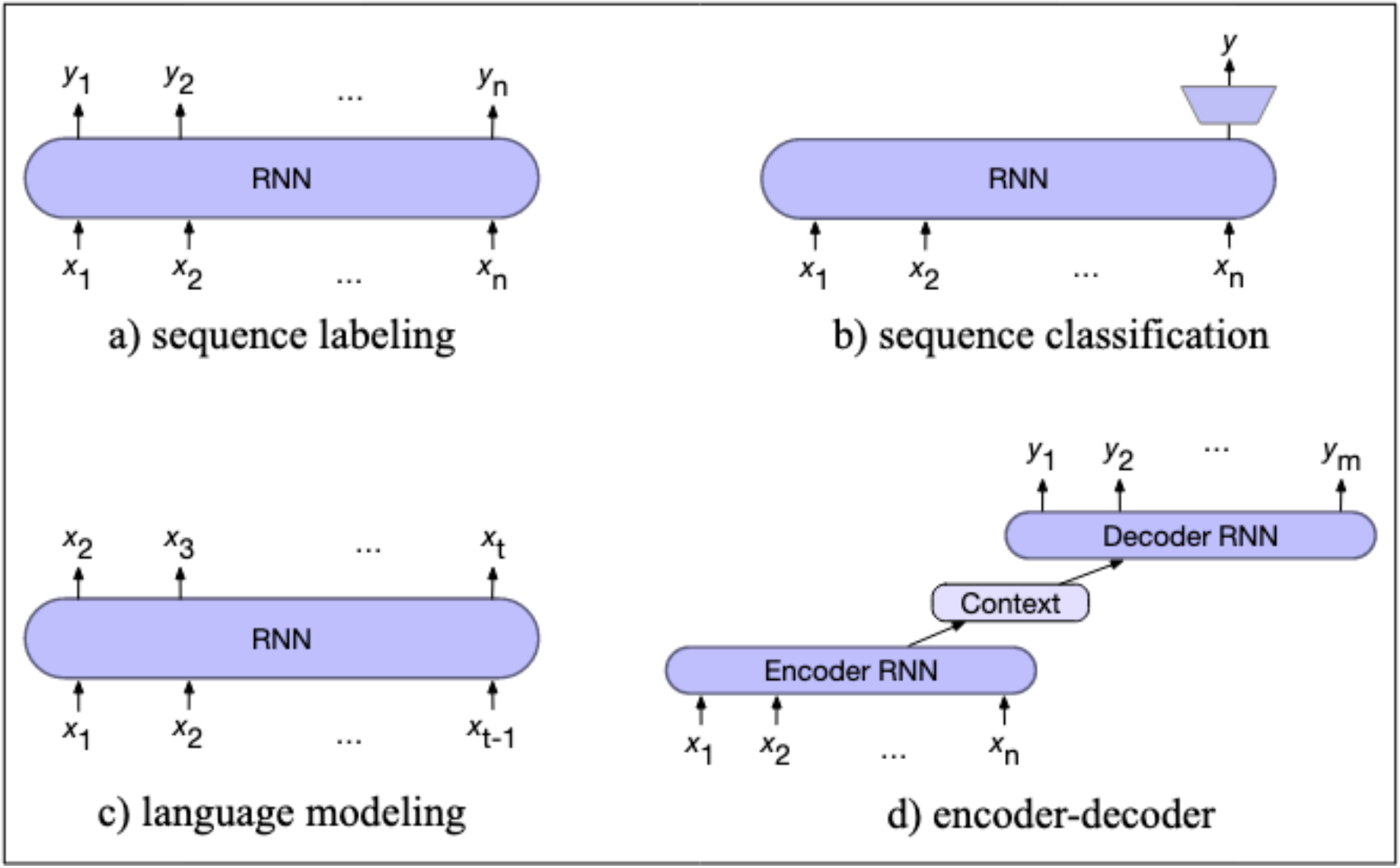
Units



The complexity of LSTM

- the complexity of LSTM is encapsulated within one processing unit $\rightarrow$ modularity and experiments with different architectures
- LSTM is a more complex version of an RNN
- RNN units can be substituted by LSTM units in all architectures $\rightarrow$ LSTMs rather than RNNs have become the standard unit for any modern system

Common RNN NLP Architectures



The Encoder-Decoder Model

- taking an input sequence and translating it to an output sequence with different lengths, without word-to-word alignment
- sequence-to-sequence, seq-to-seq models
- applied to machine translation, question answering, text summarization, dialogue systems
- the encoder network takes an input sequence and creates a contextualized representation of it $\rightarrow$ context
- the decoder network generates a task-specific output sequence
- encoders and decoders can be any architectures

Sentence separator

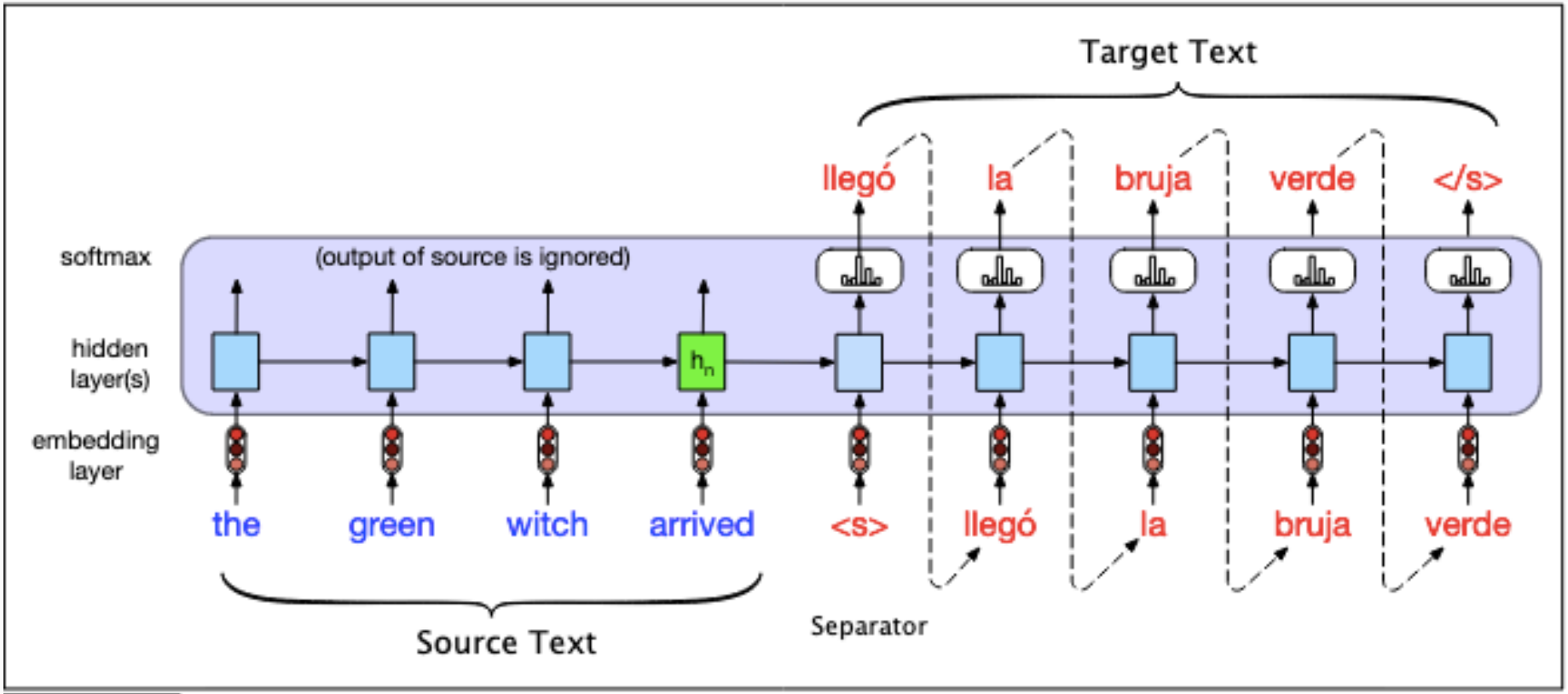


Figure 9.17 Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

Translating a sentence

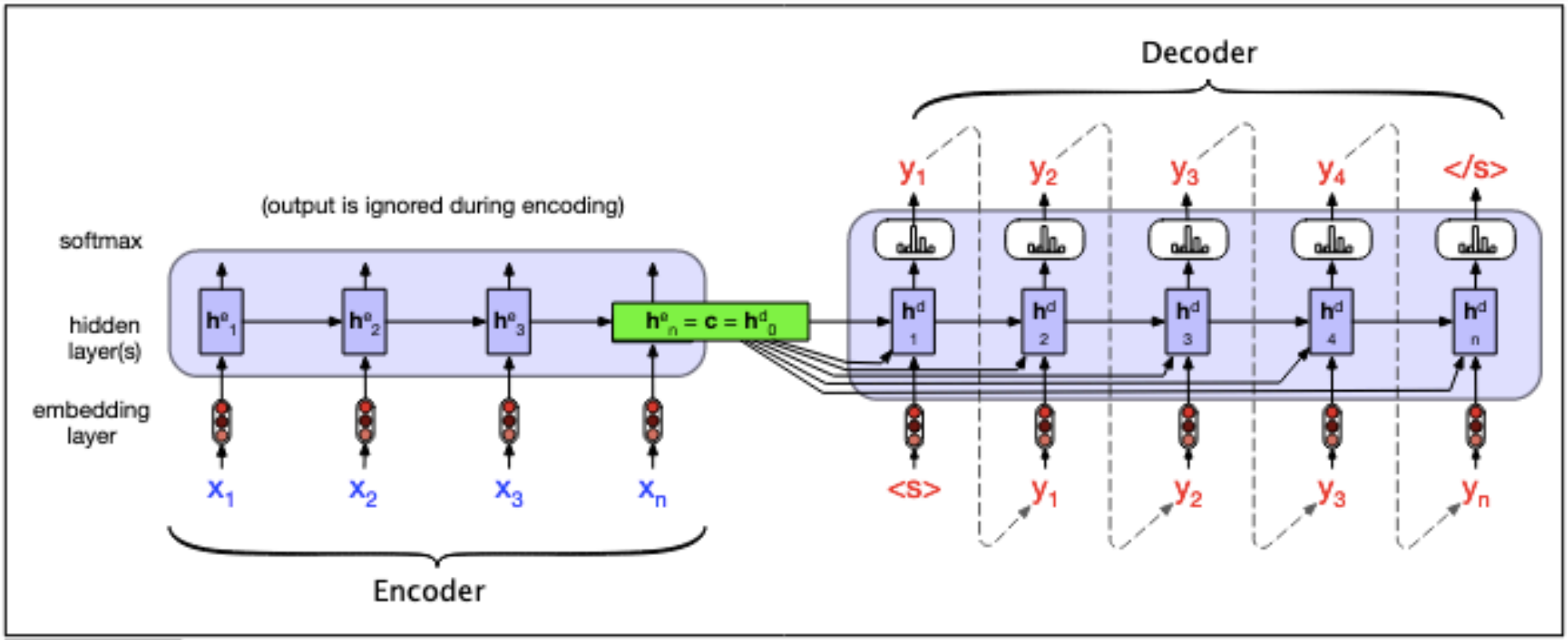


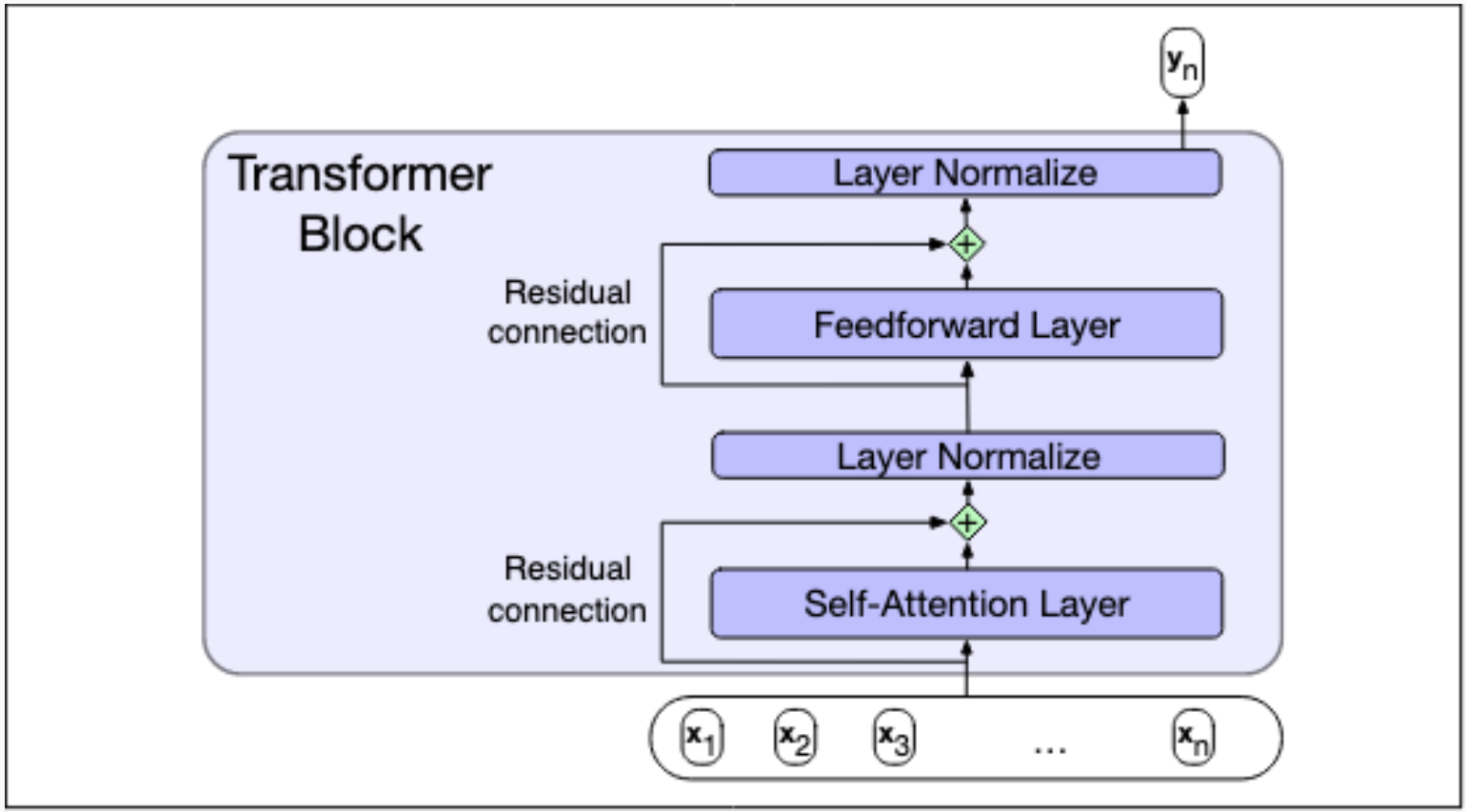
Figure 9.18 A more formal version of translating a sentence at inference time in the basic RNN-based encoder-decoder architecture. The final hidden state of the encoder RNN, h_n^e, serves as the context for the decoder in its role as h_0^d in the decoder RNN.

Attention

- bottleneck of the encoder-decoder model: information at the beginning of the sentence , especially for long sentences, may not be equally represented in the context vector
- the idea of attention: to create the vector c by taking the weighted sum of all the encoder hidden states
- the weights *attend to* a particular part of the source text that is relevant for the token the encoder is currently producing
- attention replaces the static context vector with one that is dinamically derived from the encoder hidden states
- the simplest such score: dot-product attention $\rightarrow$ relevance as similarity

Transformers and Pretrained Language Models

Transformers



transformers are made up of stacks of transformer blocks

Self-attention layer

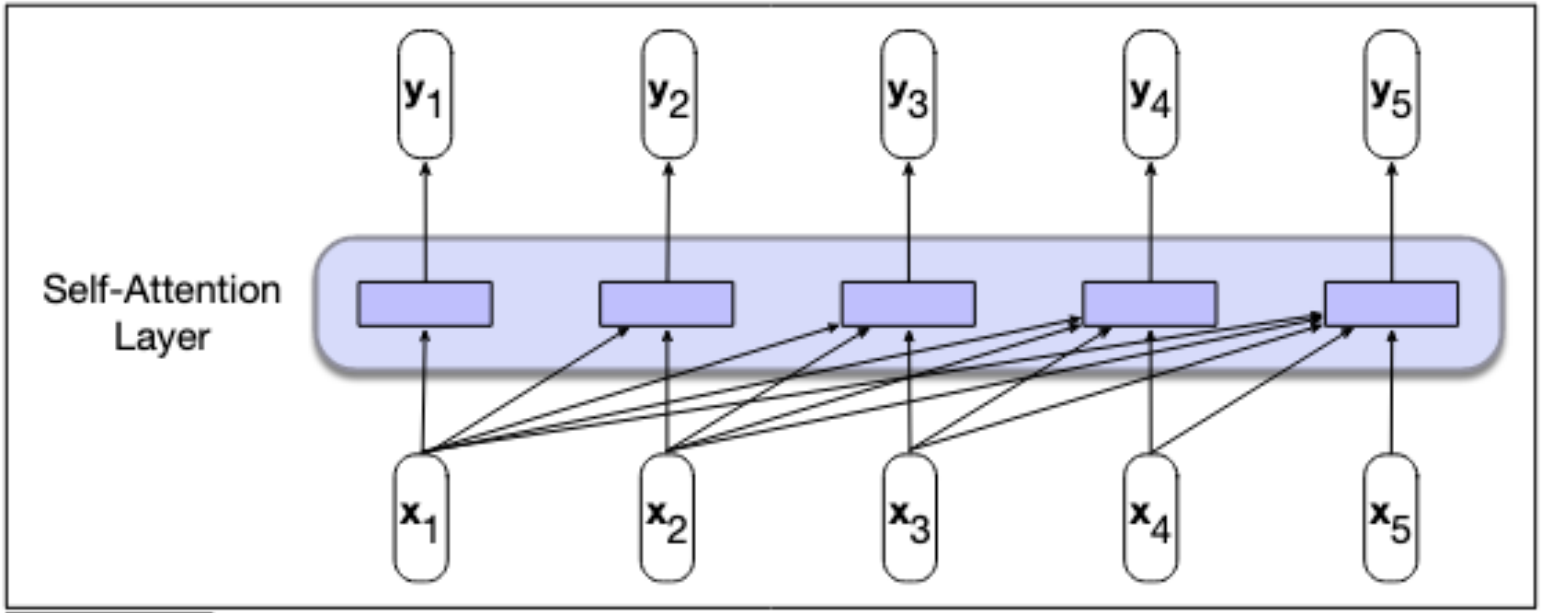


Figure 10.1 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be performed in parallel.

Transformers as Language Models

Training:

- teacher forcing: we force the system to use the gold target token rather than allowing it to rely on the decoder output
- the final transformer layer produces an output distribution over the entire vocabulary → cross-entropy loss for each token

→ autoregressive generation or causal LM generation → generative AI

Sampling technique

adapted from the n-gram language modeling to a neural context:

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

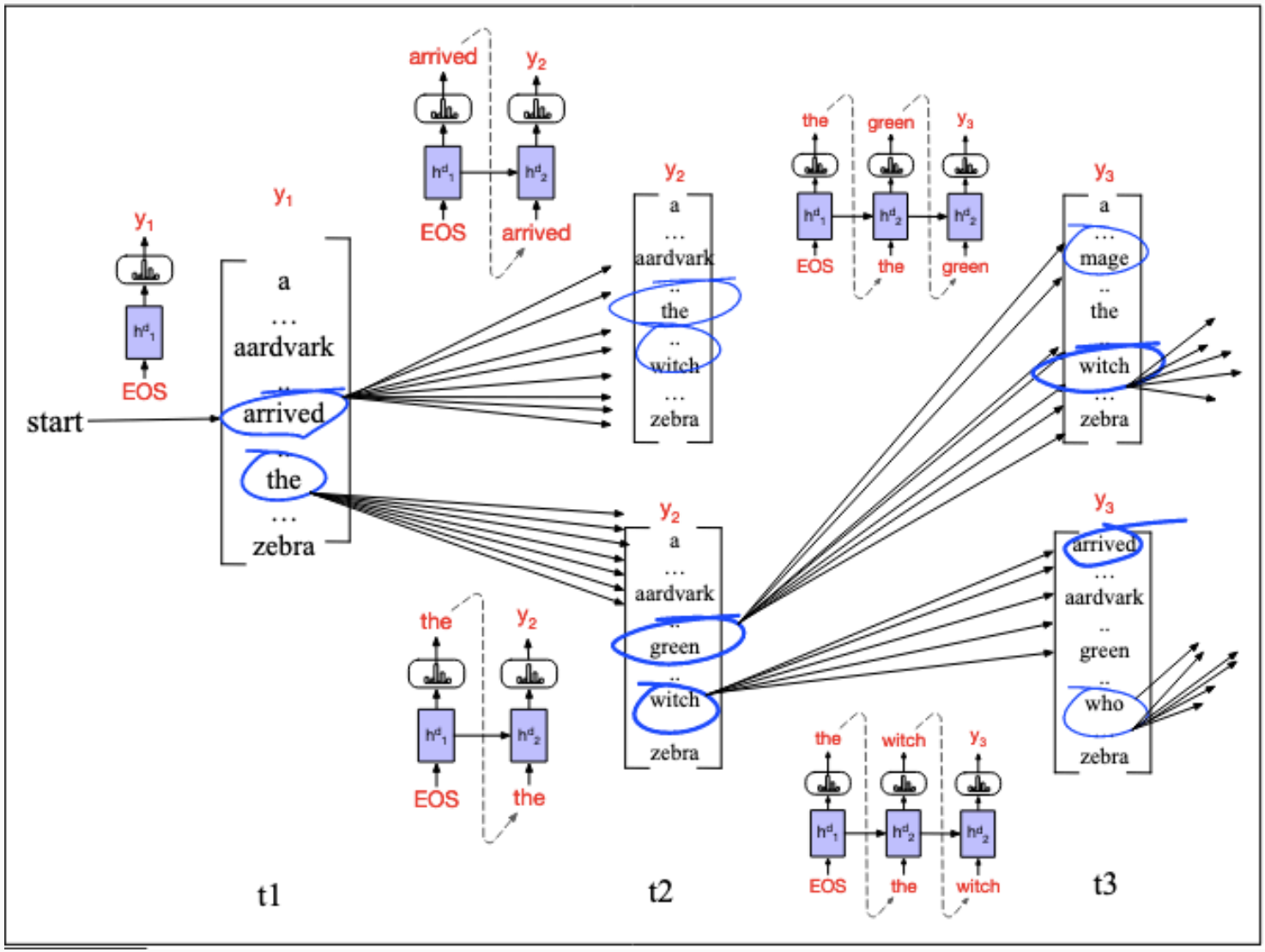
Choosing the output y

- computing a softmax over the set of possible outputs $\rightarrow$ choosing the highest probability token $\rightarrow$ greedy decoding
- locally optimal $\neq$ globally optimal $\rightarrow$ Viterbi search or beam search

beam search

- instead of choosing the best token, we keep k possible tokens $\rightarrow$ beam width
- we select the k best from the softmax output
- each of the k best hypotheses is passed to decoders each generating a softmax over the entire vocabulary
- each of these hypotheses is scored by the product of the probability of current word choice multiplied by the probability of the path that led to it $\rightarrow$ choosing the k best hypotheses again

Beam search


start
EOS
y1
a
aardvark
arrived
the
zebra
arrived
y2
EOS
arrived
y2
a
aardvark
the
witch
zebra
the
green
y3
EOS
the
green
y3
a
mage
the
witch
zebra
the
y2
EOS
the
y2
a
aardvark
green
witch
zebra
the
witch
y3
EOS
the
witch
y3
a
arrived
aardvark
green
who
zebra
t1
t2
t3

Fine-Tuning and Masked Language Models

Bidirectional Transformer Encoder

- models we introduced so far: causal or left-to-right transformer models
- bidirectional transformers: allows the model to see the entire text, including the right and left context
- trained via masked language modeling
- the most widely used version is BERT (Devlin et al., 2019)

Bidirectional self-attention model

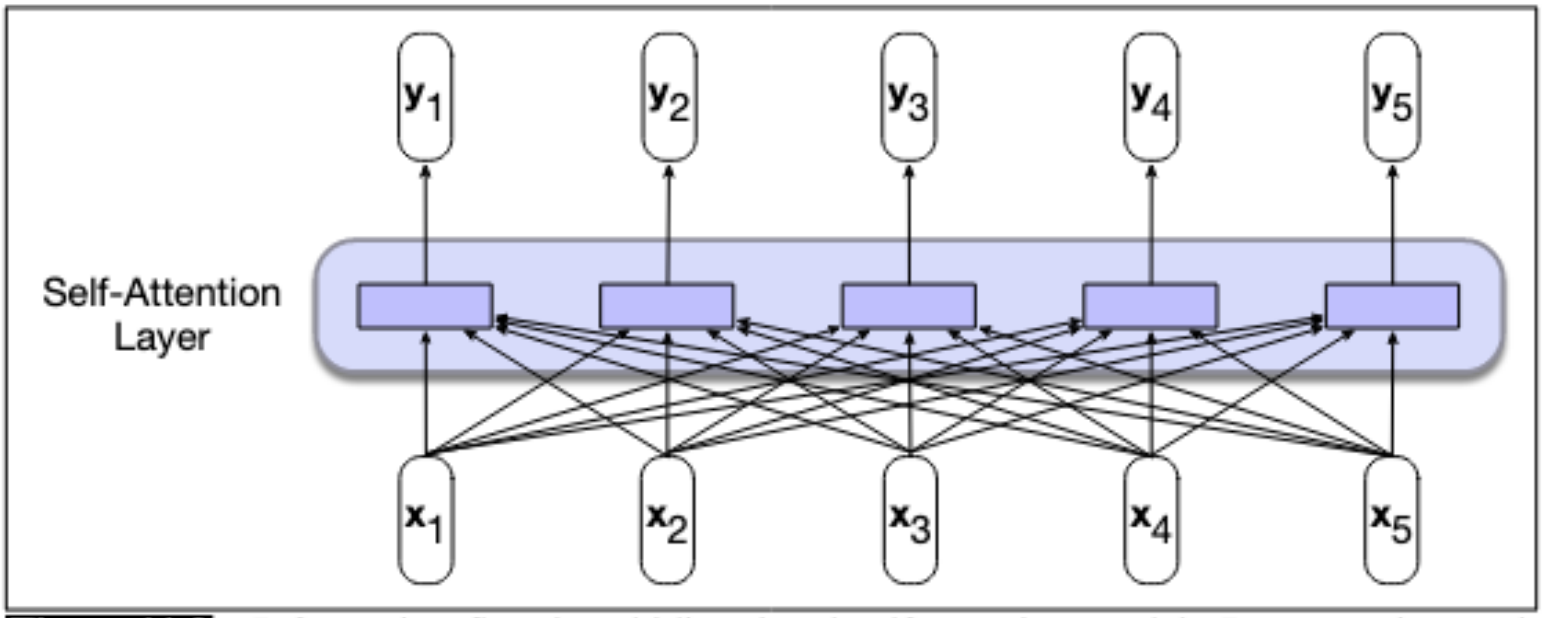


Figure 11.2 Information flow in a bidirectional self-attention model. In processing each element of the sequence, the model attends to all inputs, both before and after the current one.

Subword tokens

- BERT and its descendants use subword tokens rather than words
- every input sequence first has to be tokenized $\rightarrow$ all further steps on subwords
- need to map subwords back to words
- size of the input layer dictates the complexity of model $\rightarrow$ sufficient context, but not too long $\rightarrow$ 512 subword tokens

Training Bidirectional Encoders

instead of trying to predict the next word, the model learns to perform a fill-in-the-blank task:

```
Please turn _____ homework in.
```

- generate a probability dictribution over the vocabulary for each of the missing items $\rightarrow$ cross-entropy loss
- masking: masks, substitutions, reorderings, deletions, insertions

Masked Language Modeling (MLM)

Contextual Embeddings

- the result of the pretraining process consist of both learned word embeddings, as well as the all the parameters of the bidirectional encoder that are used to produce contextual embeddings for novel inputs
- contextual embeddings: vectors representing some aspects of the meaning of a token in context
- static embeddings represent the meaning of word types, contextual embeddings represent the meaning of word tokens

Transfer Learning through Fine-Tuning

transfer learning:

acquring knowledge from one task, and then applying it (transferring it) to solve a new task

fine-tuning:

- to create interfaces from pretrained language models to downstream applications
- applications on top of pretrained models through the addition of a small set of application-specific parameters
- using labeled data

Sequence labeling

- the final putput vector is passed to a classifier $\rightarrow$ produces a softmax distribution over the possible set of tags
- greedy approach: choosing the most probable tag OR Viterbi-algorithm OR the distribution can be passed to a CRF
- complication: subword tokenization

Irodalom

Irodalom

- Jurafsky 3rd edition 3.,7.,9.,10.,11. chapter: https://web.stanford.edu/~jurafsky/slp3/
- Francois Chollet: Deep Learning with Python. Manning, Shelter Island, 2018.: https://www.manning.com/books/deep-learning-with-python