Neural Networks 2 - Natural language processing

18NES2 - Week 11, Winter semester 2025/26

Zuzana Petříčková

December 9, 2025

What We Covered Last Time

Processing sequences

- Recurrent Neural Networks - models, training, aplications, architecture pattersns
- Time series prediction - Jena Climate Dataset

Natural Language Processing

- Introduction to text preprocessing
- TextVectorization Layer in Keras

This Week

1. Natural Language Processing
   - Deep Learning Architectures for Text
   - Text Data Preprocessing
   - Bag-of-words representation
   - Word Embedding
2. Transformers
   - Attention
3. Introduction To Generative models for Image generation

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

$\rightarrow$ require different data representations

Which model to choose?

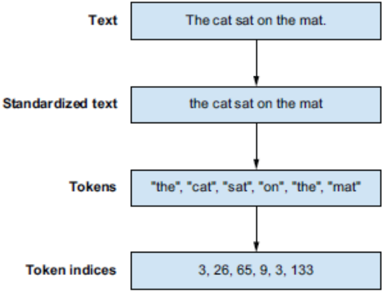
- Depends on the task, sequence length, training dataset size, compute resources
- For a "small" tasks $\rightarrow$ simpler models often generalize better

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" $\rightarrow$ "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" $\rightarrow$ "play")
- **Lemmatization** – reduce words to dictionary form
  (e.g., "better" $\rightarrow$ "good", "were" $\rightarrow$ "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" $\rightarrow$ [3, 28, 65, 9, 3, 1]
- Sequences can be:
  - variable-length
  - fixed-length (with padding/truncation)

Encoding the vectorized text:

- Vectorized sequences can be further transformed into tensors more suitable as neural network inputs, e.g. Bag-of-Words (BoW)

Text Data Preprocessing: Vectorization

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)

Vectorization in Keras: TextVectorization Layer

- Provides the whole end-to-end pipeline: **standardization → tokenization → indexing → vectorization**
- Vocabulary is learned from data

Key parameters:

- **standardize**: "lower_and_strip_punctuation" (standardization)
- **split**: "whitespace" (word-level tokenization)
- **output_mode**: "int" (sequence of token IDs)
- **max_tokens**: size of the vocabulary (e.g., 200 000 most common words)
- **output_sequence_length**: fixes the length of sequences (padding/truncation)

Practical example: text_processing_layers.ipynb

Vectorization in Keras: TextVectorization Layer

Using the layer:

- Build the vocabulary with **adapt(dataset)**
- Inspect vocabulary with **get_vocabulary()**
- Apply to raw text to receive vectorized text

Practical example: text_processing_layers.ipynb

Text Data Preprocessing: Vectorization

Encoding the vectorized text:

- The sequential representation is not suitable for some NN models (e.g., MLPs)
- Vectorized sequences can be transformed into tensors more suitable as neural network inputs.
- Two main representations:
  1. Bag-of-Words (BoW) representation - works best with MLPs
  2. Keeping 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

Bag-of-Words representation

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 text is represented by a fixed-length binary vector of vocabulary size.)

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...]
- n-grams (can be combined with multi-hot/counts/TF-IDF/...)

Vectorization in Keras: TextVectorization Layer

text_processing_layers.ipynb

- Additional parameters for various data representations:
  - **output_mode**:
    - "int"– sequence of token IDs (for RNN, CNN, Transformers)
    - "binary"– multi-hot bag-of-words vector
    - "count"– bag-of-words with counts
    - "tf-idf"– weighted bag-of-words
  - **ngrams**: automatic generation of *n*-grams for bag-of-words models

Bag-of-Words representation

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

Practical examples

Practical example: IMDB Dataset
text_classification_IMDB.ipynb

- binary classification task: sentiment analysis *positive / negative review*
- Demonstrates: preprocessing, tokenization, vectorization, various models

Models trained on Bag-of-Words

- Simple MLP / linear classifier on Bag-of-Words — **fast**, surprisingly strong baseline
- MLP trained on **bigrams** — captures limited local context → **even better accuracy**
- Both models train fastly but suffer from large memory requirements (very high-dimensional sparse vectors)

Encoding the vectorized text

2) Sequential representation

- **one-hot sequence**: each word is one-hot encoded [...,0,1,0...] $\rightarrow$ text becomes a binary matrix of shape (sequence length, vocabulary size)
- **word embeddings**: most commonly used

Practical example: IMDB Dataset

- RNN trained on **one-hot sequences** - extremely slow, large memory requirements, results comparable to the baseline

Word Embedding

Why not one-hot encoding?

- The matrix (sequence length $\times$ vocabulary size) is very high dimensional and extremely sparse (mostly zeros).
- Further limitation: all pairs of words are equally distant in this space — no notion of similarity

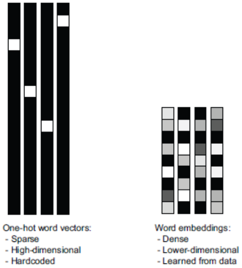


Image source: F. Chollet, *Deep Learning with Python*, Fig. 14.3

Word embeddings

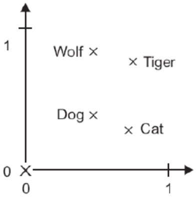
- Each word is represented as a **dense** vector of small dimensionality (e.g., 64–512).
- Words with similar meaning are close in this vector space.

Embedding Space Intuition

Simplified view: semantic dimensions

Each dimension of the embedding vector may encode a latent semantic feature of words:

- Semantic similarity: e.g., "pet-likeness": dog: 1.0, cat: 1.0, snake: 0.5, whale: 0.2, battery: 0.0
- Grammatical relations: verb, predicate, word stem
- Analogical relations: "king - male + female = queen" "king + plural = kings"



Source: F. Chollet, Deep Learning with Python, Fig. 14.4

Embedding Layer

- A simple lookup table mapping token indices to dense vectors.
- Input: 2D tensor (batch_size, sequence_length).
- Output: 3D tensor (batch_size, sequence_length, embedding_dimension).
- Internally contains an **embedding matrix** of shape (vocabulary_size $\times$ embedding_dimension).

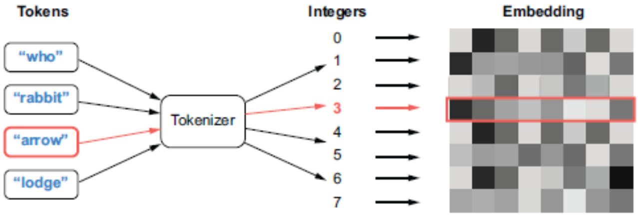


Image source: F. Chollet, *Deep Learning with Python*, Fig. 14.5

Embedding Layer

How to obtain word embeddings?

1. **Learned from scratch** during model training → task-specific embeddings
2. **Pretrained static embeddings**
   - **GloVe** (Stanford), **Word2Vec** (Google), **fastText** (Meta)
   - learned offline on huge corpora (Wikipedia, Common Crawl).
   - useful for small datasets or when training from scratch would overfit
3. **Pretrained custom embeddings**
   - create custom GloVe/Word2Vec embeddings trained on your data - e.g., via Continuous Bag of Words (CBOW) model
   - often yields better performance for domain-specific tasks (medical, legal, sentiment, chat logs, reviews...)
4. **Contextual embeddings**
   - produced by pretrained Transformer models (BERT, RoBERTa, DistilBERT, GPT)
   - each word gets a **different embedding depending on context**

Embedding Layer

Practical example: IMDB Dataset

- RNN trained on embeddings trained from scratch
  - the training is slow,
  - the model overfitts early (the accuracy is affected by the limited sequence length)
- RNN trained on pretrained embeddings (GloVe on Wikipedia and Common Crawl) - the overfitting is reduced
  - reduces overfitting
  - worse results for domain-specific texts
- RNN trained on pretrained embeddings (self-trained CBOW on IMDB)
  - better results for domain-specific texts

Embedding Layer

Continuous Bag of Words (CBOW) model

- We create a dataset using a sliding window over the text.
- For each window, we predict the **center word** from the surrounding context words.
- We train a small model on this task and learn a **word embedding** as its first layer.
- We then reuse this trained embedding layer in our **RNN sentiment model**.

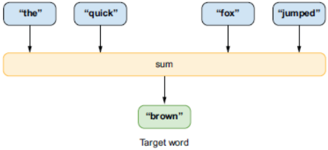


Image source: F. Chollet, *Deep Learning with Python*, Fig. 14.6

Practical example: IMDB Dataset - Overal results

- Unigrams + MLP: accuracy: 0.88
- Bigrams + MLP: accuracy: 0.90

Shortened sequences (60 words)

- Binary encoding +LSTM : 0.87
- Co-trained embeddings +GRU: 0.87
- Pre-trained embeddings +GRU: 0.88
- CBOW embeddings +GRU: 0.89

Takeaways:

- Always start with a simple baseline — bag-of-words + linear model is strong on small datasets.
- RNN-based models outperform MLP only if input representations carry enough semantic information.
- Self-supervised CBOW embeddings adapt to the domain and therefore outperform generic pretrained embeddings.

Practical example: IMDB Dataset - Overal results

Why do bigram-based MLP models achieve the best results?

- The dataset is relatively small (20,000 samples) – RNNs and Transformers need more data.
- Bigrams capture key sentiment cues ("not good", "very bad", "absolutely wonderful").
- Bag-of-words MLPs have a simple optimization landscape $\rightarrow$ fast and stable training.
- Sequential models must learn word order and context $\rightarrow$ much harder on small datasets.

Can we improve beyond these results?

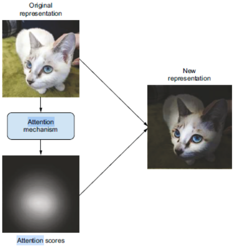
Yes – by using **finetuning** of a pretrained language model (e.g., BERT, DistilBERT, RoBERTa), which dramatically outperforms classic RNN/MLP/CNN models on small datasets.

Transformers

- Ashish Vaswani et al.: *Attention Is All You Need* (2017), .
- the key innovation: the **attention** mechanism, especially **self-attention**

Core idea: when people read text, they focus more on some parts than on others.

- what if a model could do the same and weight tokens by importance?



Source: F. Chollet: *Deep Learning with Python*, Fig. 11.5

Transformers

Attention mechanisms can do far more than simply weight tokens:

- they allow the model to fully capture and use **context**

"He sat down by the **bank***."*

- river bank?
- financial bank?
- blood bank?
- power bank?

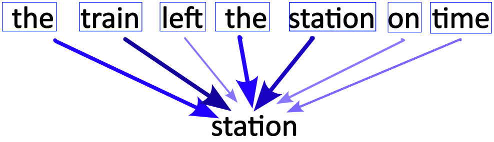
"He will **charge** *at dawn."*

- attack?
- ask for payment?
- load a battery?

"**See** *you soon*" vs. "*I* **see** *what you mean*"

Attention

Attention: which parts of the sentence are most important for the current token?

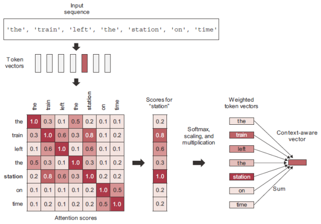


token embedding → context-aware embedding

Attention

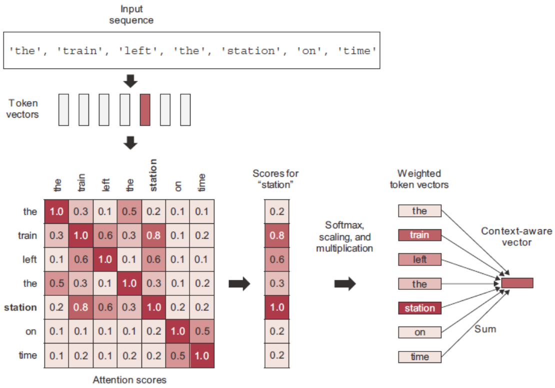
token embedding → context-aware embedding

1. Tokens are first converted into embedding vectors.
2. Compute an **attention score** for each pair of tokens (a dot product between their embeddings that measures how strongly they relate).
3. For each token, create a new **context-aware representation**: a weighted sum of all token embeddings, using the attention scores as weights.



Source: F. Chollet: *Deep Learning with Python*, Fig. 11.6

Attention - for text classification (many-to-one)

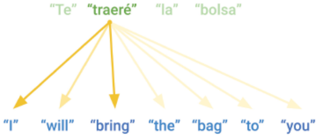


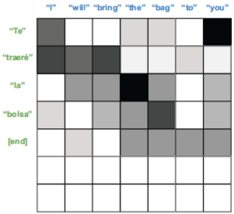
Source: F. Chollet: *Deep Learning with Python*, Fig. 11.6

- `output = sum(input * attention_score(input, input))`

Attention - General Formula

- In more general settings (e.g., machine translation, text generation):
- **Attention:** which parts of the *source* sentence are most important for generating the *current* token?





Source: F. Chollet: *Deep Learning with Python*, Fig. 15.4, 15.5

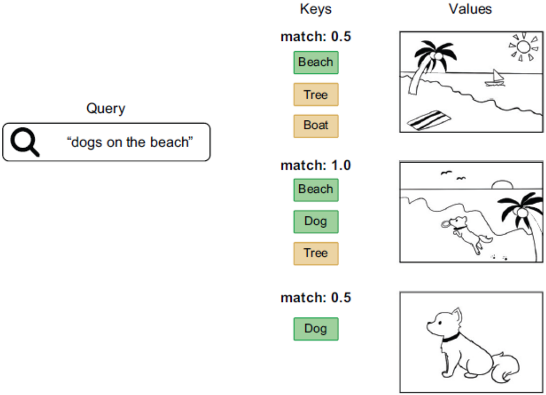
More general form:

- `output = sum(source * attention_score(target, source))`

Attention

Terminology (borrowed from search engines):

- `output = sum(value * attention_score(query, key))`



Source: F. Chollet: *Deep Learning with Python*, Fig. 15.6

Attention

Terminology (borrowed from search engines):

- `output = sum(value * attention_score(query, key))`

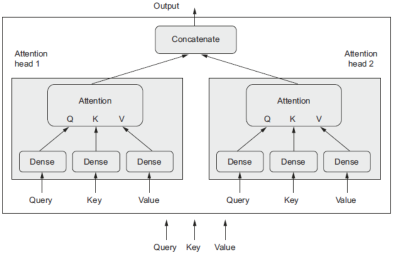
Common variants:

- **Classification:** value = key = query = input text
- **Machine translation:** value = key = source text, query = generated target text
- **Summarization:** value = key = source text, query = summary prompt
- **Text generation:** value = key = previous tokens, query = current position
- **Recommender systems:** value = key = user history, query = currently recommended item

Multi-head Attention

Each head focuses on a different aspect of the input:

- it applies its own linear projection to the same token embedding, creating different views and attention patterns

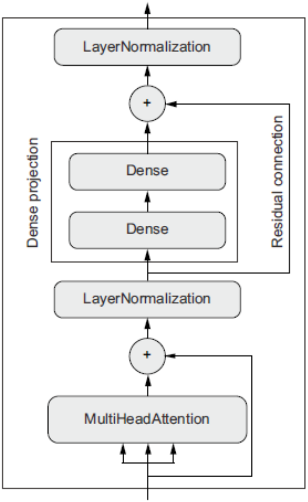


Source: F. Chollet: *Deep Learning with Python*, Fig. 11.8

In Keras:

- **self.attention = layers.MultiHeadAttention(num_heads=4, key_dim=256)**

Transformer Encoder



For classification tasks, the encoder alone is enough:

- multi-head attention
- layer normalization
- MLP
- residual connections

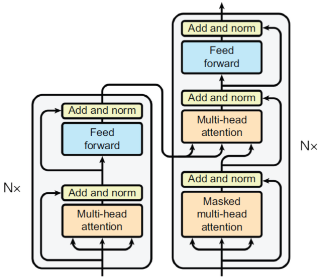
Transformer Encoder

Practical example: IMDB sentiment classification

- **Simple Transformer Encoder trained from scratch**
  - uses word embeddings + positional encodings
  - performs **worse than RNNs** (accuracy 0.81–0.87)
  - reason: Transformers require **much larger datasets** to avoid overfitting (IMDB has only 20k training samples)
- **Pretrained Transformer Encoder (finetuned)**
  - **RoBERTa-base**
  - fine-tuning on IMDB dramatically improves accuracy → **0.93–0.97**, even on short sequences
  - benefits from:
    - large pretraining corpora (billions of tokens)
    - contextual embeddings
    - subword tokenization (WordPiece / BPE)
- **Takeaway:**
  - Transformers trained from scratch perform poorly on small datasets.
  - Pretrained Transformers dominate NLP tasks today.

Transformer

- For many-to-one tasks, the encoder alone is enough
- For machine translation or text generation, we need both the **encoder** and the **decoder** — together they form the transformer.



Source: F. Chollet: *Deep Learning with Python*, Fig. 15.8

Transformer

What does the decoder do?

- Generates the output sequence token by token (autoregressive).
- Uses **masked self-attention** to look at previously generated tokens.
- Uses **encoder–decoder attention** to focus on the relevant parts of the source sentence.
- Combines both sources of information to predict the next token.

Decoder attention

- **Query (Q):** current decoder state — what I need to generate now.
- **Key (K):** encoder outputs — which parts of the source sentence might be relevant.
- **Value (V):** encoder outputs — the actual information returned after weighting.

Transformers – Key Takeaways

- **Attention** allows the model to focus on what matters.
- **Self-attention** creates context-aware representations.
- **Multi-head attention** = multiple perspectives on the same input.
- **Transformer** = encoder + decoder (for sequence-to-sequence tasks).
- Modern variants:
  - encoder-only (e.g., BERT)
  - decoder-only (e.g., GPT)
  - encoder–decoder (e.g., T5)

Evolution of Language Models (LMs)

From word embeddings → Transformers → Large Language Models

- **2013–2016: Static word embeddings** Word2Vec, GloVe
  - one vector per word, no context
- **2017: Transformer architecture** (Vaswani et al.)
  - multi-head self-attention, positional encoding
  - scalable stacking of encoder/decoder blocks
- **2018: Contextual LMs – Encoder-based BERT**, RoBERTa, ALBERT, DeBERTa
  - bidirectional encoding, masked-language modeling (MLM)
  - excellent for classification, NER (named entity recognition),
  - contextual embedding, not generative
- **2018–2020: Decoder-based generative models GPT, GPT-2, GPT-3**
  - autoregressive LM (predict next token)
  - extremely scalable (billions of parameters)
- **2022+: Instruction-tuned LLMs** GPT-3.5/4, PaLM, LLaMA, Mistral

What Makes a Modern LLM?

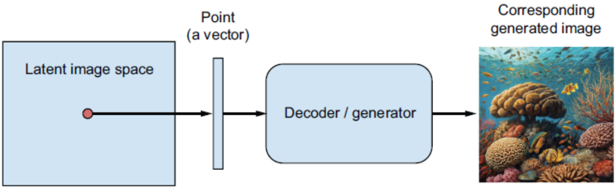
Key ingredients:

- **Transformer Decoder blocks** : self-attention + feedforward layers + residual connections + layer normalization
  usually 12–80 stacked layers
- **Subword vocabulary**: BPE / SentencePiece (30k–100k tokens)
- **Massive pretraining**
  - trillions of tokens (web, books, code)
  - objective: next-token prediction (autoregressive LM)
- **Instruction tuning** :
  - supervised finetuning (SFT)
  - reinforcement learning from human feedback (RLHF)
- **Scalability laws**: more data + more parameters = better performance

Introduction to Generative models for Image generation

Image generation Key idea

- **encoder** develops a low-dimensional latent space
- **generator (decoder)** maps each point from the latent space to a ,,valid" image
- **interpolation**

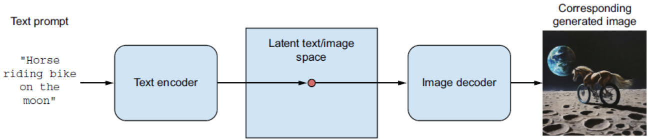


Source: F. Chollet: *Deep Learning with Python*, Fig. 17.1

Introduction to Generative models

Language-guided Image generation: text-to-image models

- **encoder** maps the space of prompts to low-dimensional latent space



Source: F. Chollet: *Deep Learning with Python*, Fig. 17.2

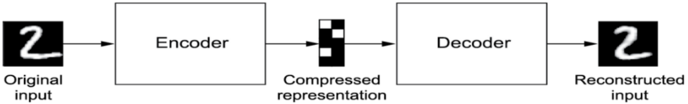
Introduction to Generative models

Main families of generative models for images

- Variational autoencoders (VAEs, 2014)
- Generative adversarial networks (GANs, 2014)
- Autoregressive models (2016) - Generate image pixel-by-pixel or patch-by-patch
- Diffusion models (2020-today, Stable Diffusion, DALL·E 2, Imagen)

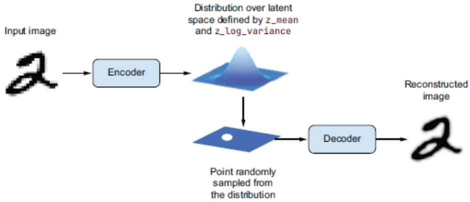
Variational Autoencoders

Autoencoders



Variational Autoencoders

- Probabilistic latent space

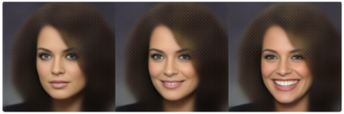


F. Chollet: Deep Learning with Python, Fig. 17.3, 17.4

Variational Autoencoders

Variational Autoencoders

- Probabilistic latent space
- Capable of generating new data samples or creating smooth interpolations
- Blurrier outputs (due to likelihood training)

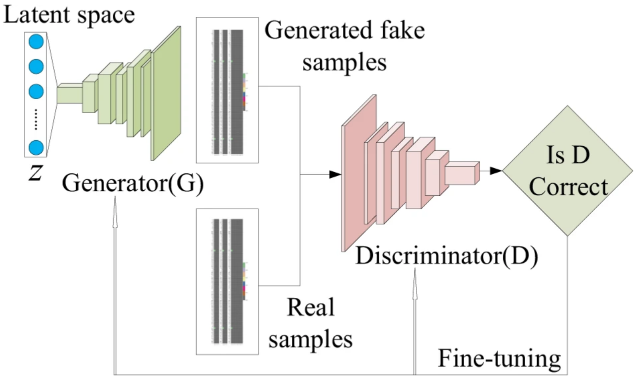


F. Chollet: Deep learning v jazyku Python, obr. 5.7

Source: Tom White, "Sampling Generative Networks, "https://arxiv.org/pdf/1609.04468

Generative adversarial networks (GANs)

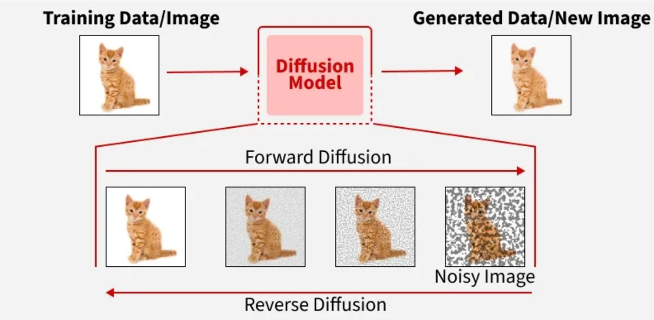
- Two-player minimax game (generator vs. discriminator)
- Hard to train (instability, mode collapse)



Dan, Y., Zhao, Y., Li, X. et al. Generative adversarial networks (GAN) based efficient sampling of chemical composition space for inverse design of inorganic materials. (2020)

Diffusion models

- Start with an image → gradually add noise → get pure noise
- Gradually remove noise → generate highly realistic images
- State-of-the-art for image generation
- Stable training, excellent detail and composition



https://www.geeksforgeeks.org/artificial-intelligence/what-are-diffusion-models/