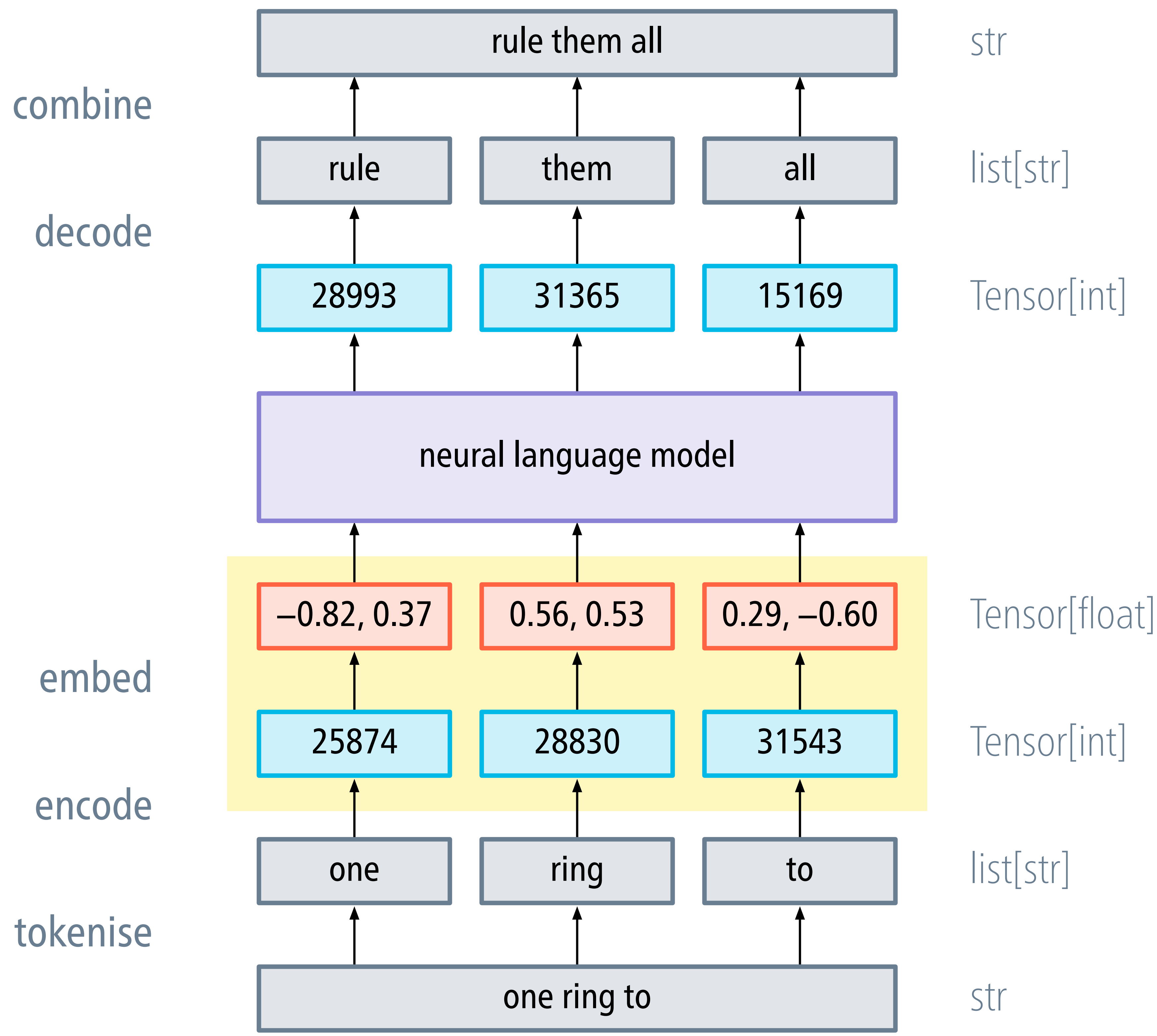
Natural Language Processing

Introduction to embeddings

Marco Kuhlmann
Department of Computer and Information Science

LINKÖPING UNIVERSITY

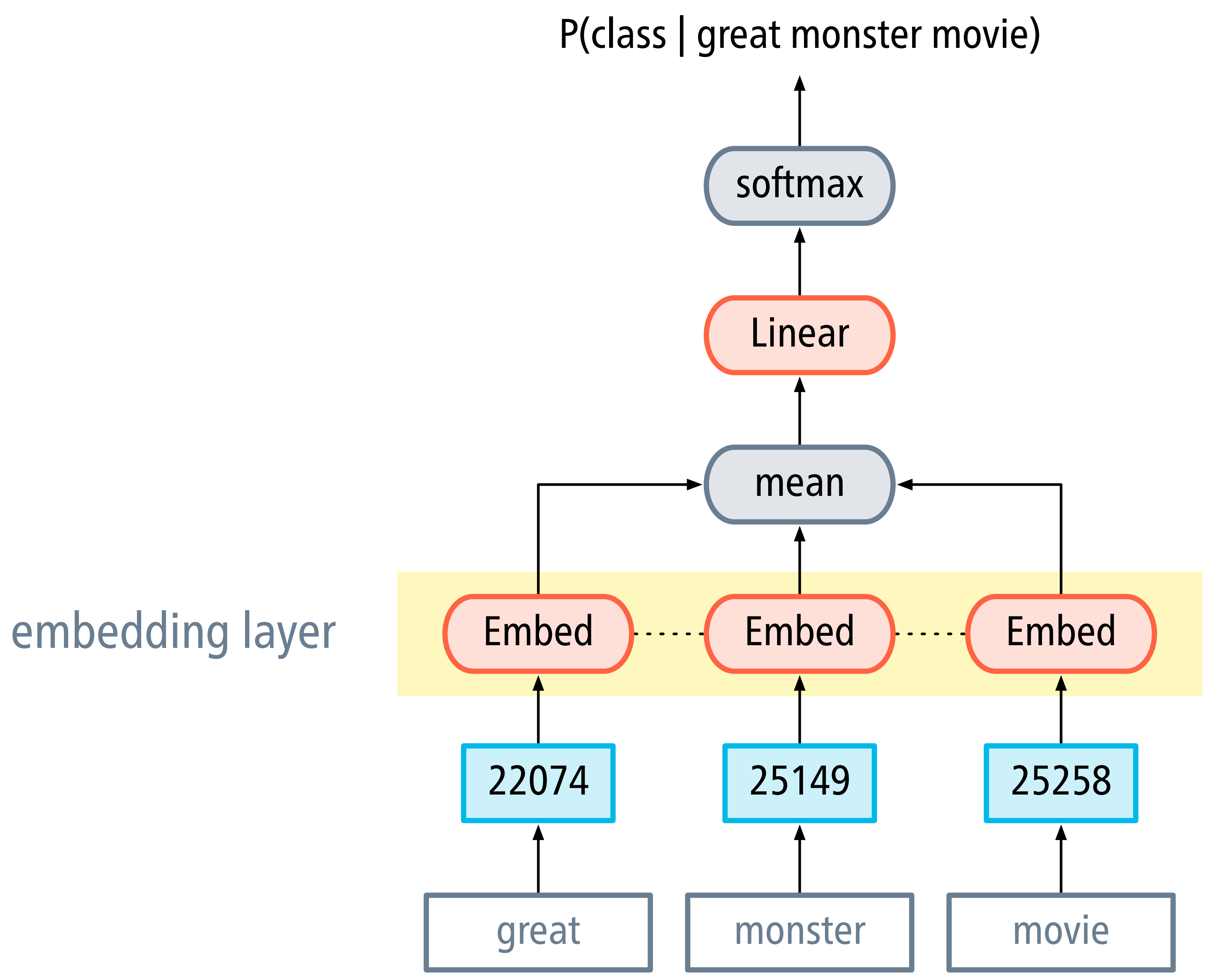
This work is licensed under a
Creative Commons Attribution 4.0 International License.


rule them all
str
combine
rule
them
all
list[str]
decode
28993
31365
15169
Tensor[int]
neural language model
–0.82, 0.37
0.56, 0.53
0.29, –0.60
Tensor[float]
embed
25874
28830
31543
Tensor[int]
encode
one
ring
to
list[str]
tokenise
one ring to
str

Embedding layers

- After each token has been encoded as an integer, it is sent through an **embedding layer**.
- The embedding layer assigns each token a fixed-size vector of floating-point numbers.
- Initially, these numbers are random, but we will tune them when training the embedding layer.

Bag-of-words classifier



Embedding layers in PyTorch

```
s2i = {'great': 0, 'monster': 1, 'movie': 2}

import torch
```

number of words to embed

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

size of each embedding vector

```
emb(torch.tensor(s2i['monster'], dtype=torch.long))
# tensor([0.6399, 0.1779], grad_fn=<EmbeddingBackward>)

emb(torch.tensor([s2i[s] for s in s2i], dtype=torch.long))
tensor([[ 0.4503, -0.1549],
        [ 0.6399,  0.1779],
        [-0.6537, -0.5875]], grad_fn=<EmbeddingBackward>)
```

Implementation of the bag-of-words classifier

```
class Classifier(nn.Module):

    def __init__(self, num_embeddings, embedding_dim, num_classes):
        super().__init__()
        self.embedding = nn.Embedding(num_embeddings, embedding_dim)
        self.linear = nn.Linear(embedding_dim, num_classes)

    def forward(self, x):
        # x is a tensor containing token IDs
        return self.linear(self.embedding(x).mean(dim=-2))
```

Embedding layers as linear layers

one-hot vector for *monster* | embedding weights | embedding vector for *monster*

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

$1 \times V$ | $V \times d$ | $1 \times d$

size of the vocabulary | embedding size

Trained embeddings are task-specific

- Initially, the embedding vectors are filled with random values.

  In PyTorch, these values come from the standard normal distribution.

- During training, backpropagation optimises the embedding vectors for the task at hand.

- After training, embedding vectors for which the network produces similar outputs will be similar to each other.

  as measured, for example, by cosine similarity

Embeddings for transfer learning

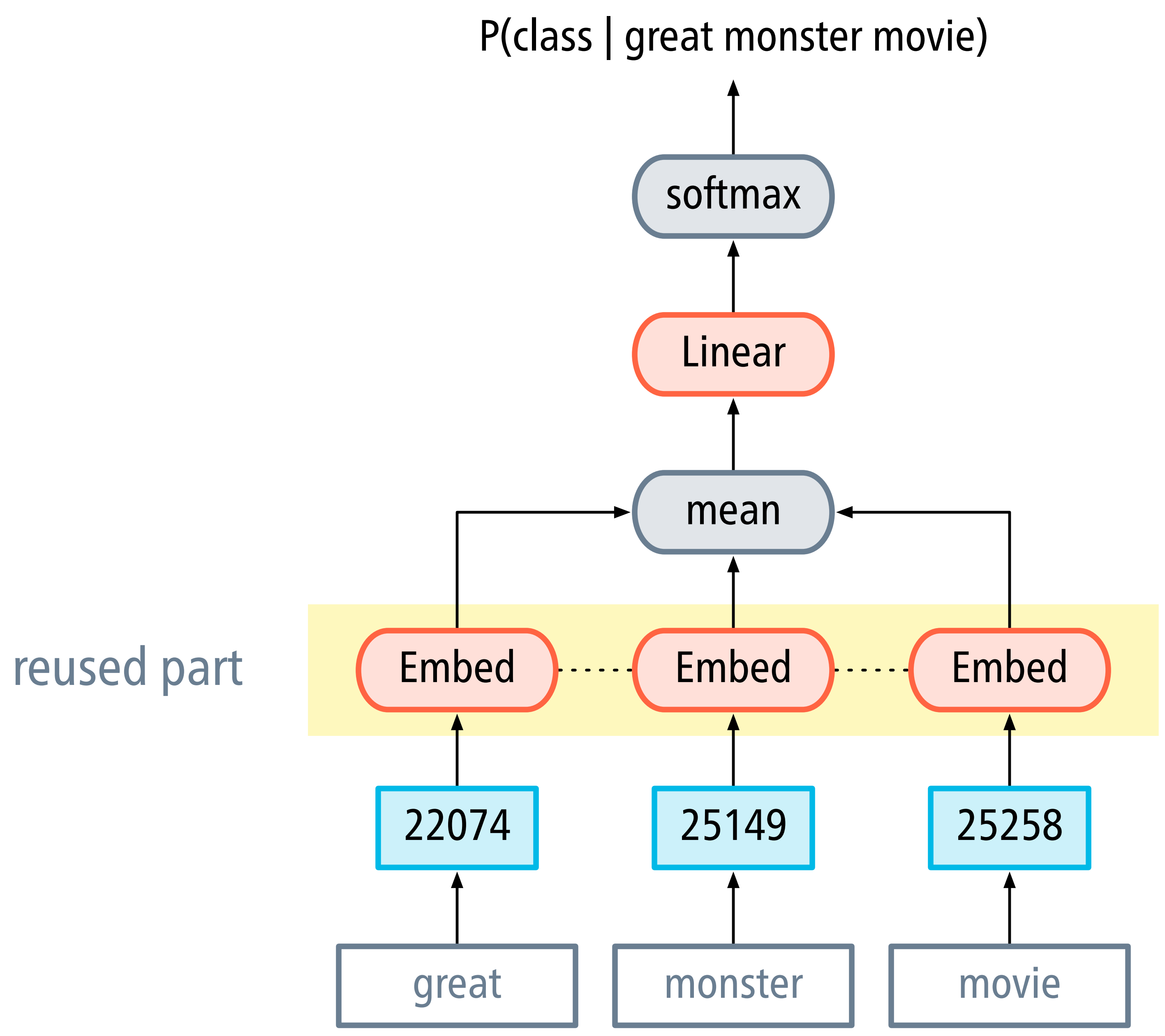
- **Transfer learning** aims to re-use knowledge gained while solving some previous task when solving the next task.

  speed up training, reduce the need for training data

- In the context of deep learning, transfer learning is typically implemented by re-using some part of a trained model.

- In particular, we could try re-using the embedding layers, instead of learning embeddings from scratch for each task.

Bag-of-words classifier



Re-using pre-trained embeddings

Pre-train embeddings on task *A* and use them to initialise the embedding layers of the network for task *B*. Then:

- **Alternative 1:** Train as usual, effectively fine-tuning the pre-trained embeddings to the task at hand.
- **Alternative 2:** Freeze the weights of the embedding layers, to prevent the pre-trained embeddings from being modified.

What pre-training tasks should we use?

- We want to learn representations that are generally useful, so we prefer pre-training tasks that are general.
- We need to find training data for the pre-training tasks, so we prefer tasks for which data is abundant.

  ideal candidate: raw text
- This makes language modelling an attractive pre-training task.