Neural language models

Marco Kuhlmann

Department of Computer and Information Science

Limitations of statistical n-gram models

Goldberg § 9.3.2

- Scaling to larger *n*-gram sizes is problematic, both for computational reasons and because of data sparsity.
- Techniques for mitigating these issues require careful engineering and are not sufficiently flexible.

  smoothing, interpolation

- Without additional effort, *n*-gram models are unable to share statistical strength across "similar" words.

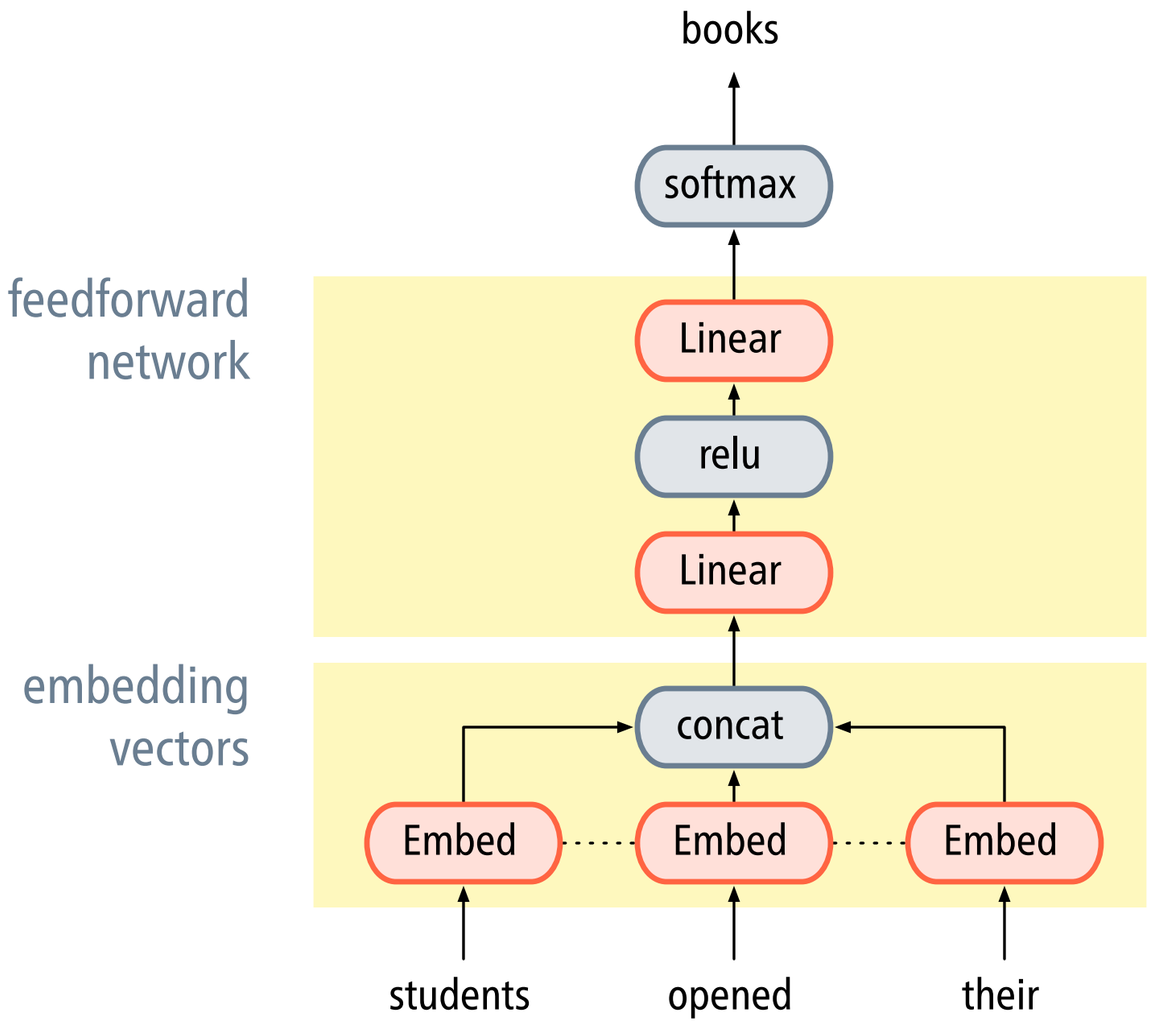
  Observations of *a red apple* do not affect estimates for *the yellow apples*.

A Neural Probabilistic Language Model

- Associate each word in the discrete vocabulary with a continuous **embedding vector**.
- Set up a neural network that computes the probability of a word sequence as a function of its embedding vectors.
- During training, simultaneously learn the embedding vectors and the weights of the neural network.

Bengio et al. (2003)

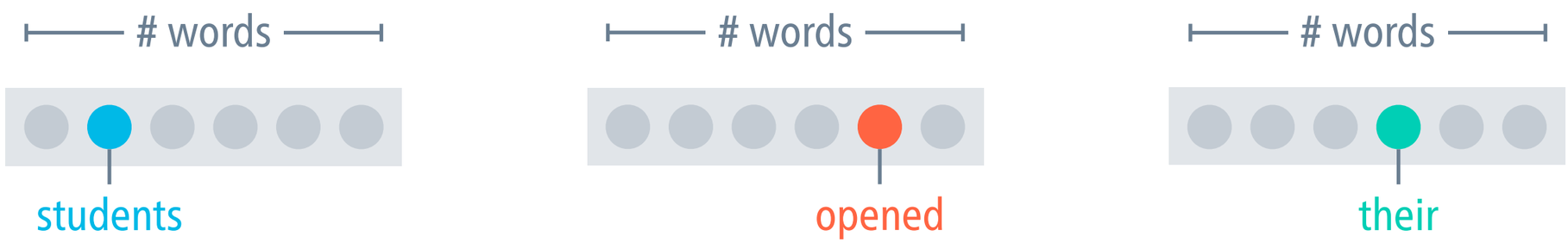
A neural four-gram model



Bengio et al. (2003)

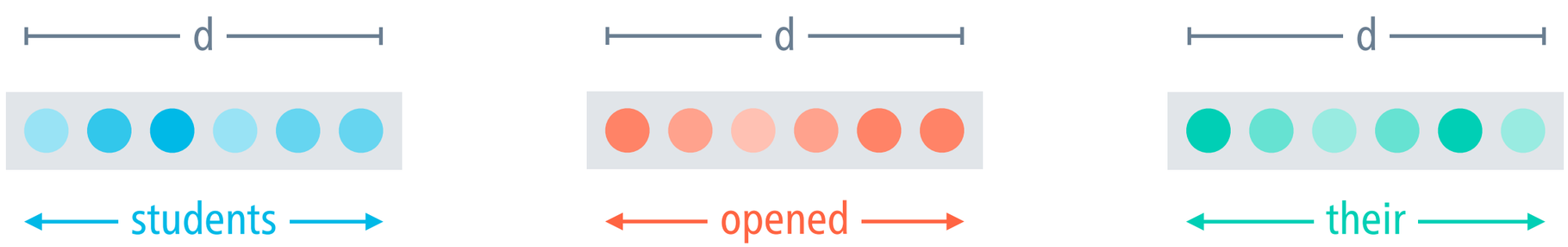
One-hot vectors

- To process words using deep learning libraries, we must represent them as vectors (lists of numbers).
- A simple way to do this is to use **one-hot vectors** – vectors in which all components but one are zero.



Embedding vectors

- The word embeddings used in the neural n-gram model are realised by **embedding layers**.
- An embedding layer implements a mapping from a discrete vocabulary to some d-dimensional vector space.



Embedding layers in PyTorch

```
vocab = {'students': 0, 'opened': 1, 'their': 2}

import torch

e = torch.nn.Embedding(3, 2)
```

number of words to embed

size of each embedding vector

```
e(torch.tensor(vocab['opened']))
>>> tensor([0.6399, 0.1779], grad_fn=<EmbeddingBackward>)

e(torch.tensor([0, 1, 2]))
>>> tensor([[ 0.4503, -0.1549],
>>>         [ 0.6399,  0.1779],
>>>         [-0.6537, -0.5875]], grad_fn=<EmbeddingBackward>)
```

Embedding layers as linear layers

one-hot vector for *opened*

embedding weights

embedding vector for *opened*

$$\begin{bmatrix} 0 & \mathbf{1} & 0 \end{bmatrix} \begin{bmatrix} 0.4503 & -0.1549 \\ \mathbf{0.6399} & \mathbf{0.1779} \\ -0.6537 & -0.5875 \end{bmatrix} = \begin{bmatrix} \mathbf{0.6399} & \mathbf{0.1779} \end{bmatrix}$$

$1 \times V$

size of the vocabulary

$V \times d$

embedding width

$1 \times d$

Embedding layers as linear layers

- An embedding layer can be understood as a linear layer that takes one-hot word vectors as inputs.

  embedding vectors = word-specific weights of the linear layer

- From a practical point of view, embedding layers are more efficiently implemented as lookup tables.

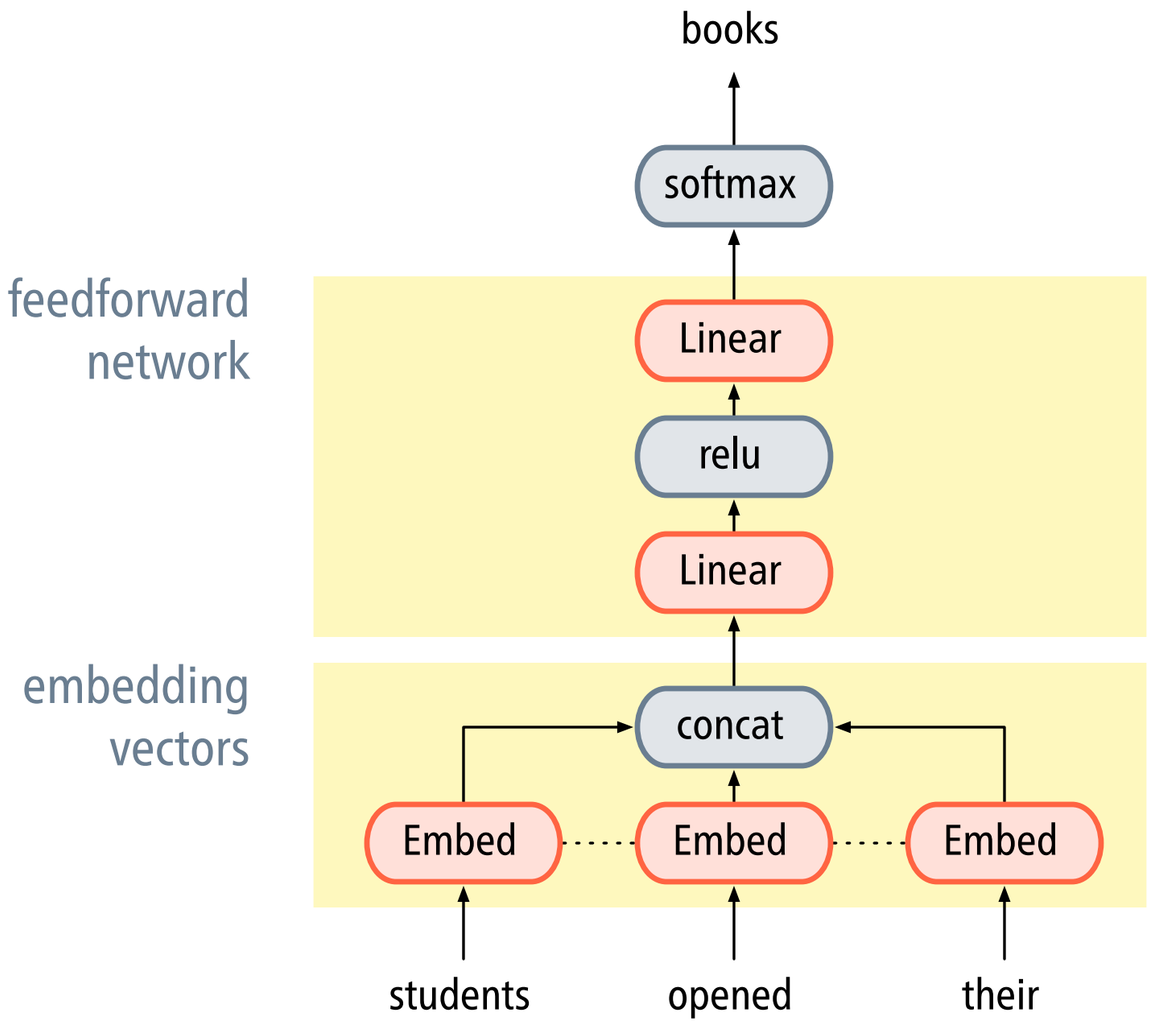
- Embedding layers are initialised with random values, and then updated through backpropagation, just like any other layer.

  default in PyTorch: $N(0, 1)$

Comparison of statistical and neural n-gram models

Property	Statistical model	Neural model
number of parameters	exponential in n	linear in n
parameter sparsity	mostly zeros	no zeros
learning of parameters	count-based MLE	gradient search

A neural four-gram model



Bengio et al. (2003)