Neural Networks 2 - Sequences and RNNs, Introduction to natural language processing

18NES2 - Week 10, Winter semester 2025/26

Zuzana Petříčková

December 2, 2025

What We Covered Last Time

Applications of CNNs

- Object detection
- Instance segmentation
- And others (1D convolution, 3D convolution,...)

Processing sequences

- About sequential data
- Time series prediction
- Practical example - Jena Climate Dataset
- Recurrent Neural Networks - Introduction

This Week

1. Sequential data
   - Time Series Forecating
2. Recurrent Neural Network (RNN)
   - Vanilla RNN
   - LSTM and GRU
   - RNN Architecture patterns
3. Natural Language Processing
   - Deep Learning Architectures for Text
   - Text Data Preprocessing
   - Bag-of-words representation

Sequential Data

Time series

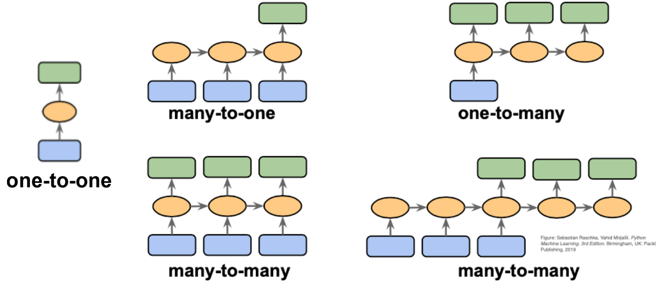
- Different granularity: daily stock prices, hourly electricity consumption, weekly store sales, ...
- Different dynamics: website traffic, credit-card transactions, seismic activity, weather evolution, ...

Sequential data are not only time series:

- **Audio:** speech recognition, speaker identification, emotion detection, acoustic localization, music analysis
- **Text:** sentiment analysis, machine translation, next-word prediction
- **Video:** action recognition, object tracking, trajectory prediction, video captioning
- **Biological signals:** DNA sequence analysis, heart-rate monitoring (ECG), . . .

Types of Tasks on Sequential Data

- **one-to-one** — standard classification
- **many-to-one** — sentiment analysis, action recognition
- **one-to-many** — image captioning, sentence tagging, music generation
- **many-to-many** (direct /delayed) — object tracking, machine translation,



Source: S. Raschka: Introduction to Recurrent Neural Networks

Time Series

Time series

- Sequential data with typical dynamics:
  - periodicity (daily, annual, ...)
  - trends in time: regular regime, sudden spikes,...

Typical tasks:

- **Forecasting** — predict the next value (many-to-one) or the next sequence (many-to-many)
- **Classification** — e.g., bot vs. human web visitor, heart-attack risk from ECG
- **Event detection** — seismic activity, keyword spotting ("OK Google")
- **Anomaly detection** — unusual behavior in network traffic
  - usually solved with unsupervised learning: clustering, autoencoders

Example: Time Series Forecasting

Example: Jena Climate Dataset time_series_jena.ipynb

- Meteorological data recorded at the Max Planck Institute in Jena (Germany), years 2009–2016.
- 15 features (timestamp, temperature, pressure, humidity, wind speed/direction, ...)
- Measurements taken every 10 minutes — roughly $\sim$ 400,000 samples.

Task

- Predict the temperature **24 hours into the future** using the previous 5 days of data.
- Use 14 input features (timestamp omitted).
- We downsample to hourly measurements $\rightarrow 24 \times 5 = 120$ time steps.
- Each training example has shape 120×14, and the output is a single value (temperature in 24 hours).

Example: Time Series Forecasting

Loading and preparing the data

- **Example:** reading the CSV file and creating time-series datasets using Keras utilities.
- **Important:** Validation and test sets must follow the training set **in time**.
  - Common mistake: randomly shuffling data before splitting into train/val/test.
  - Only the **training samples** may be shuffled.

Baseline level

- **Naive forecast:** "The temperature in 24 hours will be the same as now."
- Baseline: $MAE_{\text{test}} = 2.62^{\circ}\mathrm{C}$ — surprisingly strong for this dataset.

Example: Time Series Forecasting

Solution using an MLP

- The MAE remains close to the baseline.
- Why is it not better?
    - The MLP sees all 120×14 numbers at once $\rightarrow$ it must "find a needle in a haystack".
    - It has no notion of temporal structure.

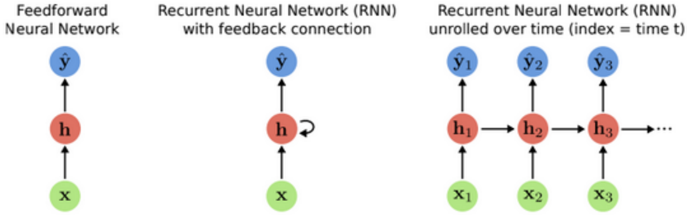
Solution using a 1D CNN

- Uses 1D convolution along the time axis.
- Performs even worse than the baseline.
- Why?
    - Time series are **not shift-invariant**.
    - The order matters — recent history is more important than distant history.

The MLP and CNN models fail. What to do next?

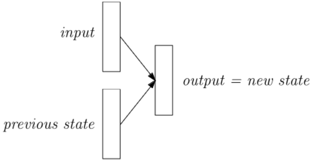
- Try a model dedicated to sequential data: a recurrent neural network (RNN)

Recurrent Neural Network (RNN)

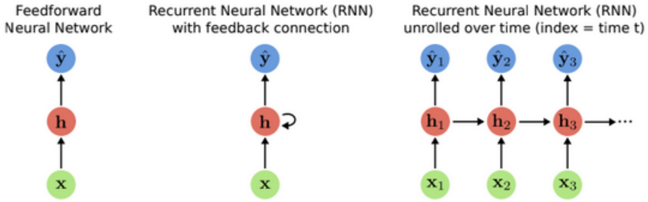


Recurrent NN model = a model with cycles

- **Idea:** Neurons maintain an internal state (**memory**).
- Output depends not only on the current input but also on the previous state.
- RNN neurons are sometimes called *memory cells*.

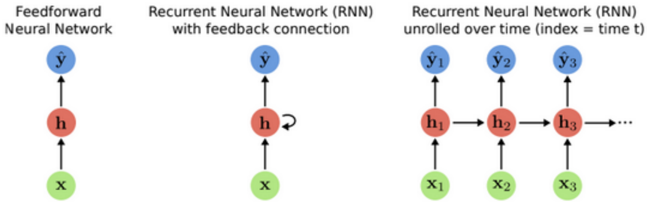


Vanilla (Elman) RNN



- The neuron has a hidden state: $h_t = f_h(h_{t-1}, x_t)$
- Output: $y_t = f_y(h_t)$
- The model processes the whole input sequence:
  - of arbitrary length,
  - can be "unrolled" over time (right figure),
  - functions f_h and f_y share parameters across time steps.
- The hidden state is reset for each new input sequence.

Vanilla (Elman) RNN: Equations



- Hidden state:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b_h)$$

- Output:

$$y_t = W_y h_t + b_y$$

- Parameters W, b stay the same for all time steps.

Vanilla RNN – unrolled

- Examples of unrolled architecture for different sequence-to-sequence mappings:

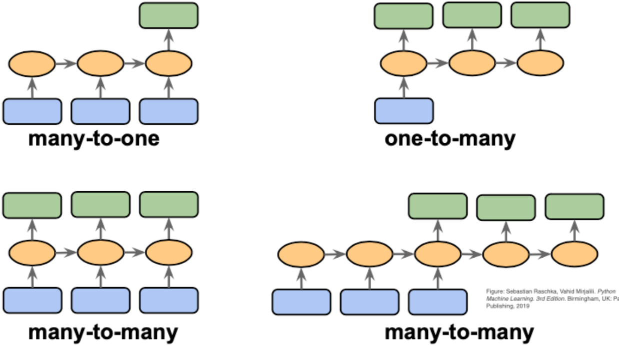
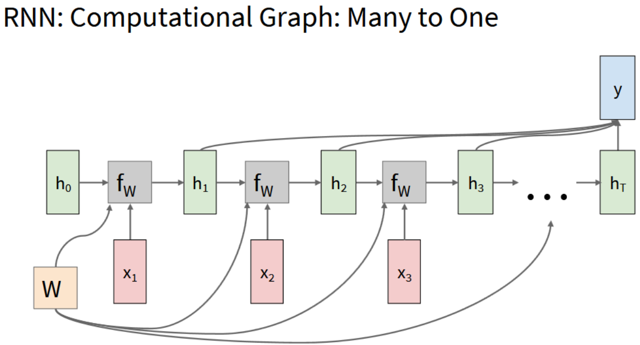


Figure: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Birmingham, UK: Packt Publishing, 2019

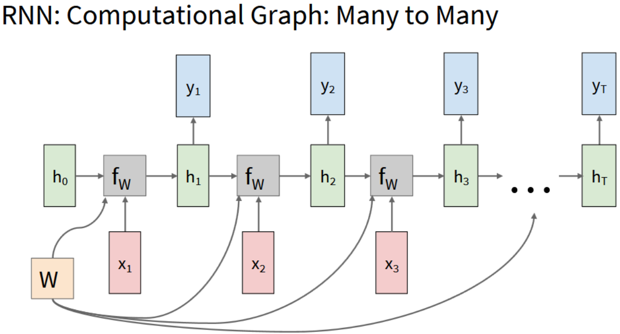
Vanilla RNN – unrolled



- RNN unrolled in time can be viewed as a deep feed-forward neural network
- The main difference: unrolled units share parameters (weights and biases)

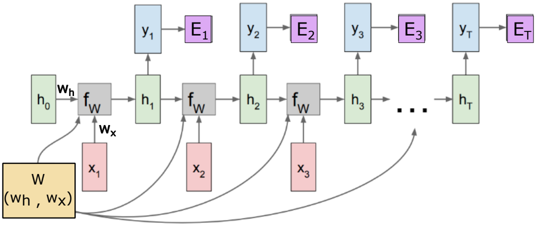
Source: S. Raschka: Introduction to Recurrent Neural Networks, https://sebastianraschka.com/blog/2021/dl-course.html

Vanilla RNN – unrolled



Source: S. Raschka: Introduction to Recurrent Neural Networks,
https://sebastianraschka.com/blog/2021/dl-course.html

Backpropagation Through Time (BPTT)



Source: S. Raschka: Introduction to Recurrent Neural Networks, 2021

- An extension of standard Backpropagation for RNN
- Shown for a many-to-many architecture.
- Gradients are backpropagated through all time steps at once (for a sequence or batch of sequences).

Practical examples

Practical example: RNN Layers in Keras

- **RNN_layers.ipynb**

Practical example: Jena Climate Dataset

- **time_series_jena.ipynb**

Challenges of Training Vanilla RNNs

Backpropagation Through Time (BPTT)

- Problematic computation of the error terms: too many multiplications of the derivatives

Vanishing gradients:

- Many multiplications of small derivatives $\rightarrow$ gradients shrink zo zero.
- The model fails to learn long-term dependencies. It forgets quickly.

Exploding gradients:

- Many multiplications of large values $\rightarrow$ gradients blow up.
- Leads to unstable training (NaNs).

$\Rightarrow$ Vanilla RNNs are hard to train on long sequences.

Dealing with Vanishing and Exploding Gradients

Exploding gradients: gradient clipping

- Rescale the gradient when its norm exceeds a threshold.

Vanishing gradients: need of explicit memory

- The model forgets long-term information:
  "The dogs in the neighborhood are ..." (barking)
- Solutions:
  - Echo State Networks,
  - **LSTM** (Long Short-Term Memory, 1997),
  - **GRU** (Gated Recurrent Unit, 2014).

LSTM (Long Short-Term Memory)

Key idea:

- Store the information for later (similar to the idea of skip connections)

Baggage carousel (conveyor): created by AI and not perfect :)

LSTM (Long Short-Term Memory)

Key idea:

- Replace repeated multiplications with additive memory updates.
- Two states:
  - short-term state h_t (the cell actual output),
  - long-term state c_t (the “cell state”).
- Four gates control information flow: input (i), forget (f), output (o), and the candidate update (g).

LSTM cell:

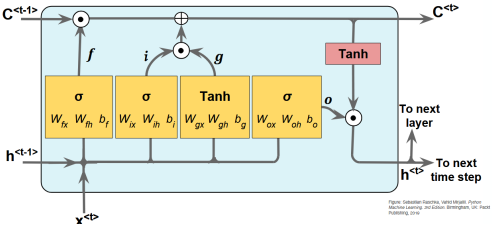
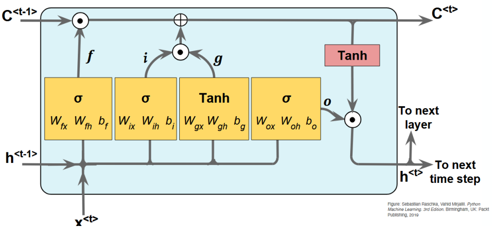


Figure: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Birmingham, UK: Packt Publishing, 2019

LSTM: Gate Interpretation

- Candidate state: g_t — what content to write.
- Input gate: i_t — how much new information to write.
- Forget gate: f_t — how much of the old memory to erase.
- Output gate: o_t — how much of the cell state to expose.

LSTM cell:



Source: S. Raschka: Introduction to Recurrent Neural Networks,
https://sebastianraschka.com/blog/2021/dl-course.html

Example: Time Series Forecasting

Practical example: Jena Climate Dataset

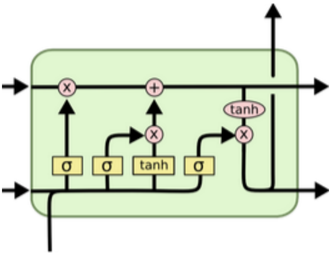
- **time_series_jena.ipynb**

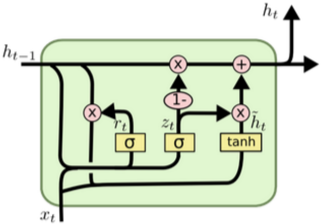
Solution using an LSTM (simple recurrent model)

- Finally improves over the baseline.
- LSTM explicitly models temporal dependencies and varying importance of past events.

Gated Recurrent Unit (GRU)

- Simpler than LSTM — fewer parameters, faster training.
- Two gates:
  - update gate — how much of the new information to write,
  - reset gate — how much of the old state to forget.
- Only one state vector (no separate cell state).





Source: dprogrammer.org

GRU: Advantages and Disadvantages

Advantages:

- Simpler structure, fewer parameters, faster to train.
- Performs similarly to LSTM on many tasks.
- Often better for smaller datasets.

Disadvantages:

- Less expressive than LSTM for complex temporal patterns.
- Sometimes more sensitive to hyperparameters.

RNNs Architecture patterns

- gradient clipping - resolves the exploding gradient problem
- regularization (recurrent dropout, layer normalization)
- stacked RNNs
- bidirectional RNNs

Practical examples

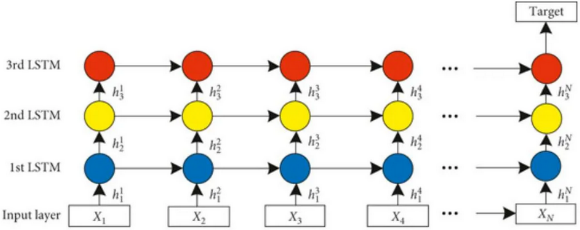
- **RNN_layers.ipynb**
- **time_series_jena.ipynb**

Recurrent Neural Networks and Generalization

- RNNs are prone to overfitting, especially on long sequences.
- Standard dropout applied before a recurrent layer does not work well – it disrupts learning long-term dependencies.
- Instead, we use **recurrent dropout**:
    - randomly disables recurrent connections (between time steps),
    - uses the **same dropout mask at every time step** for consistency.
- **Layer normalization** (instead of batch normalization) is commonly used in deeper RNNs.

Stacked (Multi-layer) RNNs

- Increase model capacity by stacking multiple recurrent layers.
- Capture more complex and longer-term dependencies.
- Higher risk of overfitting → combine with dropout or normalization.



Source: https://python.plainenglish.io/stacked-rnns-in-nlp-936e6eecf37a

Bidirectional Recurrent Neural Networks

- Process the sequence in both forward and backward directions.
- Particularly useful for natural language processing.
- Can be viewed as a simple **ensemble** of forward and backward models.

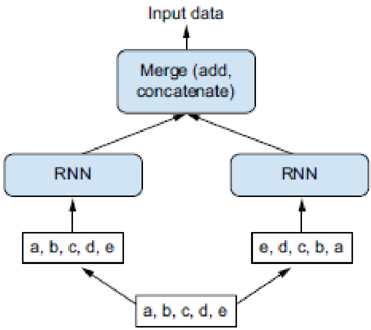


Image source: F. Chollet, *Deep Learning with Python*, Fig. 13.13

Recurrent Neural Networks — Summary

- RNNs can process input sequences of variable length.
- Support multiple architectures: one-to-many, many-to-one, many-to-many, etc.
- Suitable for **sequential data** with temporal dependencies (time series, text, etc.).
- Simple RNNs are hard to train → in practice we use LSTM, GRU, and gradient clipping.
- RNNs are sensitive to overfitting → regularization is crucial.
- The architecture is "deep" in the **time** dimension; stacked RNNs deepen it further.
- Many extended RNN architectures exist.
- **Weaknesses**: difficult GPU parallelization due to sequential computation.
- RNNs influenced the design of modern **transformers**.

Deep Learning and Text

Natural Language Processing (NLP)

- A broad and rapidly evolving area of AI.
- Focuses on understanding, analysing, and generating natural language.

Example tasks:

- **Many-to-one:**
  - Text classification, sentiment analysis.
  - Content filtering (spam detection, toxic content), keyword spotting.
  - Language modeling: next-word prediction, spelling correction.
- **Many-to-many:**
  - Machine translation (e.g., Google Translate).
  - Text summarization.
  - Text generation (GPT, BERT), chatbots (ChatGPT, Gemini, etc.).

Deep Learning Architectures for Text

MLP:

- Does not capture sequential structure or long-term dependencies.
- Works well only with heavy preprocessing (e.g., n-grams).
- Useful for small datasets and simple classification tasks.

RNN (Seq2Seq, LSTM, GRU)':

- Popular from 2014-2017 for translation, sentiment analysis, language modelling.
- Capture sequential structure, but slow to train on GPU and hard to parallelize.

CNN for text (TextCNN, etc.):

- Often underestimated in NLP, yet very effective.
- Capture local patterns and short-range dependencies.
- Suitable mainly for: classification, content filtering, keyword detection.

Deep Learning Architectures for Text

Transformers (BERT, GPT, ...)

- Dominant architecture in NLP since 2018.
- Use **self-attention** to model global context.

Advantages:

- Excellent parallelization and efficiency during training.
- Can model long-range dependencies.
- State of the art in most NLP tasks.

Disadvantages:

- Very high memory and GPU requirements.
- High energy consumption (training and inference).
- Require large training datasets.

Deep Learning Architectures for Text

Two possible approaches to text data:

- Treat text as a set of words: **bag of words**
  - Ignores word order and long-range dependencies.
  - Suitable for simple models (MLP) and small datasets.
  - We have already seen this approach
- Treat text as a sequence of words
  - Preserves word order, context, and syntactic/semantic structure
  - Used in CNNs, RNNs (LSTM/GRU), Transformers

→ require different data representations

Which model to choose?

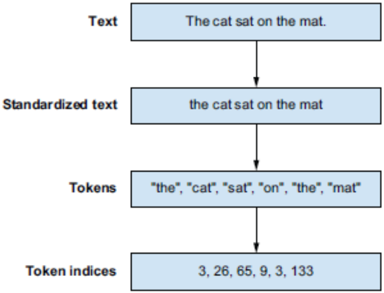
- Depends on the task, sequence length, training dataset size, compute resources
- For a "small" tasks → simpler models often generalize better

Text Data Preprocessing Pipeline

- We have to convert **raw text** into **numerical tensors** suitable for neural networks.

Typical pipeline:

1. **Standardization:**
2. **Tokenization:**
3. **Vectorization:**
   - integer indexing
   - bag-of-words / TF-IDF
   - or embeddings (learned or pretrained)



Source: F. Chollet, *Deep Learning with Python*, Fig. 14.1

Text Data Preprocessing Pipeline: Standardization

Goal: Reduce variability in raw text before tokenization.

Common standardization steps:

- convert all characters to lowercase
  (e.g., “The CAT” → “the cat”).
- remove punctuation (.,?!’:,...) or HTML tags (for web pages)
- remove or normalize special characters (e.g., emojis, accents, currency symbols)
- handle whitespace (e.g., collapse repeated spaces, ...)

Optional linguistic normalization:

- **Stemming** – reduce words to their root
  (e.g., “playing”, “plays” → “play”)
- **Lemmatization** – reduce words to dictionary form
  (e.g., “better” → “good”, “were” → “to be”)

In Keras:
TextVectorization(standardize=’lower_and_strip_punctuation’)

Text Data Preprocessing: Tokenization

Tokeniation: split text into tokens

1) Word tokenization:

- Natural and intuitive representation (split by spaces).
- Works well for sequential models (RNN, Transformers), CNN.
- Problems: rare words, typos, rich morphology (e.g., "running", "run", "ran"), suffers from large vocabularies

2) n-grams:

- Suitable for non-sequential models (MLP).
- Captures short-range context (bigrams, trigrams).
  - Example for the sentence *"the cat sat on the mat"*: $\rightarrow$
  - Bigrams: "the", "the cat", "cat", "cat sat", "sat", "sat on", "on", "on the", "the", "the mat", "mat"
  - Trigrams: "the", "the cat", "the cat sat", "cat", "cat on", "cat on the", ...,"on the mat", "the", "the mat", "mat"

Text Data Preprocessing: Subword Tokenization

3) Character-level tokenization:

- Useful for languages without whitespaces (Chinese, Japanese).
- Robust to typos and unknown/rare words
- Problems: loses semantic information and produces very long sequences → harder learning

4) Subword tokenization:

- Splits words into smaller meaningful units
  "unbelievable" → *"un"*, *"believ"*, *"able"*
- Used in most modern NLP models (BERT, GPT, T5, ...)
- Solves the problem of rare words, typos and morphological variants — new words can be composed from known pieces
- Keeps a compact vocabulary but still preserves semantic meaning
- BPE (Byte Pair Encoding) (GPT-2/3), WordPiece (BERT, RoBERTa), SentencePiece (T5)

Text Data Preprocessing: Indexing

Indexing:

- Build a **vocabulary** of tokens (from the training set or a larger corpus).
- Assign each token a unique integer index (the order is based on word frequency).
- Limit vocabulary size to the n most frequent tokens (efficiency, better training).
- Conventions:
  - index **0**: "not a token" (e.g., padding to equalize sequence lengths)
  - index **1**: "token exists but is out-of-vocabulary (OOV)"
  - higher indices: : most frequent token = index 2, and so on

Text Data Preprocessing: Vectorization

Vectorization:

- Convert text to a sequence of token indices, e.g.: "the cat sat on the mat" → [3, 28, 65, 9, 3, 1, 0]
- Sequences can be:
  - variable-length
  - fixed-length (with padding/truncation)

Practical Example: IMDB Dataset :

- **50,000** movie reviews: **25,000 train** / **25,000 test**
- Sentiment analysis (binary classification): *positive* / *negative review*

Practical Example: IMDB Dataset

Original reviews (toy example)

Review 1: *"The movie was great."*
Review 2: *"The movie was definitely not good."*

Vocabulary (top 6 words)

{PAD: 0, OOV: 1, the: 2, movie: 3, was: 4, great: 5, not: 6, good: 7}

Integer-encoded representation

	Word indices	With padding
Review 1	[2, 3, 4, 5]	[2, 3, 4, 5, 0, 0]
Review 2	[2, 3, 4, 1, 6, 7]	[2, 3, 4, 1, 6, 7]

(Each review becomes a sequence of integers; lengths differ)

Text Data Preprocessing: Vectorization

Vectorization in Keras: TextVectorization

- Provides the whole end-to-end pipeline.
- Key parameters:
  - **max_tokens**: maximum vocabulary size (e.g., 200 000 most common words)
  - **output_mode**: - "int" (sequence of indices), - "binary", "count", "tf-idf" (bag-of-words)
  - **output_sequence_length** (e.g., 200 tokens)
  - **ngrams**: generate n-grams automatically
- Apply to the dataset using **adapt()**.

Practical example:

- **text_processing_layers.ipynb**

Text Data Preprocessing: Vectorization

Encoding the vectorized text:

- sequences like [3, 28, 65, 9, 3, 0] must be converted into tensors suitable as neural network inputs.
- Two main representations:
  1. Bag-of-words representation - works best with MLPs
  2. Sequential representation - works best with RNNs, CNNs, transformers

1) Bag-of-Words representation

- represent each token as a one-hot vector [...,0,1,0...] (size = vocabulary size)
- represent the text as a multi-hot vector [...0,1,1,0,1,0...] (size = vocabulary size)
- order of words is lost

Practical Example: IMDB Dataset: Bag-of-Words Representation (BoW)

Original reviews (toy example)

Review 1: *"The movie was great"*
Review 2: *"The movie was definitely not good"*

Vocabulary (top 6 words)

{PAD: 0, OOV: 1, the: 2, movie: 3, was: 4, great: 5, not: 6, good: 7}

Bag-of-Words representation (binary / multi-hot encoding)

	OOV	the	movie	was	great	not	good
Review 1	0	1	1	1	1	0	0
Review 2	1	1	1	1	0	1	1

(Each review is represented by a fixed-length binary vector of vocabulary size.)

Encoding the vectorized text

1) Bag-of-Words representation

- multi-hot variant: represent the text as a multi-hot vector [...0,1,1,0,1,0...] (size = vocabulary size)
- counts variant: represent the text as a vector of token counts
- TF-IDF variant: weights the token by its importance in text/corpus [...0,2.8,1.5,0,0.9,0...]

Advantages

- Easy to implement and interpret
- Works surprisingly well for simple tasks (e.g., sentiment or topic classification)

Limitations

- Loses information about the word order, context and word relationships
- Produces large, sparse matrices with high memory requirements
- **No notion of similarity:** all words are equally distant