Recurrent Neural Networks (RNNs)

Nikolai Hartmann

LMU Munich

February 8, 2023, ErUM-Data-Hub Train-the-Trainer Workshop

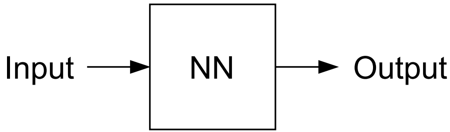
Outline at the "Basic Concepts" school last August:

- 9:00-10:30 Lecture
  - General introduction
  - The LSTM (long short term memory)
  - Model building with RNNs in keras
  - Hands-on: Understand RNN implementation
  - Advanced concepts
- 11:00-12:30 Hands-on
  - Predict a sin curve
  - Detect a cosmic ray signal in noisy radio wave data
- 14:30-16:30 Hands-on
  - Continue with exercises
  - Additional exercise on variable-length sequences
    $\rightarrow$ + train an RNN classifier on the TopTagging dataset

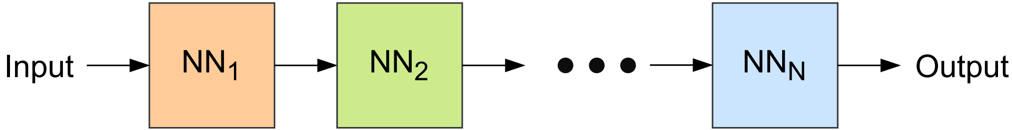
Today: Lecture + presentation of exercises

Non-recurrent neural networks

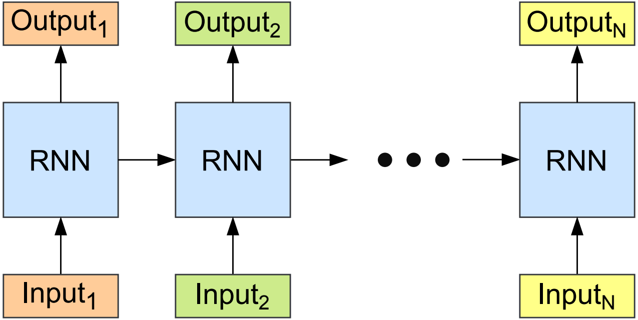
Blackbox view: Fixed-size input, fixed-size output:



Typically implemented as a stack of layers:

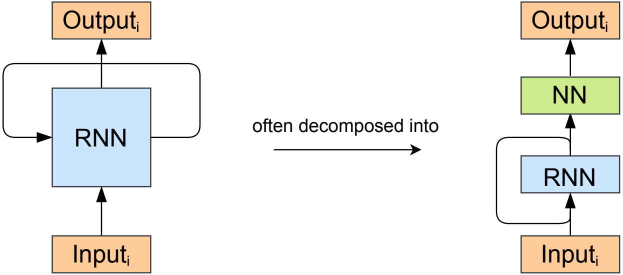


Recurrent neural networks



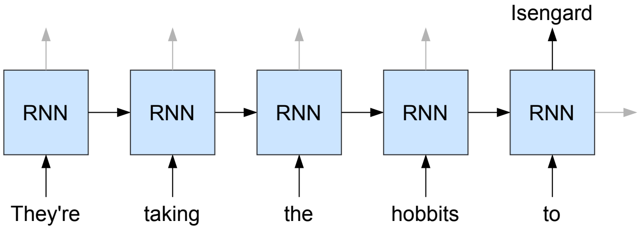
- Operate on a sequence, passing-on a hidden **state**
- **Shared weights** across the sequence
- Usually thought of as a sequence *in-time*, but can be any **ordered sequence**
- Usually trained with Backpropagation through time (BPTT)
  (nothing special when using modern ML libraries)

That's what's meant by this diagram



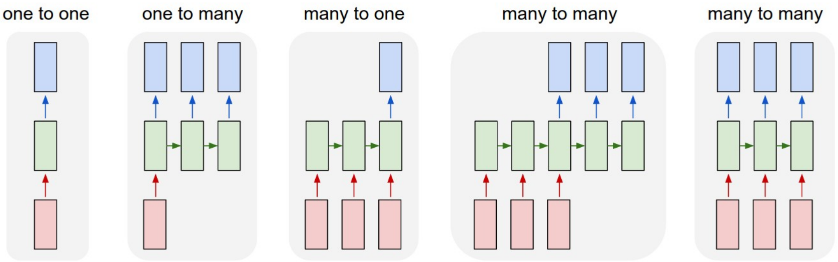
RNN block has a **feedback** connection
$\rightarrow$ (part of) output fed back as input to the next element of the sequence

Example: predict the next word



- Don't need to use output at every step
  → here: feed in a several words
  → use prediction after a few steps
- Don't need to feed input at every step
  → here we do, but could also feed back in prediction
  → let the model fantasize new text
- NB: need to represent words somehow
  → learnable embedding of a list of possible words to a fixed-length vector
  → alternative: go character by character

Different possibilities for inputs/outputs

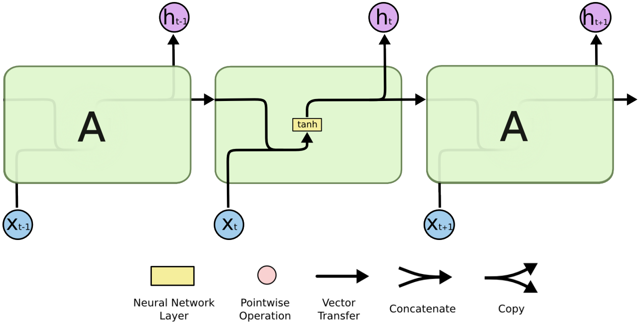


[1]

- one-to-one
  → non-recurrent neural network
- one-to-many
  → **sequence output**
  → e.g. image captioning
- many-to-one
  → **sequence input**
  → e.g. time series prediction, sequence classification
- many-to-many
  → **sequence input and output**
  → e.g. machine translation

Not always strict distinction, e.g. many-to-many models may also act as many-to-one

The inside of the box



[2]

$$\vec{h}_t = \tanh(\mathbf{W_{xh}}\vec{x}_t + \mathbf{W_{hh}}\vec{h}_{t-1} + \vec{b}) = \tanh(\mathbf{W_h}[\vec{x}_t, \vec{h}_{t-1}] + \vec{b}) \quad [,] := \text{concatenation}$$

- Simplemost example: concatenate input and state
- Then fully connected layer with bias and activation function
  $\rightarrow$ typically tanh for RNNs
- Output and updated state is the same
- This is what you get with `keras.layers.SimpleRNN`

More general

$$\vec{h}_t = \tanh(\mathbf{W_h}[\vec{x}_t, \vec{h}_{t-1}] + \vec{b}_h) \qquad \vec{y}_t = \sigma(\mathbf{W_y}\vec{h}_t + \vec{b}_y)$$

If output used as target y:

- Separate layer from hidden state to output
  $\rightarrow$ hidden state needs to carry over information on past sequence
- Combination like this theoretically turing complete [4]
- `keras` implementation example (hidden state size 32, 1D target for each time step):

```
rnn = tf.keras.Sequential([
    layers.SimpleRNN(32, return_sequences=True),
    layers.Dense(1, activation="sigmoid")
])
```

However: In practice struggles to learn long-range dependencies
(has a very short *short-term-memory*)

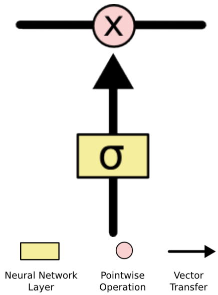
What have we learned so far?

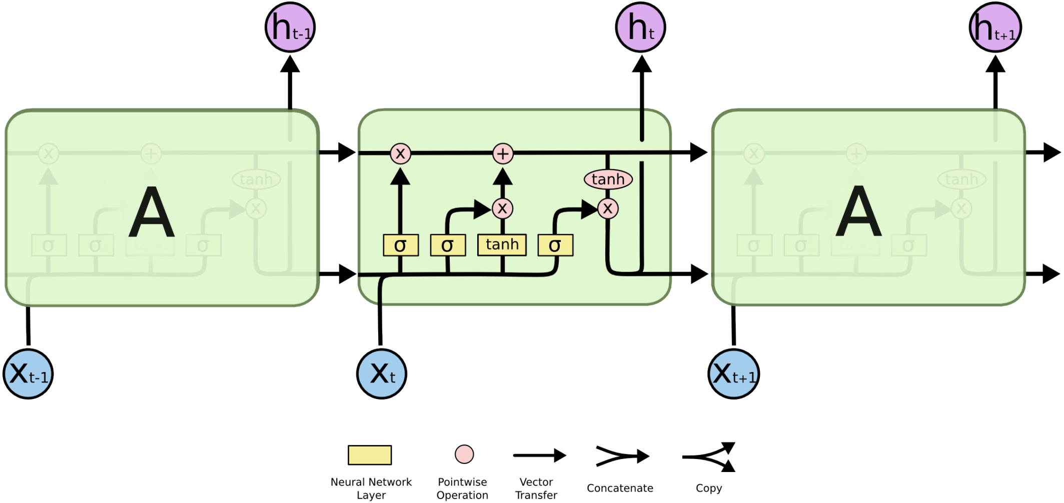
- Recurrent neural networks operate on sequences
- The weights are shared across the sequence
  $\rightarrow$ they are effectively trainable state-machines
- Depending on the application can have sequences both as input and as output
- Simplest recurrent cell consists of concatenation of (previous) hidden state with new input that is then passed through fully connected NN layer

The Long Short Term Memory (LSTM)

- Introduced in 1997 by Hochreiter und Schmidhuber
- Basic idea: make keeping a memory the default
  $\rightarrow$ called *cell state* C
  $\rightarrow$ NN layers learn what to forget and what to add to the memory
- Realized by *gates*:
  - NN layers with sigmoid activation function
  - Act as mask (numbers between 0 and 1) to be multiplied with a vector
    $\rightarrow$ can gradually turn on/off certain features
- LSTM until today the working horse for RNN architectures

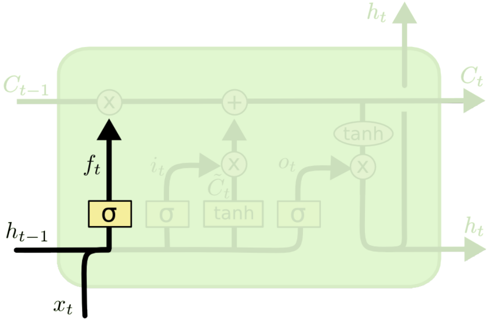
Gate:



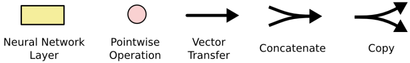


→ let's go through it step by step (using the illustrations from Christopher Olah's blog [2])

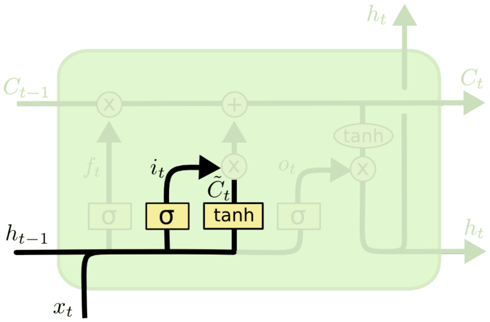
The forget gate - decide what to forget



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

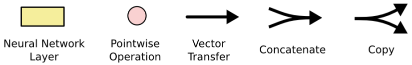


The input gate - decide what to add

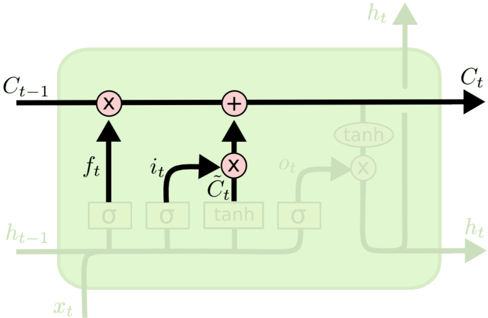


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

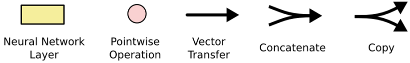
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$



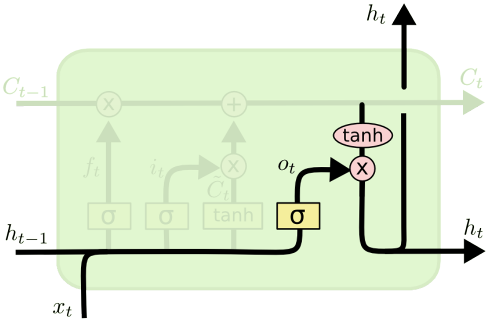
Update the cell state



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

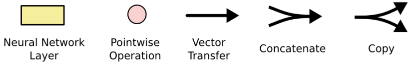


The output gate - decide what to output

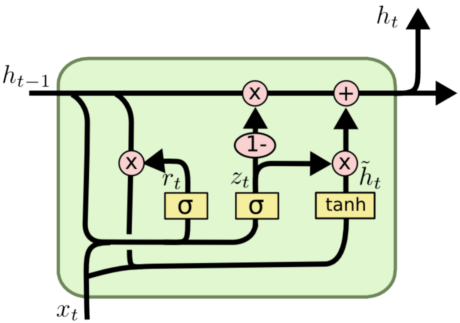


$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$



Gated Recurrent Units - GRU



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- modification of LSTM *without* separate cell state (just a single hidden state)
- less parameters and operations than LSTM
  $\rightarrow$ 2 instead of 3 gates
- in practice shown to have comparable performance

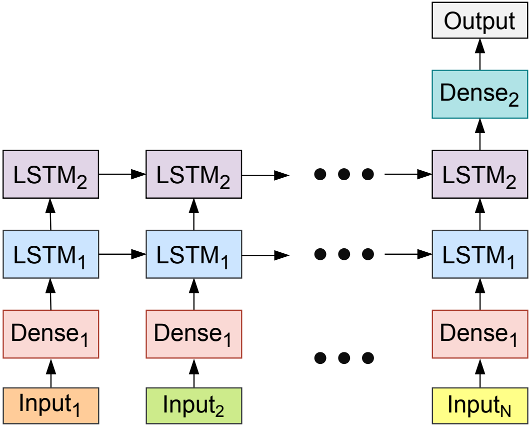
Model building with RNNs in keras

https://keras.io/guides/working_with_rnns/

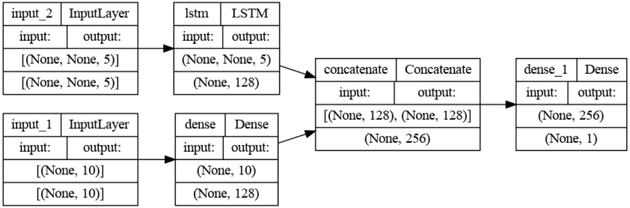
- Included as layers: `SimpleRNN`, `LSTM`, `GRU`
- Default mode: Take sequence input, output single vector
- For sequence output, pass `return_sequences=True`
- `Dense` layers will operate on the last dimension
  → can be on top of sequences (with shared weights)
- Use `TimeDistributed` wrapper for other layers

Stack RNN layers

```python
import tensorflow as tf
from tensorflow.keras.layers import (
    Dense, LSTM
)
model = tf.keras.Sequential([
    Dense(128, activation="relu"),
    LSTM(128, return_sequences=True),
    LSTM(128),
    Dense(1, activation="sigmoid"),
])
# batch_size, sequence_length, n_features
model.build((None, None, 4))
```

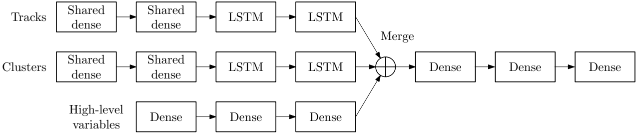


Combine sequence with fixed length input

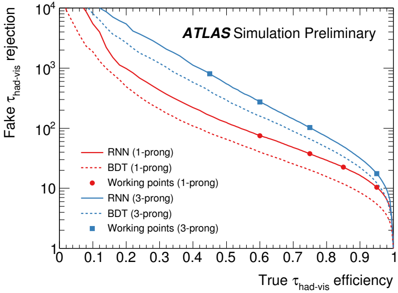



```python
import tensorflow as tf
from tensorflow.keras.layers import (
    Dense, LSTM, Input, Concatenate
)
def build_model(input_dim_fixed, input_dim_sequence):
    input_fixed = Input((input_dim_fixed,))
    h_fixed = Dense(128, activation="relu")(input_fixed)
    input_sequence = Input((None, input_dim_sequence))
    h_sequence = LSTM(128)(input_sequence)
    h = Concatenate()([h_sequence, h_fixed])
    out = Dense(1, activation="sigmoid")(h)
    return tf.keras.Model(
        inputs=[input_fixed, input_sequence],
        outputs=[out]
    )
```

Example application: τ identification at ATLAS



- Use LSTM on sequence of tracks/clusters
  $\rightarrow$ order by transverse momentum
  $\rightarrow$ encode variable length into fixed length
  $\rightarrow$ allows inclusion of low-level variables
- Greatly improved performance
  (previous classifiers used only high level variables)



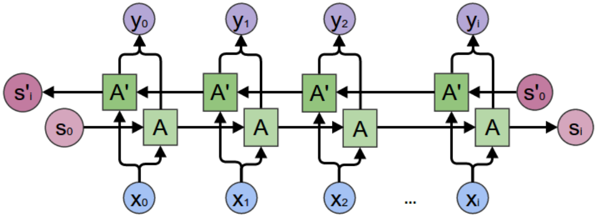
[1] ATL-PHYS-PUB-2019-033

Hands-on Exercises

https://github.com/nikoladze/deep-learning-rnn-tutorial

$\rightarrow$ start with understand_rnns.ipynb

Bi-Directional RNNs

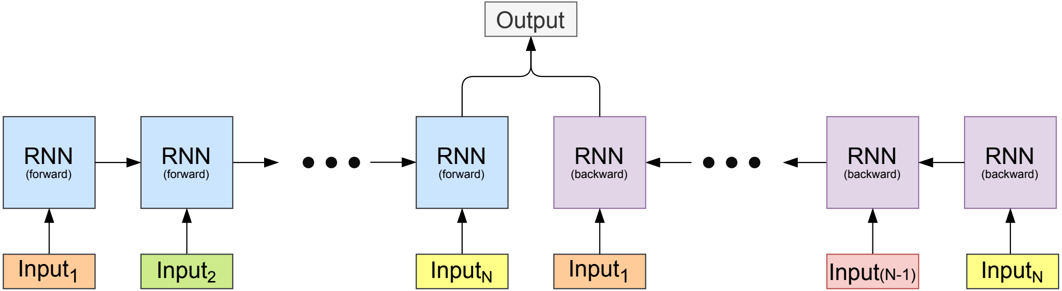


- prediction can depend on elements further in sequence
- have one RNN block going forward in sequence and one backward
- combine outputs of both
- useful if outputs at each time step depend on whole sequence
- `keras`: can wrap any RNN layer to be bi-directional

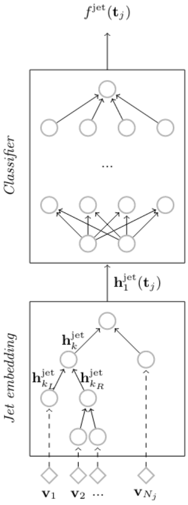
  → e.g. `layers.Bidirectional(layers.LSTM(32, return_sequences=True))`

[1] https://colah.github.io/posts/2015-09-NN-Types-FP

Note: without `return_sequences=True` you get this:



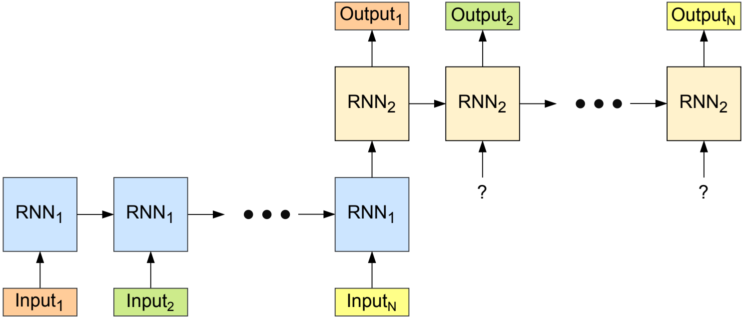
Recursive Neural Networks



- Generalizes the concept of RNN to directed acyclic graphs (RNNs are then the special case of a linear chain)
- Need to process graph in a defined order
  $\rightarrow$ from leave nodes to root nodes
- Example on the left: follow jet clustering sequence
- Possible update rule for fixed number of child nodes:
  concatenate N child vectors with node input, e.g. *N-ary Tree-LSTM*
- For trees/graphs with variable number of children:
  sum over child vectors, e.g. *Child-Sum Tree-LSTM*

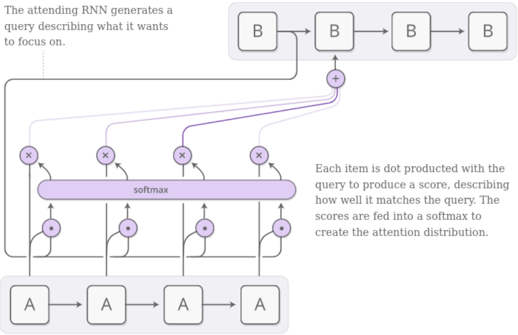
[1] *QCD-Aware Recursive Neural Networks for Jet Physics, arXiv:1702.00748*

Encoder-Decoder RNNs



- Used for delayed many-to-many models
- Prominent use-case: Machine Translation
- Need to decide what to feed as input to the decoder
  $\rightarrow$ 0? Previous Output? Encoded state? Both?
- In practice struggles for long output sequences

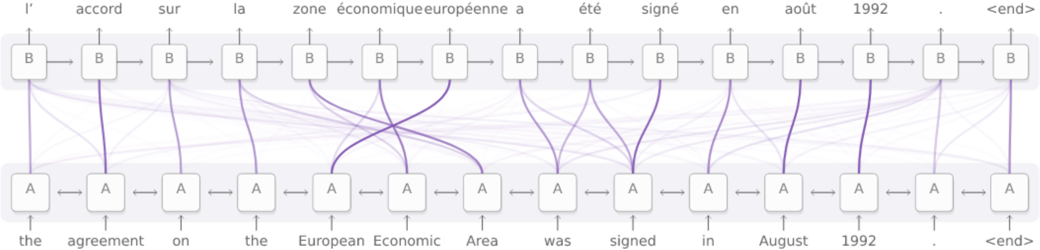
Attention mechanisms



- Have each element of decoder sequence *attend* to elements from encoder sequence
- Possible implementation: score from dot product of each encoder, decoder step pair
- Precursor of transformers - *Attention is all you need*

[1] https://distill.pub/2016/augmented-rnns

Example for machine translation



1 https://distill.pub/2016/augmented-rnns

RNNs vs Pointcloud and Graph models

- There is a trend to work with models of unordered sets
  - Pointclouds/Deep sets
  - Graphs
  - Transformers
- Sometimes motivated by data (e.g., no sensible ordering, graph structure)
- Also by computational advantages (RNNs inherently sequential)
  $\rightarrow$ Transformers for large language models

If you have ordered sequences and it's computationally doable, RNNs are still worth a try!

Challenge for the experts

Take Andrej Karpathy's shakespeare dataset
https://cs.stanford.edu/people/karpathy/char-rnn/shakespeare_input.txt
Can you beat this 3-layer LSTM with a transformer?

```
num_tokens = 67
tf.keras.Sequential([
    LSTM(512, input_shape=(None, num_tokens), return_sequences=True),
    Dropout(0.5),
    LSTM(512, return_sequences=True),
    Dropout(0.5),
    LSTM(512, return_sequences=True),
    Dropout(0.5),
    Dense(num_tokens),
])
```

- Task: predict the next character (67 different characters)
- Metric: mean cross entropy on chunks of 100 characters
  $\rightarrow$ 1st prediction based on 1 previous character, 2nd on 2, 3rd on 3 etc
  $\rightarrow$ evaluate on the last 10% of the dataset (don't use for fitting)
  $\rightarrow$ my result: 1.34 (random guessing would be $-\log(1/67) \approx 4.2$)

Random sample generated from this

```
QUEEN GERTRUDE:
If we shall do rush evenborth have her; and myself,
Led him from her puppily sons.

WARWICK:
Now, good lord, well your mind.

AUTOLYCUS:
Who dost inconcing yet before man: O, Give metto,
Direction. Therefond pass ga find to chain:
A gopposite him 'scapes almost, hath will she sweet men
From a lord deivise chequal were unworthself.'

CELIA:
No.

JAQUES:
It is Achilles' fain of our ladies, about that was gone.
Now many unseen bring off.

PORTIA:
Did you go of the law, where accommodated
Deserve with madness: comes for me no more:
What we will our soldier for and true by worth,
And bad talk o' the scorn of his conversant,
But now harm to this none wise love.
```

References

- [1] A. Karpathy, *The Unreasonable Effectiveness of Recurrent Neural Networks*
  http://karpathy.github.io/2015/05/21/rnn-effectiveness
  → Youtube channel: https://www.youtube.com/@AndrejKarpathy
- [2] C. Olah, *Understanding LSTM networks*
  https://colah.github.io/posts/2015-08-Understanding-LSTMs
- [3] Uwe Klemradt's lecture on ErUM-Data Hub Train-the-trainer workshop
  https://indico.scc.kit.edu/event/2645/contributions/9861/attachments/4962/7494/Lecture_RNN_final.pdf
  → Recording on https://erumdatahub.de/videos
- [4] I. Goodfellow et al., *Deep Learning*
  https://www.deeplearningbook.org

Hands-on Exercises

https://github.com/nikoladze/deep-learning-rnn-tutorial