# Recurrent Neural Networks

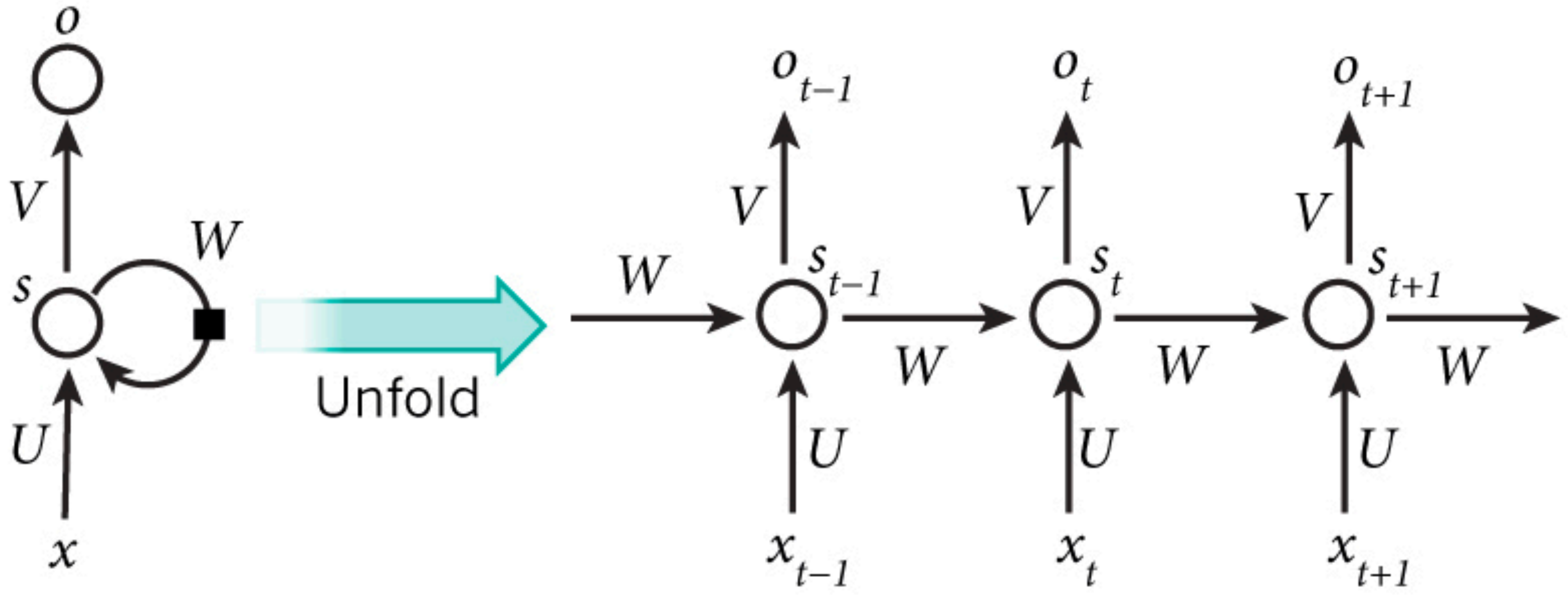
11-785 / 2019 Fall / Recitation 7

Kangrui, Hanna, Natnael

**"Drop your RNN and LSTM, they are no good!"**

The fall of RNN / LSTM, Eugenio Culurciello

*A recurrent neural network and the unfolding in time of the computation involved in its forward computation.*



## Recap: RNNs are hard to train

They suffer from :

- Saturation
- Vanishing/exploding gradients
- Complex loss surfaces with tons of bad local minima
- They don’t usually like dropout
- ...

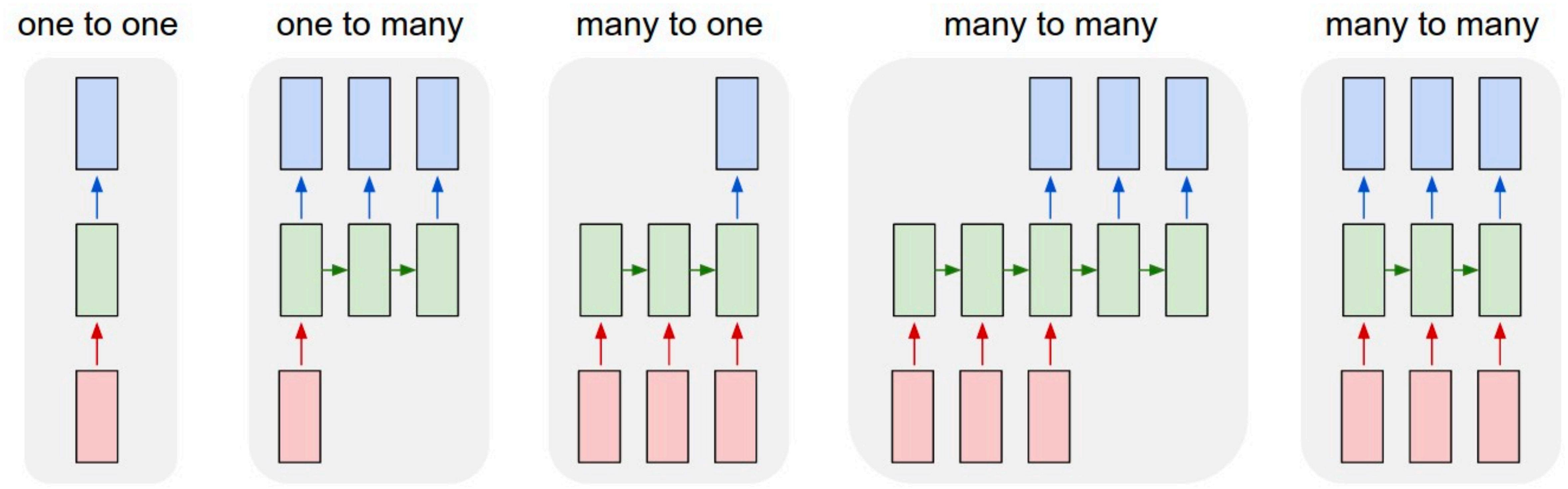
LSTMs/GRUs address some of these issues, but they’re not perfect.

When you use RNNs, you will spend most of your time tuning hyper-parameters (or looking for hacks in papers).

## Content

- 1 Language Model
- 2 RNNs in PyTorch
- 3 Train RNNs
- 4 Generation with an RNN
- 5 Variable length inputs

# Different Tasks



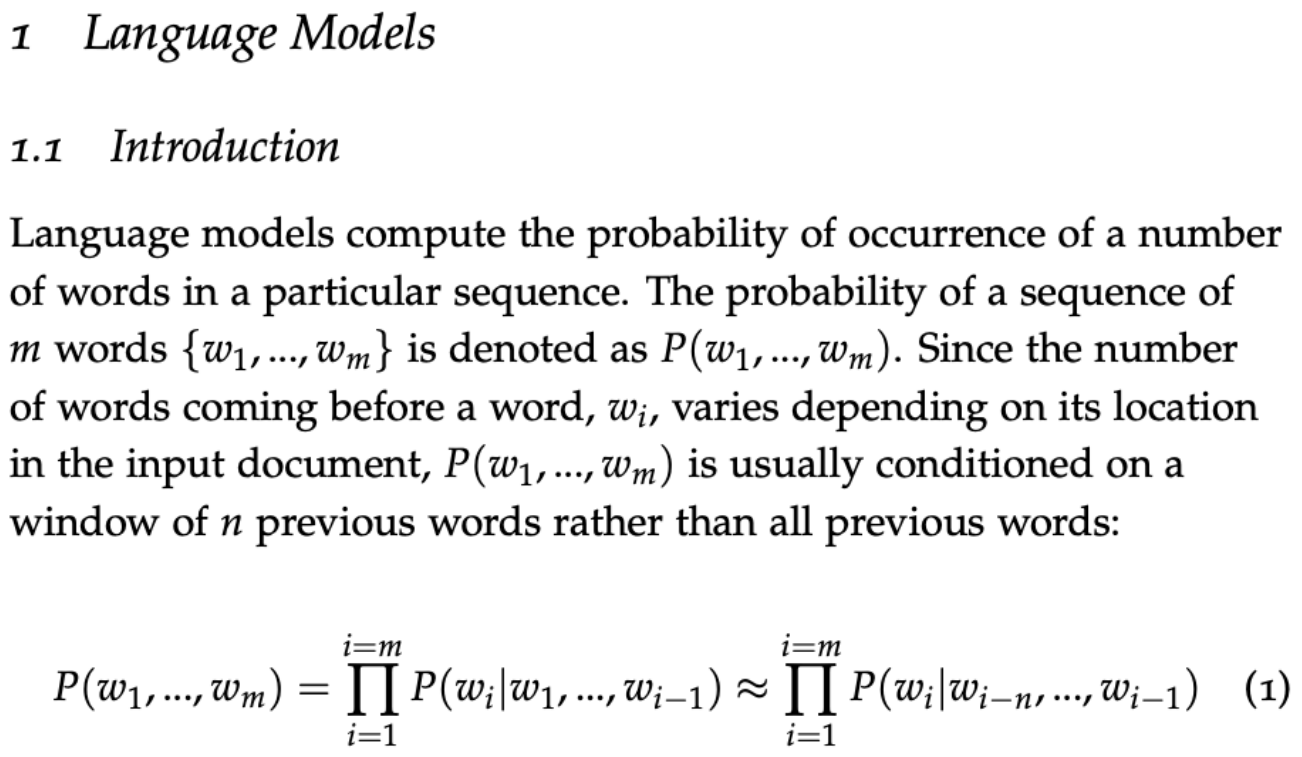
# Different Tasks

- Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right:
- **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification).
- **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words).
- **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment).
- **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French).
- **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

Credits to Andrej Karpathy

# 1 Language Models

- Goal: predict the “probability of a sentence” P(E)
- How likely it is to be an actual sentence



## 1 *Language Models*

### 1.1 *Introduction*

Language models compute the probability of occurrence of a number of words in a particular sequence. The probability of a sequence of $m$ words $\{w_1, ..., w_m\}$ is denoted as $P(w_1, ..., w_m)$. Since the number of words coming before a word, $w_i$, varies depending on its location in the input document, $P(w_1, ..., w_m)$ is usually conditioned on a window of $n$ previous words rather than all previous words:

$$P(w_1, ..., w_m) = \prod_{i=1}^{i=m} P(w_i | w_1, ..., w_{i-1}) \approx \prod_{i=1}^{i=m} P(w_i | w_{i-n}, ..., w_{i-1}) \quad (1)$$

Credits to cs224n

## 1 An RNN Language Model

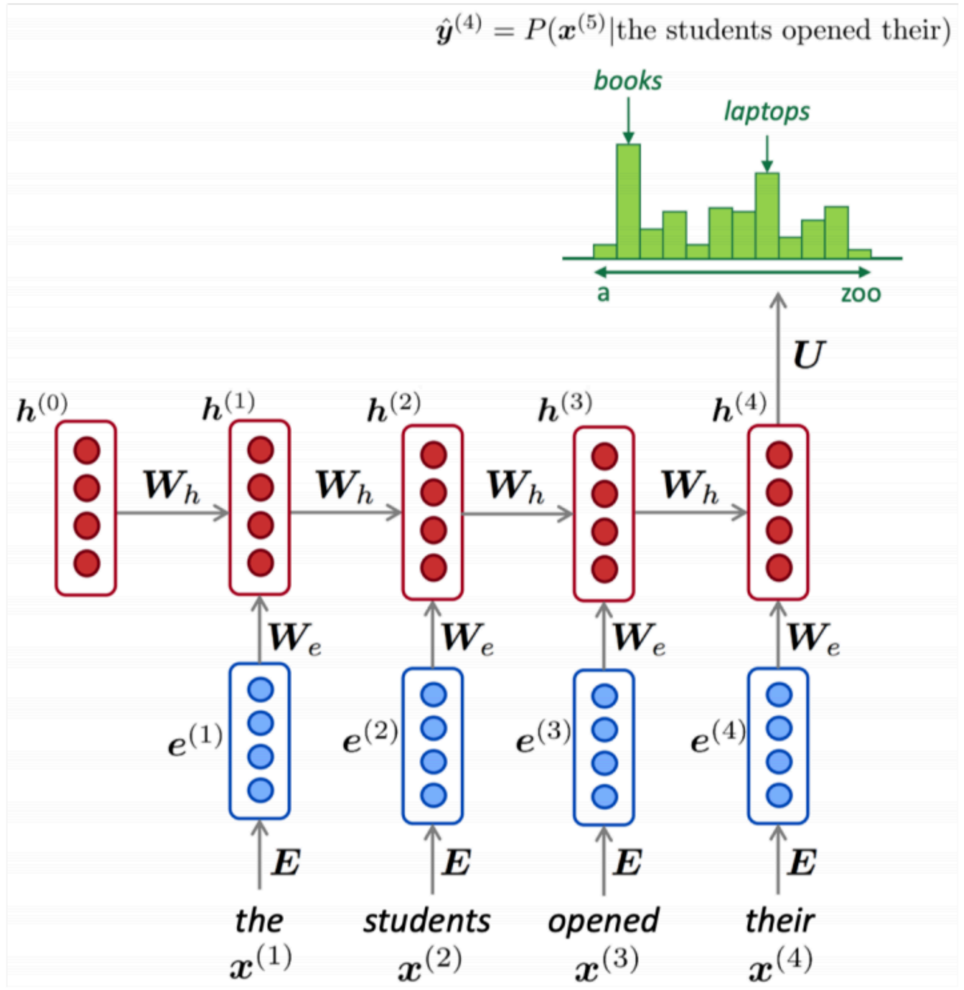
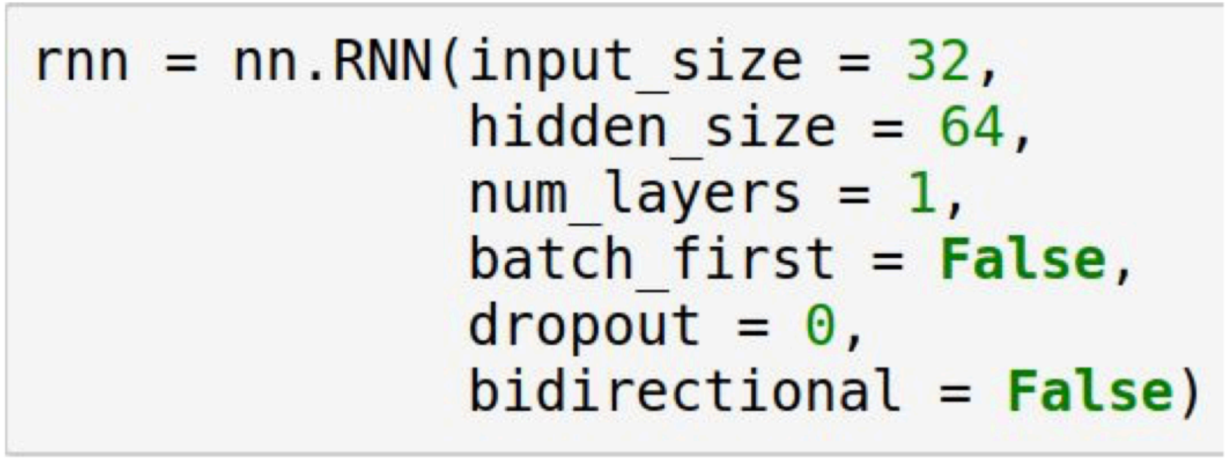


Figure 5: An RNN Language Model

# 2 RNN modules in Pytorch



```
rnn = nn.RNN(input_size = 32,
             hidden_size = 64,
             num_layers = 1,
             batch_first = False,
             dropout = 0,
             bidirectional = False)
```

- num_layers is the number of **stacked** (vertical) layers
- dropout is the dropout **between stacked layers**

The .forward() method takes an input of size
*seq_length* x *batch_size* x *input_size*
and an optional initial hidden state (defaults to 0) of size
*num_layers* x *batch_size* x *hidden_size*. It returns an output of
same size and the final hidden state.

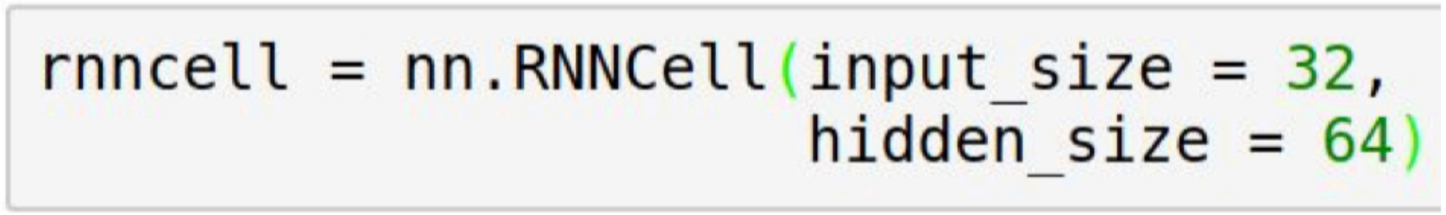
## 2 RNN modules in Pytorch

- Important: the outputs are exactly the hidden states of the final layer. Hence if the model returns y, h:
- y: (seq_len, batch, num_directions * hidden_size)
- h: (num_layers * num_directions, batch, hidden_size)
- Lets num_directions = 1, y[-1] == h[-1]
- But for LSTM and GRU:
- (h_0, c_0): (hidden_states, emory_cell)

## RNN cell

Sometimes you want to have more control between the stacked layers and the time steps (for example to access the intermediate hidden states).

For that you have the RNNCell module.



```
rnncell = nn.RNNCell(input_size = 32,
                     hidden_size = 64)
```

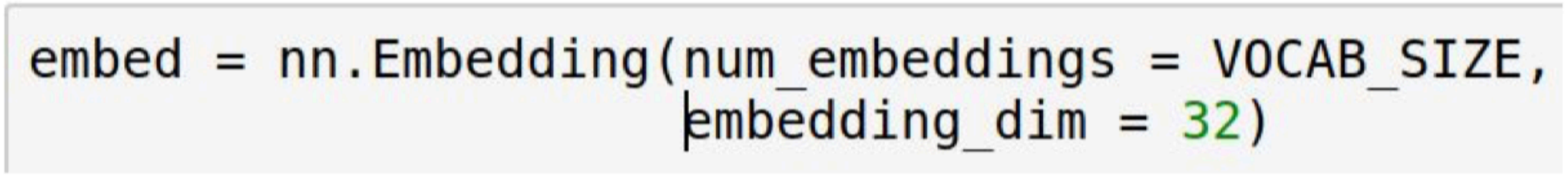
It takes an input of size *batch_size* x *input_size* and a hidden state, and returns the next hidden state.

## Embedding Layers

After the RNN module, you stack a linear layer of size

*hidden_size* x *vocabulary_size*

Before it, you need a *word projection* aka an embedding.



```
embed = nn.Embedding(num_embeddings = VOCAB_SIZE,
                     embedding_dim = 32)
```

Takes a LongTensor of arbitrary shape.

## Training a Language Model

Now you need batches to feed your model. Initially, you only have one big text.

The simplest way :

- Fix a sequence length L
- Concatenate all your words into one big (long) tensor of size N
- Divide it into N // L tensors of size L
- These are your elements.

Even if you train on a fixed size, the network should learn to generate text of arbitrary length.

## Evaluate your model

To evaluate how good your model is, you usually feed it with actual text from the (validation) set and look at :

- The loss per word : *l = loss/n_words*
- The **perplexity :** p = exp(l)

It quantifies how well your model predicts that sentence.

A perplexity of 100 (loosely) means that your model performed as if it had to choose uniformly and independently among 100 possibilities for each word

Let’s try all that out !

## Generation

To generate N words, you have N*vocabulary_size possible sequences. Recall that

$$P(E) = P(e_1, e_2, ..., e_M)$$
$$= \prod_{m=1}^{M} P(e_m | e_1, ..., e_{m-1})$$

To know each sentence’s probability you’d need to feed all (N-1)-length beginnings ⟶ (N-1) * vocabulary_size forward passes ! Unfeasible.

⟶ Need another way to get the most likely sequence, or at least a very likely one.

## Greedy Search, Random Search and Beam Search

1. Greedy search: select the most likely word
2. Random Search: sample a word from the distribution
3. Beam Search: keep the n best words at each step, n is the beam size

## Are we done?

- No.

## How to train a LM: fixed length

Now you need batches to feed your model. Initially, you only have one big text.

The simplest way :

- Fix a sequence length L
- Concatenate all your words into one big (long) tensor of size N
- Divide it into N // L tensors of size L
- These are your elements.

Even if you train on a fixed size, the network should learn to generate text of arbitrary length.

## Limits of fixed-length inputs

In Machine Translation, Speech recognition,etc. you have pairs of sequences.

ex : I like apples → J’aime les pommes

You need to keep these sequences as is to learn something.

We’re not dealing with any of these specific applications today but to learn RNNs you need to learn how to deal with **variable length inputs**.

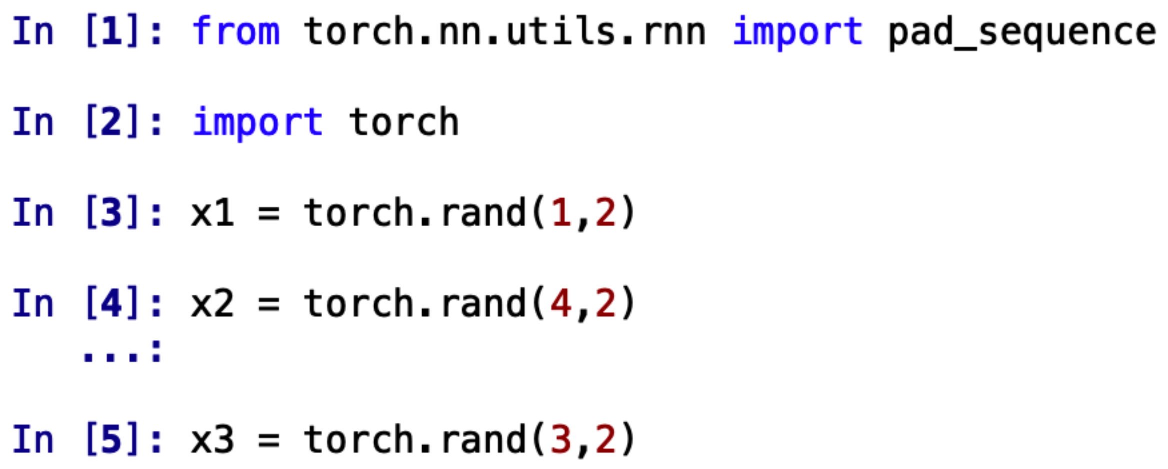
**You will need to do this in the upcoming HWs. Pay attention.**

# How to train a LM: Variable Length

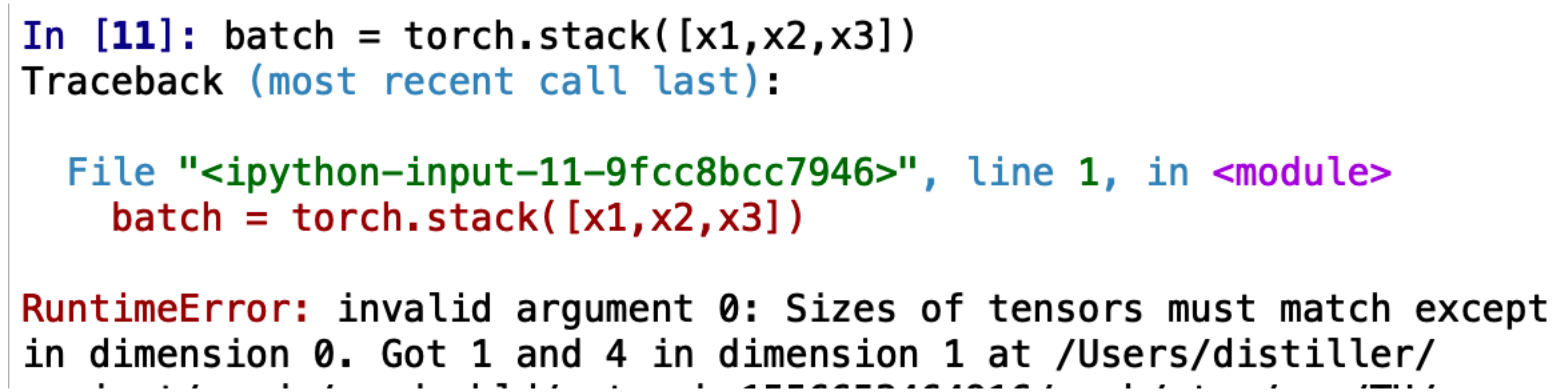
- Your dataset is now a list of N sequences of different lengths
- The input has a fixed dimension (seq_len, batch, input_size)
- How could we deal with this situation?

1. pad_sequence
2. Packed sequence

# 1 pad_sequence

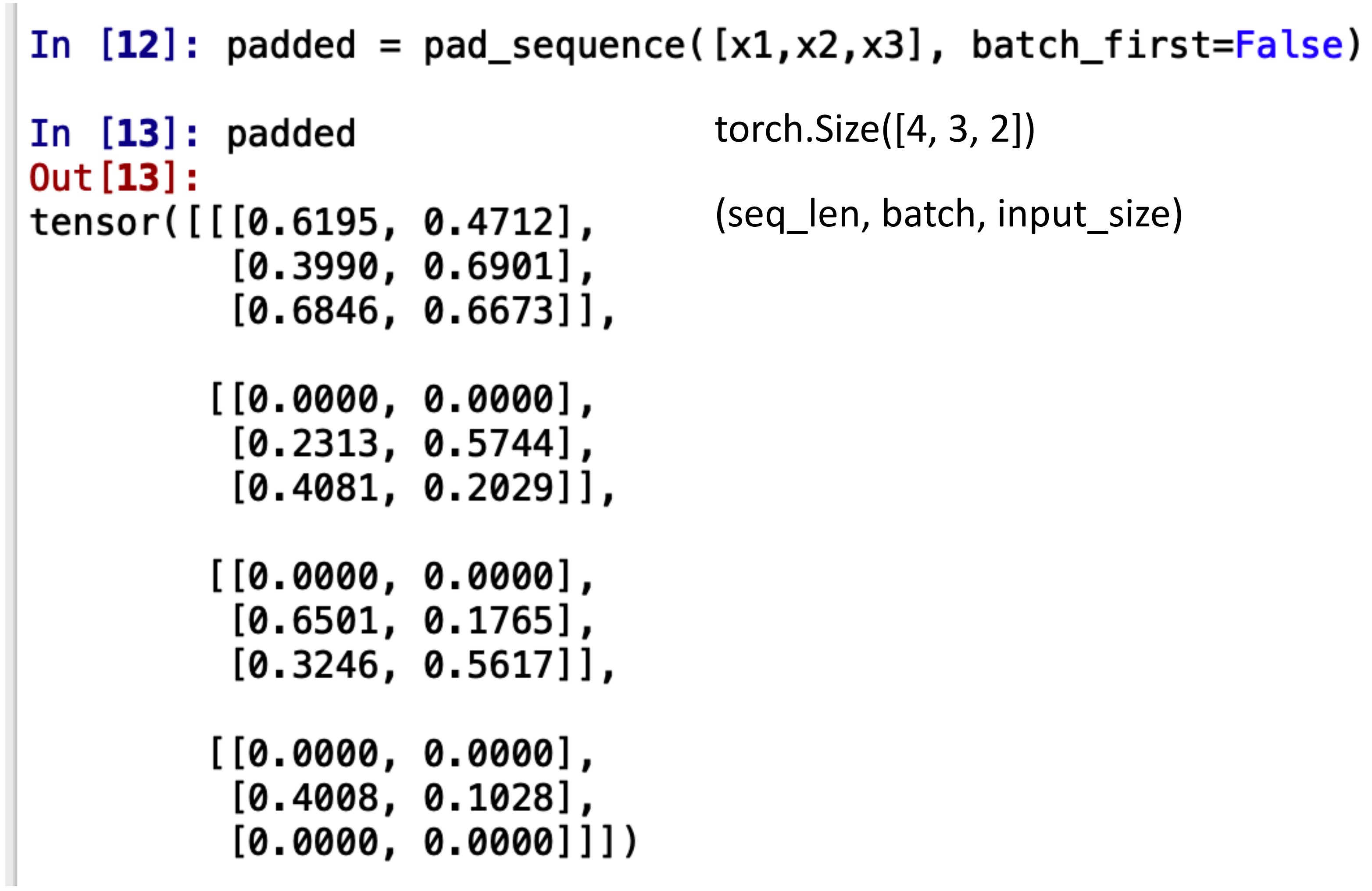


```
In [1]: from torch.nn.utils.rnn import pad_sequence

In [2]: import torch

In [3]: x1 = torch.rand(1,2)

In [4]: x2 = torch.rand(4,2)
   ...:

In [5]: x3 = torch.rand(3,2)
```



```
In [11]: batch = torch.stack([x1,x2,x3])
Traceback (most recent call last):

  File "<ipython-input-11-9fcc8bcc7946>", line 1, in <module>
    batch = torch.stack([x1,x2,x3])

RuntimeError: invalid argument 0: Sizes of tensors must match except
in dimension 0. Got 1 and 4 in dimension 1 at /Users/distiller/
```

# 1 pad_sequence



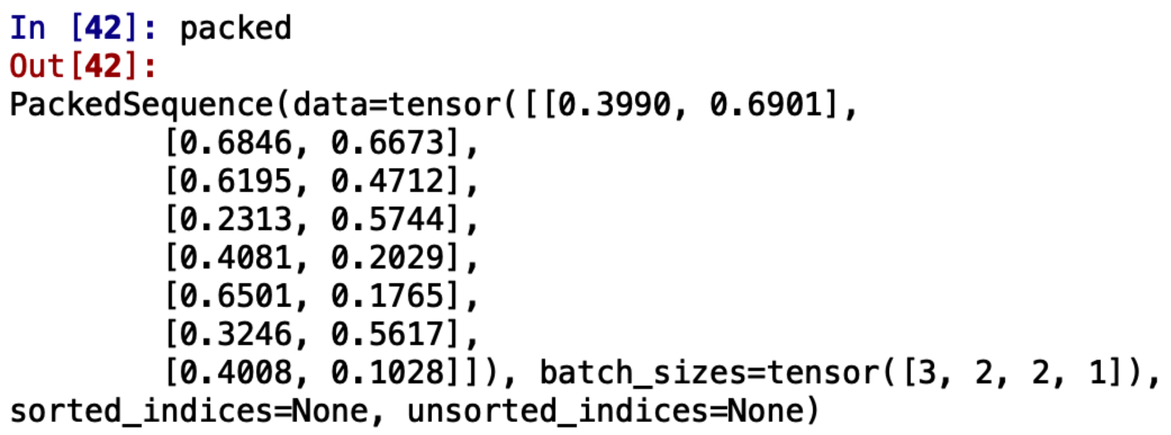
```
In [12]: padded = pad_sequence([x1,x2,x3], batch_first=False)

In [13]: padded
Out[13]:
tensor([[[0.6195, 0.4712],
         [0.3990, 0.6901],
         [0.6846, 0.6673]],

        [[0.0000, 0.0000],
         [0.2313, 0.5744],
         [0.4081, 0.2029]],

        [[0.0000, 0.0000],
         [0.6501, 0.1765],
         [0.3246, 0.5617]],

        [[0.0000, 0.0000],
         [0.4008, 0.1028],
         [0.0000, 0.0000]]])
```

torch.Size([4, 3, 2])

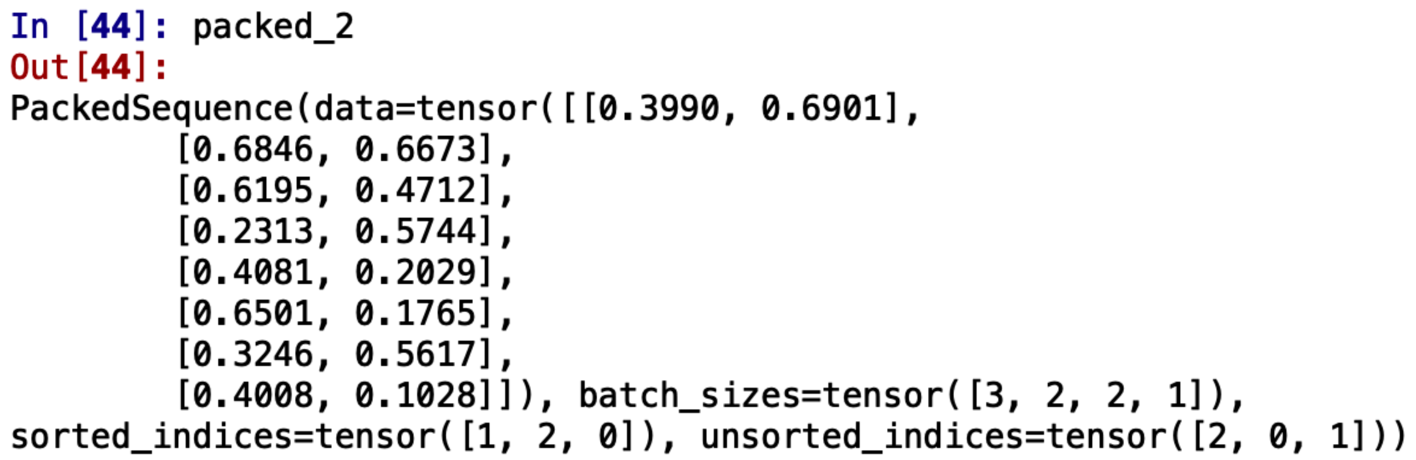
(seq_len, batch, input_size)

# 2 pack_sequence

packed_2 = pack_sequence([x1,x2,x3], enforce_sorted=True)
packed_2 = pack_sequence([x1,x2,x3], enforce_sorted=False)



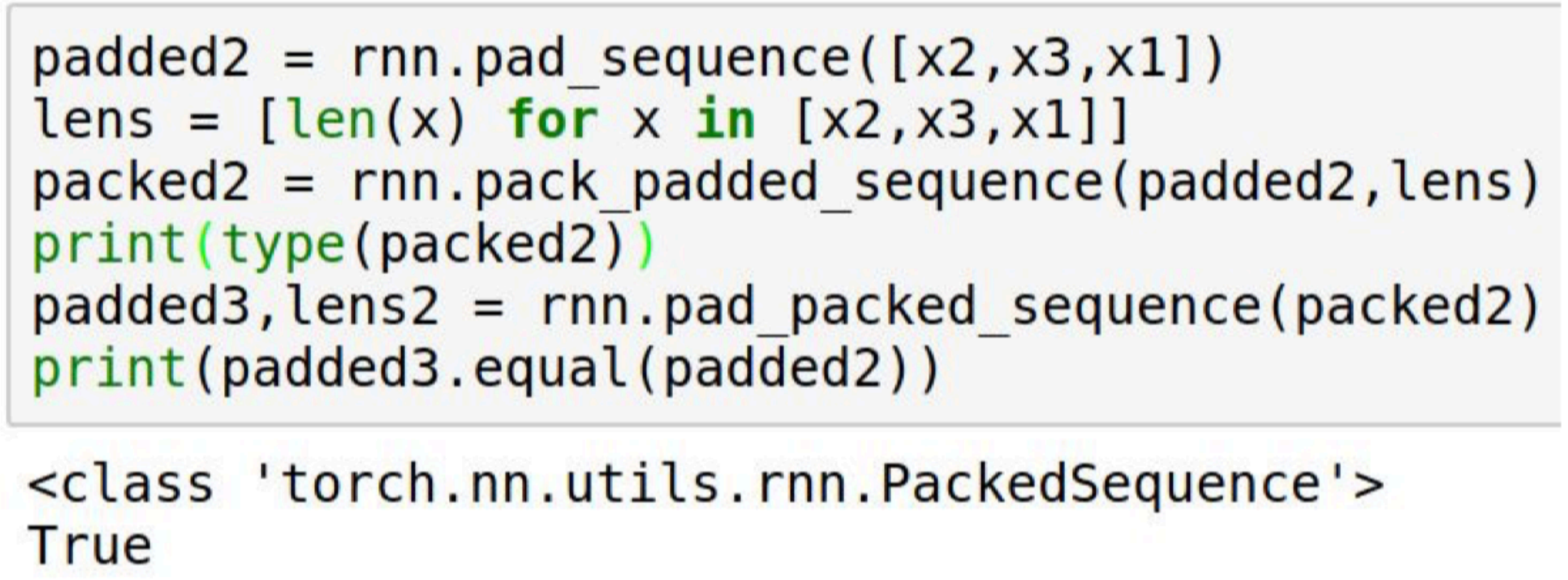
```
In [42]: packed
Out[42]:
PackedSequence(data=tensor([[0.3990, 0.6901],
        [0.6846, 0.6673],
        [0.6195, 0.4712],
        [0.2313, 0.5744],
        [0.4081, 0.2029],
        [0.6501, 0.1765],
        [0.3246, 0.5617],
        [0.4008, 0.1028]]), batch_sizes=tensor([3, 2, 2, 1]),
sorted_indices=None, unsorted_indices=None)
```



```
In [44]: packed_2
Out[44]:
PackedSequence(data=tensor([[0.3990, 0.6901],
        [0.6846, 0.6673],
        [0.6195, 0.4712],
        [0.2313, 0.5744],
        [0.4081, 0.2029],
        [0.6501, 0.1765],
        [0.3246, 0.5617],
        [0.4008, 0.1028]]), batch_sizes=tensor([3, 2, 2, 1]),
sorted_indices=tensor([1, 2, 0]), unsorted_indices=tensor([2, 0, 1]))
```

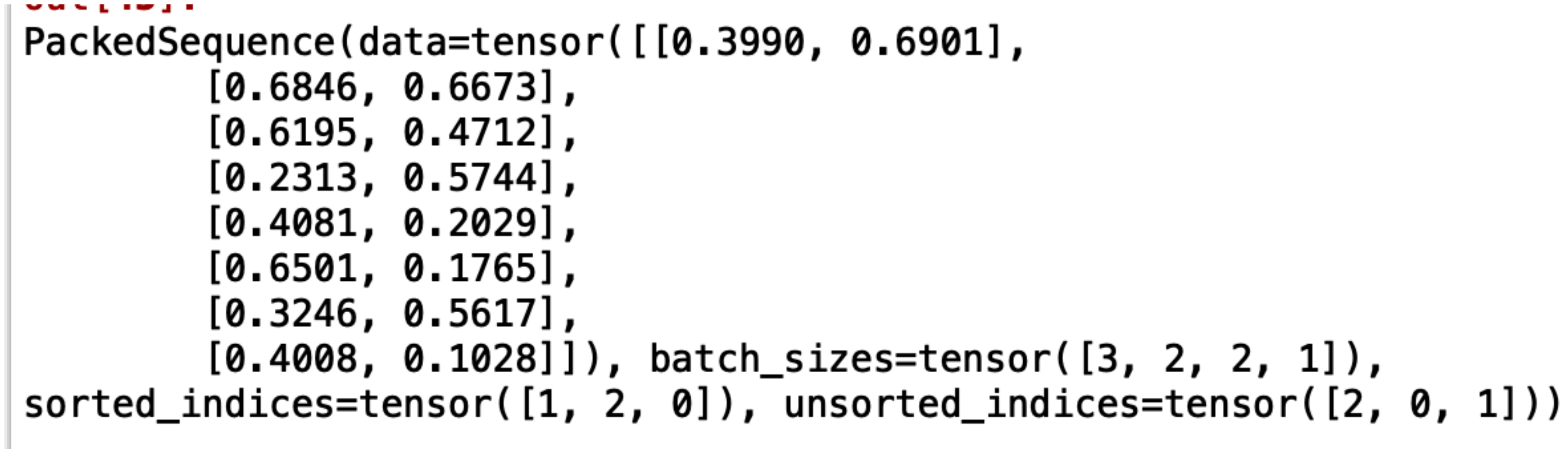
# 3 pack_padded_sequence and pad_packed_sequence

You can go from padded to packed and packed to padded, but need to track the lengths



```
padded2 = rnn.pad_sequence([x2,x3,x1])
lens = [len(x) for x in [x2,x3,x1]]
packed2 = rnn.pack_padded_sequence(padded2,lens)
print(type(packed2))
padded3,lens2 = rnn.pad_packed_sequence(packed2)
print(padded3.equal(padded2))
```

```
<class 'torch.nn.utils.rnn.PackedSequence'>
True
```



```
PackedSequence(data=tensor([[0.3990, 0.6901],
        [0.6846, 0.6673],
        [0.6195, 0.4712],
        [0.2313, 0.5744],
        [0.4081, 0.2029],
        [0.6501, 0.1765],
        [0.3246, 0.5617],
        [0.4008, 0.1028]]), batch_sizes=tensor([3, 2, 2, 1]),
sorted_indices=tensor([1, 2, 0]), unsorted_indices=tensor([2, 0, 1]))
```

Why is the batch_size = tensor([3, 2, 2, 1]) here?

batch_sizes (Tensor): Tensor of integers holding information about the batch size at each sequence step

For instance, given data ``abc`` and ``x`` the :class:`PackedSequence` would contain data ``axbc`` with ``batch_sizes=[2,1,1]``.

## Packed Sequences and RNNs

- Packed sequences are on the same device as the padded sequence
- Packed sequences could help your RNNs know the length for each instance

## MLP, RNN, CNN, Transformer

- All these layers are just features extractors
- Temporal convolutional network (TCN) "outperform canonical recurrent networks such as LSTMs across a diverse range of tasks and datasets, while demonstrating longer effective memory"

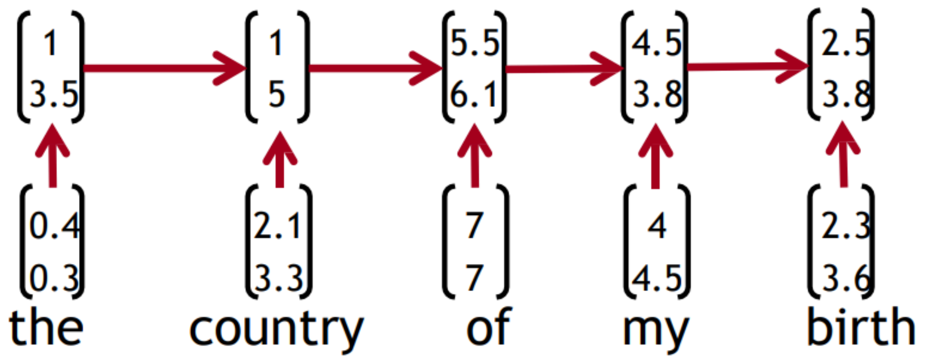
(An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling)

ai_antihype
@ai_antihype
关注

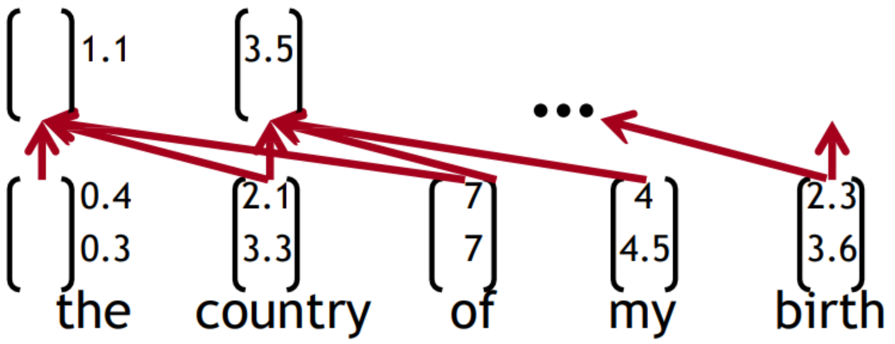
We find it extremely unfair that Schmidhuber did not get the Turing award. That is why we dedicate this song to Juergen to cheer him up.

## Comparison between RNNs and CNNs

- RNN processes input sequentially



- CNN can compute vectors for every possible phrase
  - Example: “the country of my birth”
    - “the country", "country of", "of my", "my birth", "the country of", "country of my", "of my birth", …



# Is that enough?

- RNN, LSTM, GRU
- Transformer (would be covered more in the future lectures and recitations)
- CNN

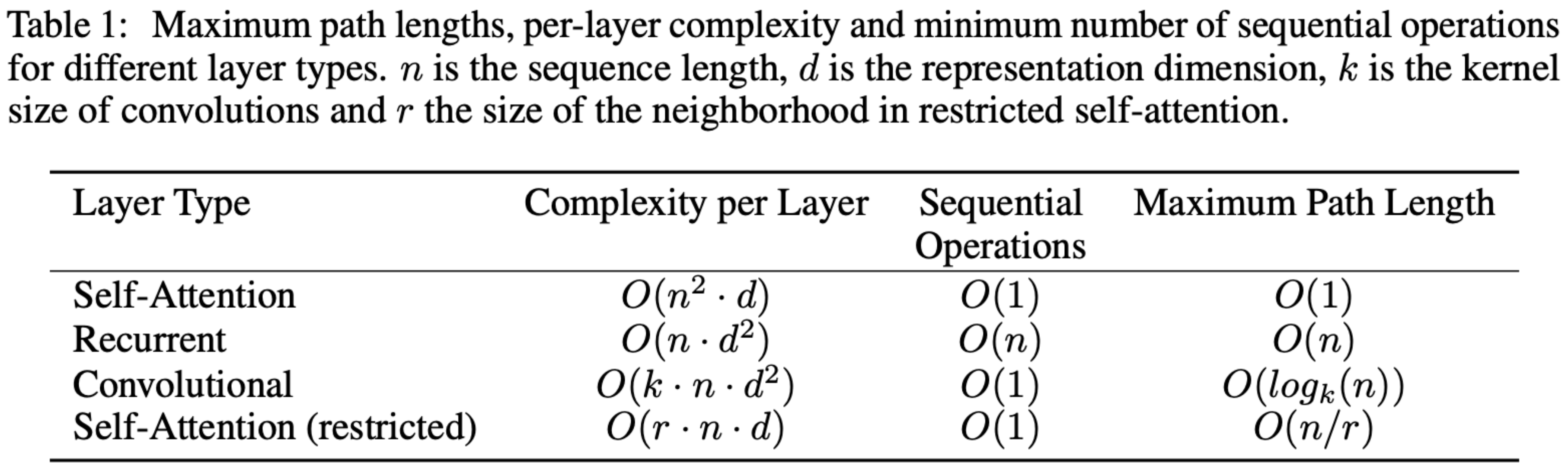


Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. $n$ is the sequence length, $d$ is the representation dimension, $k$ is the kernel size of convolutions and $r$ the size of the neighborhood in restricted self-attention.

| Layer Type | Complexity per Layer | Sequential Operations | Maximum Path Length |
|---|---|---|---|
| Self-Attention | $O(n^2 \cdot d)$ | $O(1)$ | $O(1)$ |
| Recurrent | $O(n \cdot d^2)$ | $O(n)$ | $O(n)$ |
| Convolutional | $O(k \cdot n \cdot d^2)$ | $O(1)$ | $O(log_k(n))$ |
| Self-Attention (restricted) | $O(r \cdot n \cdot d)$ | $O(1)$ | $O(n/r)$ |

Credits: Attention is all you need

## Papers

- https://arxiv.org/pdf/1708.02182.pdf
- For more tricks about Regularizing and Optimizing LSTM Language Models
- Attention is all you need!
- https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0
- LSTM song!!!