

# Intro to deep learning

Dr. Janoś Gabler, University of Bonn

## Lecture 10: Recurrent Neural Networks



# Motivation

Today you will see how a bunch of matrix multiplication can generate text and translate from English to German

# Topics

- Recap: Tokenization and embeddings
- `dataclasses` to store Parameters
- Language modelling with simple Recurent Neural Networks
- Machine translation with the encoder-decoder RNN
- Problems of RNNs

# Recap: Tokenization and embeddings

- Tokenization: Convert text to integers
  - Character level
  - (Sub-)word level
- Input embeddings: Convert tokens to one-hot vectors
- Example: `"hello"`
  - Vocabulary: ["e", "h", "l", "o"]
  - Vocabulary size: 4
  - Tokens: [1, 0, 2, 2, 3]
  - Embedding of "h": `array([0, 1, 0, 0])`

# Task 1

# Storing parameters

- In our first network
  - List of three weight matrices
  - List of three bias vectors
- Today:
  - 5 different matrices
- Need to bundle them to reduce complexity

# Option 1: Dicts

```
p = {  
    "w": np.array(...),  
    "b": np.array(...),  
}  
  
relu(p["w"] @ x + p["b"])
```

- Pros:
  - Name based access
  - We already know dicts
  - Very lightweight
- Cons
  - `p["w"]` is not pretty
  - No autocomplete
  - Easy to have typos

# Option 2: Class

```
class Params:
    def __init__(self, w, b):
        self.w = w
        self.b = b
```

```
p = Params(w, b)
p
```

```
<__main__.Params at 0x7f66f8120350>
```

- Pros:
  - Autocomplete and typo corrections
  - `p.w`` access works
- Cons
  - Redundant typing
  - No nice string representation



# Best of both worlds: dataclass

```
from dataclasses import dataclass
```

```
@dataclass
```

```
class Params:
```

```
    w: np.ndarray
```

```
    b: np.ndarray
```

```
w = np.ones(3)
```

```
b = np.arange(3)
```

```
p = Params(w, b)
```

```
p
```

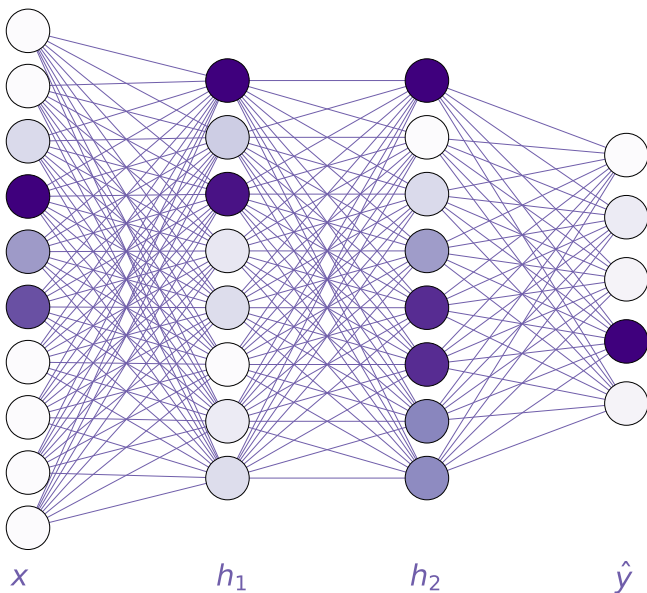
```
Params(w=array([1., 1., 1.]), b=array([0, 1, 2]))
```

- Less typing
- Type hints are self documenting
- Better string representation
- Built into Python
- More information: [Video by Raymond Hettinger](#)

# Language modelling

- Language model is a precisely defined type of model
- Task: Given a sequence of text, predict the next word
- More precisely: predict probability distribution over next word
- Input sequences of different lengths
- Most famous example: GPT

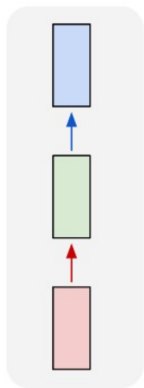
# Why our simple model can't do this



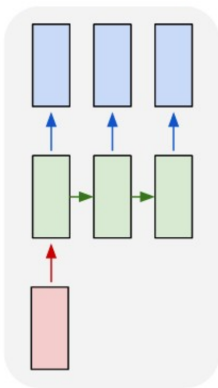
- Fixed size of model inputs
- No memory between inputs

# Model interfaces

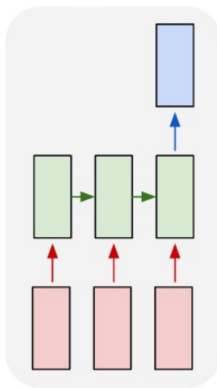
one to one



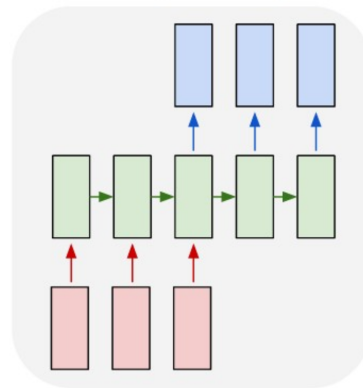
one to many



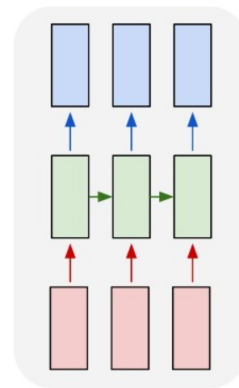
many to one



many to many



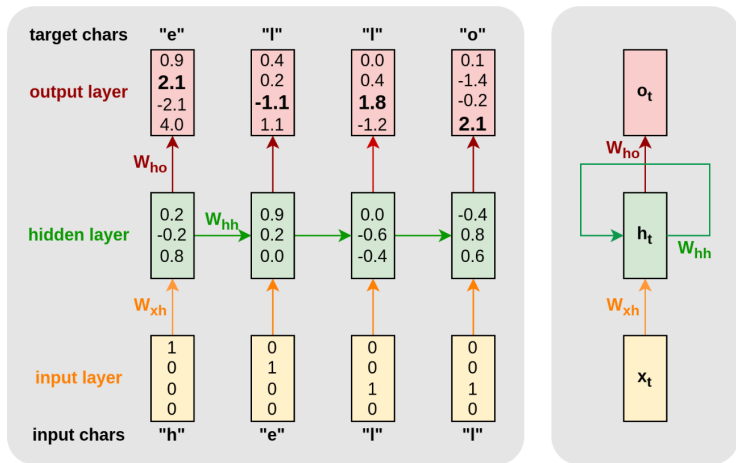
many to many



1. Image classification
2. Image captioning
3. Text classification

4. Machine translation
5. Language modelling

# The simple RNN on one slide



Source: Tobias Stenzel's Blog

- 3 Different weight matrices
- Weights stay the same between time steps
- Left: unrolled representation
- Right: recurrent representation
- $h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$
- $y_t = W_{hy} \cdot h_t$

# Task 2

# A few more details

- The model step:
  - $h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$
  - $y_t = W_{hy} \cdot h_t$
- This step is called once for each input embedding
- The initial hidden state is a vector of zeros

# Task 3



# The loss function

- Goal: Model should be good at predicting likely next words
- Same likelihood/cross-entropy approach as before
- A sequence of 5 tokens contains four training examples

```
"h", "e", "l", "l"  
"e", "l", "l", "o"
```

# Goal for the training in our example

- We just want the model to memorize the word hello
- See it as proof of concept
  - Information can be encoded in parameter matrices
  - RNN is a suitable model structure
- No holdout sample
- No worry of overfitting

# Large models

- Language models are trained on "a good chunk of the internet"
- Typically, just one epoch to avoid overfitting

# Task 4

# Writing the sequence-to-sequence function

- The model is not yet text-to-text
- Need a function that looks as follows:

```
def s2s_model(text, p, vocabulary):  
    # tokenize the text  
    # call the model  
    # translate the output embeddings into text
```

# Task 5

# Language models for question answering

- Reminder: Language model = model that predicts a likely next word
- Can be used very creatively
- For example, what is the likely continuation of this sequence?

$10 + 5 = 15$

$3 + 41 = 44$

$11 + 7 =$

- Similar approaches convert language models to chat bots

# How to go deeper

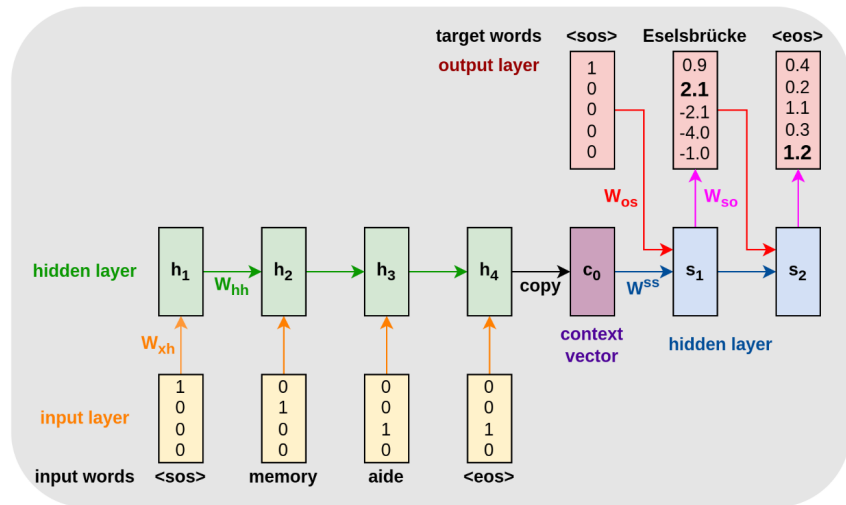
- Our RNN:
  - took input embeddings as input
  - returned output embeddings
- We could stack another network on top that
  - takes the output embeddings of the first network
  - produces other output embeddings
- Each network is called an RNN cell
- Parameters are not shared across RNN cells!
- This is just what happens in deep recurrent neural networks



# Machine translation

- Language models can also be used for translation
- However, they are not the most efficient at that
  - input and target sentence might have different lengths
  - order of words might be different
- Encoder-Decoder Architecture solves exactly that

# The encoder-decoder architecture



Source: Tobias Stenzel's Blog

- Encode step is as before but not producing  $\hat{y}$ 's
- Final hidden state becomes initial decoder state
- Decode step is similar to before:
  - Takes embedding of previous  $y$  as input
  - Maintains an internal state
  - Produces new  $\hat{y}$ 's from previous  $\hat{y}$  and internal state

# Properties

- Output sequence can now have different length than input sequence
- First produced token can take the entire input into account
- Parameters are not shared between encoder and decoder
  - In total 5 parameter matrices
- There is an input and output vocabulary
- We need a start and end token

# New example

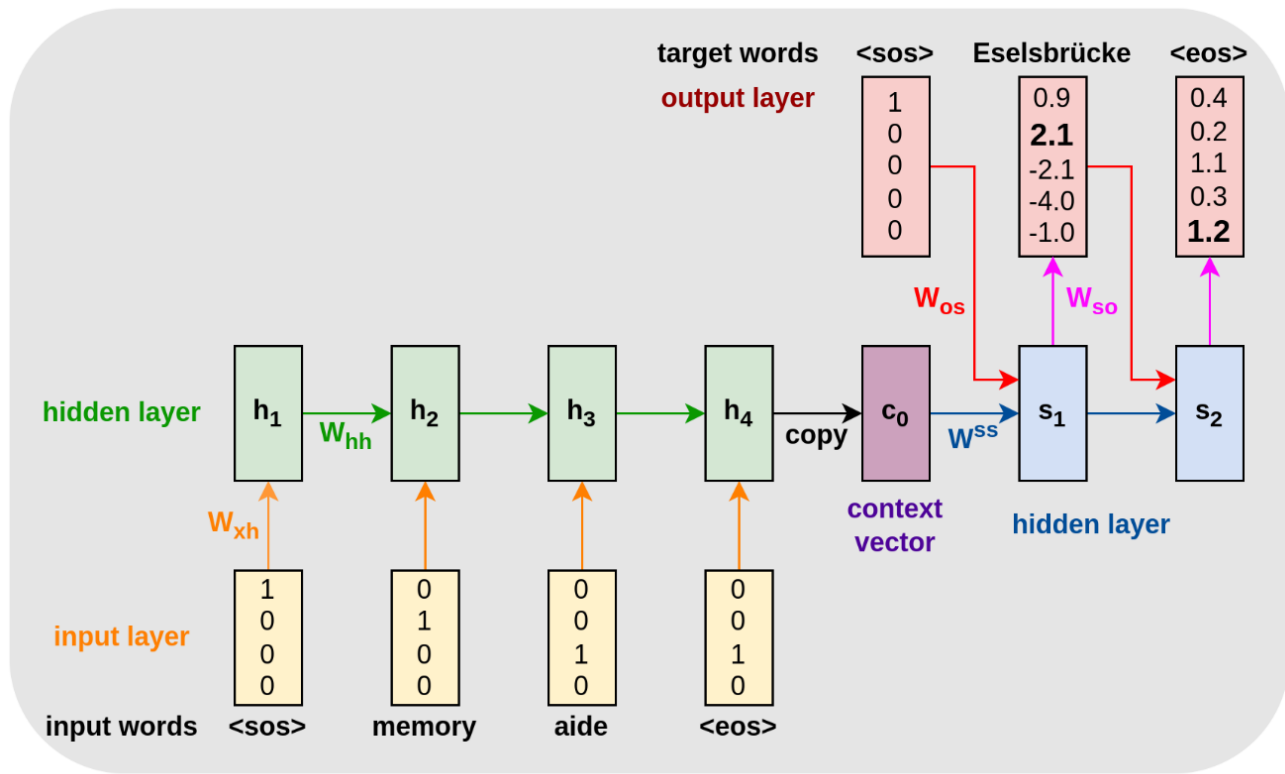
- Translate "hello world" into "hallo welt"
- Use word level embedding
- With `<SOS>` and `<EOS>` vocabulary size is 4
- Implement tiniest possible network that can memorize the correct translation!

# Encoder and decoder steps

- Encoder step
  - $h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$
  - Same as before but no equation for ``y``
- Copy step:
  - $s_0 = h_T$
- Decoder step
  - $s_t = \tanh(W_{ss} \cdot s_{t-1} + W_{ys} \cdot y_{t-1})$
  - $y_t = W_{sy} \cdot s_t$
  - $y_{t-1}$  plays role of  $x_t$ ; Rest as before

# Task 6

# Unrolled representation



Source: Tobias Stenzel's Blog

# Task 7



# Differences for text-to-text model

- Previously: One vocabulary
- Now: Input and output vocabulary
- Be careful to use the right vocabulary in each step

# Task 8

# Summary

- Feed-forward neural networks need fixed input and output size and have no memory
- RNNs are the simplest networks that can have variable size inputs and/or outputs
- Our models were tiny and could only memorize one task!
- For more useful networks you have to make them larger and deeper
- Next week we will see more complex architectures for the same cases

# Problem of RNNs

- RNNs have two problems:
  1. They cannot model relationships between words that are far apart
  2. They are hard to train
- Both come inherently from the recurrent structure where the "memory" vector  $h$  gets multiplied over and over again by the same matrix  $W_{hh}$
- There are ways to mitigate that (LSTM-RNNs)
- They were state of the art until 2017 but are now replaced by transformers
- We will skip LSTMs and move directly to transformers