L1: Intro to NLP

Documents & Knowledge

Document: any source of NLP text (books, articles, web). Text is **unstructured** and **ambiguous**. **Knowledge Representation**: structured, precise, actionable, domain-specific data a computer can use. **NLP**: converts unstructured documents → structured knowledge. Documents: humans read slowly, get tired, can't remember all. Knowledge: computers use quickly, don't tire, answer questions fast.

Definition: NLP: subfield of **linguistics**, **CS**, and **AI** for analyzing/understanding human language and processing large amounts of natural language data.

Motivation: Industry adoption (analyze customer feedback → improve sales). Accessibility (voice interfaces for people with disabilities). Career demand high, especially with LLMs. Research growth: NLP papers among highest since 2017.

AI / ML / DL / NLP

- **AI**: umbrella – making computers act like humans
- **ML**: subset of AI – learns from data (classification, clustering, forecasting, association, anomaly detection)
- **DL**: subset of ML – deep neural networks
- **NLP**: intersects ML and DL

ML data: table where rows = instances, columns = features, last column = class/decision. All values must be numeric. Text must be converted to numbers before ML/DL.

History: 1950s: translation machines. 1960s: ELIZA chatbot (**rule-based**, not DL). 1970s: statistical models. 2013: Word2Vec. 2017: "Attention Is All You Need" – Transformer. Post-2017: LLMs (GPT, etc.). **Transformer variants**: Encoder-only (BERT), Decoder-only (GPT), Encoder-Decoder (T5).

Turing Test: Judge asks questions to a human & computer. If judge can't distinguish → **pass**. Proposed by Alan Turing, 1950. Key insight: understanding and generating language ≈ intelligence.

- **NLU only**: sentiment analysis, text classification (spam, priority, category, language detection, document/topic)
- **NLG only**: speech-to-text, auto report generation (e.g., medical reports from patient data)
- **NLU+NLG**: translation, summarization, QA, chatbots

Speech pipeline: Voice → Signal Processing → Acoustic Model (signals → words) → NLP (words → text). Chatbots are **bidirectional**: speech→text (input) + text→speech (output). **Dialogue management** tracks conversation state and controls flow.

Sentiment analysis: requires **deeper** semantic understanding than text classification. **Text summarization**: one of the **hardest** NLP tasks (coherence, redundancy, complexity, language-specific). **QA**: **Extractive** = locate/extract answer from text. **Generative** = generate answer from scratch. **Text classif. vs Sentiment**: classif. = shallow/keyword-based; sentiment = deep semantic understanding of emotion/attitude/trends. **LLMs** (e.g., ChatGPT): text generation + **reinforcement learning**, built on Transformer architecture.

Challenges in NLP

1. **Ambiguity**: lexical ("bank"), attachment ("saw man with telescope"), coreference ("she")
2. **Sparsity / Zipf's Law**: $f(w) \propto 1/r$ (frequency **inversely** proportional to rank); rare words are **not** outliers in NLP – can carry critical meaning (e.g., "fuzzy logic" appears once but is important). >1/3 of words occur only once
3. **Variation**: lexical ("gave the book to Tom" = "gave Tom the book"), geographical (regional dialects), social (professor vs friend), stylistic, generational, cross-linguistic (English NLP ≠ French NLP)
4. **Common knowledge**: humans share implicit knowledge computers lack ("a man with a dog" vs "a dog with a man" – both valid, humans know which is natural; "Earth is round")
5. Volume, accents/slang, computation, security/privacy

Approaches to NLP: Heuristic/Rule-based (regex) → Machine Learning → Deep Learning (RNN, LSTM) → Transformers (2017+). Old techniques still useful for edge cases.**NLTK**: learning/prototyping. **spaCy**: industrial-strength. **Hugging Face**: pre-trained models & Transformers. **scikit-learn**: preprocessing, vectorization, ML models.

L2: Regex, Pipeline, Preprocessing

Regular Expressions

`.`	any single char	`a.` → `ab`, `a1`
`?`	0 or 1 of preceding	`colou?r` → color/colour
`+`	1 or more of preceding	`o+h!` → oh!, ooh!
`*`	0 or more of preceding	`oo*h!` → oh!, ooh!, oooh!
`[]`	any one char inside	`[abc]` → a, b, or c
`[^]`	negation (not in set)	`[^A-Z]` → a, 1, !
`^`	start of string	`^[A-Z]` → uppercase at start
`$`	end of string	`world$` → "Hello world"
`{}`	specific repetitions	`a{2,3}` → aa, aaa
`\`	escape metachar	`\.` → literal dot
`|`	alternation (or)	`aa|b` → aa or b
`()`	grouping (exact sequence)	`(abc)` → exact "abc"

Key: `[abc]` = any **one** of a,b,c. `(abc)` = exact sequence a,b,c **in order**. **Ranges**: `[a-z]`, `[A-Z]`, `[0-9]`. **Negation**: `^` has **dual meaning** – inside `[]`: negation (`[^A-Z]` = NOT uppercase); outside `[]`: anchor (`^[A-Z]` = uppercase at **start**). Examples: `colou?r` → color/colour. `beg.n` → begin/begun. `\.$` → literal period at end.

Char classes: `\s` whitespace, `\w` word char (slides: `[A-Za-z]`; Python `re`: `[A-Za-z0-9_]`), `\d` digit, `\b` word boundary. Uppercase = negation.

Python `re` Library

`re.match` – match at beginning only. `re.search` – first match anywhere. `re.findall` – list of all non-overlapping matches. `re.sub` – replace all matches. `re.compile` – compile into reusable pattern object. `re.split` – split string at pattern matches. `re.finditer` – iterator of match objects. Match objects: `.group()` – matched text. `.span()` – (start, end) indices.

Regex Applications in NLP

Text cleaning, tokenization, info retrieval, simple sentiment (count good vs bad words), language detection (pattern matching for language-specific chars).

NLP Dev Lifecycle

1. Understand problem & requirements 2. Data collection (large + relevant) 3. Text cleaning 4. Preprocessing 5. Feature extraction 6. Modeling 7. Evaluation 8. Deployment 9. Monitoring. Non-linear: loop back from eval/monitoring. Step 1: decide if NLP is even the right approach. Step 9: watch for **model drift** (new terminology/patterns over time degrade performance). **Garbage in, garbage out.** Always explore data first (like EDA in ML).

Preprocessing Pipeline

Raw doc → **Tokenization** → **Noise Removal** → **Normalization** → clean tokens. **Why**: improve model performance, reduce dimensionality, standardize input from different sources (PDFs, web, text files).

Building Blocks of Language

Phonemes (44 sounds) → Morphemes/Lexemes (smallest meaningful unit, e.g., "untangling" = "un" + "tangle" + "ing") → Syntax (grammar rules) → Context. **Corpus**: collection of docs. **Vocabulary**: set of unique words. **Word**: unit of language separated by spaces/punctuation.

Tokenization

Divide text into **tokens** (not just words). Punctuation & contractions become separate tokens ("can't" → "can" + "'t"). Semicolons/punctuation are tokens but **not** words. Whitespace is **not** the only split criterion. NLTK: `word_tokenize()`, `sent_tokenize()`, `regexp_tokenize()` (custom regex). **Vocabulary** ≠ tokens: vocab = set of **unique** words; tokens = all units including duplicates.

Noise Removal

Remove: numbers, punctuation, **stop words** (179 in NLTK), URLs, HTML tags, handles, hashtags (keep/remove depends on task). Keep emojis for sentiment (replace with word equivalents). Lowercase conversion. **Compound words** ("New York", "machine learning"): keep as one token using dictionary. Stop words ↔ **Zipf's law**: most frequent words carry **least** meaning.

Code: `re.sub(r"\d+", " ", text)` rm numbers. Remove punct: `text.translate(str.maketrans("","",string.punctuation))` `[w for w in tokens if w not in stop_words]` filter stop words.

Normalization

Stemming: rule-based suffix stripping. Fast, may produce invalid words ("studies"→"studi", "helps"→"help"). Porter, Snowball (multilingual), Lancaster (aggressive). Useful for: classification, clustering, search engines/info retrieval.

Lemmatization: vocabulary + morphological analysis → **lemma** (root word). Always valid words ("studies"→"study", "better"→"good", "am/is/are"→"be"). Slower but more accurate. NLTK WordNetLemmatizer, spaCy. **spaCy: lemmatization only (no stemming)**. Use lemmatization when **accuracy > speed**.

Skip normalization for: poetry analysis, morphological analysis, social media analysis (variations convey emotion/attitude).

POS Tagging & NER

POS: identifies noun, verb, adj, etc. `pos_tag(tokens)`. Tags: NN (noun), NNS (plural), VB (verb), VBZ (3rd person), JJ (adj), RB (adv), DT (det), IN (prep), PRP (pronoun). `nltk.help.upenn_tagset()` for full list. Why: syntactic/semantic analysis, improves downstream tasks (NER, parsing, translation). **NER**: identifies people, places, orgs, phone numbers, emails. `ne_chunk(pos_tag(...))`. Entity types: PERSON, GPE (geo-political), ORGANIZATION. Why: info extraction, search/indexing, identify which entity a sentiment targets.

L3: Feature Representation & Similarity

Feature Engineering

Convert text → numerical table for ML. Equally important as preprocessing. On Kaggle, a single new feature can win a contest. **Data representations**: images = matrix (pixel intensity), speech = waveform (amplitude), text = **vectors**. **Evolution**: Frequency-based (statistical) → Word Embedding (NN) → Transformer-based (LLMs).

Vector Space Model

Each word/doc = **vector** of numbers. Similar words → nearby vectors. **Vectorization**: encoding text as integers → **feature vectors**. All techniques produce vectors; key difference = how well values reflect **semantic meaning**.

Norm: $\|\vec{v}\| = \sqrt{v_1^2 + \cdots + v_n^2}$ **Dot product**: $\vec{a} \cdot \vec{b} = \sum a_i b_i$

Early Attempt: Linguistic Vectors

Experts manually answered questions per word ("Is it male?", "Is it living?", "Can it talk?") → binary vectors. First attempt at vector space. Not scalable, language dependent.

One Hot Encoding

Binary vector: 1 if token exists, 0 otherwise. Dim = vocab size (300K+ English words). Steps: tokenize → build vocab → assign unique IDs → binary vector. Pros: reversible, interpretable, preserves position. Cons: high dimensionality, sparse, no semantic relation between words. Ex: "This is an example" → This=[1,0,0,0], is=[0,1,0,0], an=[0,0,1,0], example=[0,0,0,1].

Bag of Words (BoW)

Vector of **word frequencies**. Ignores word order. Improved over OHE. "child makes dog happy" = "dog makes child happy" (same BoW!). "John is quicker than Mary" = "Mary is quicker than John". Pros: simple, language independent, works for text classif/info retrieval. `cv=CountVectorizer(); X=cv.fit_transform(corpus)` – builds BoW matrix from corpus. Defaults: `lowercase=True`, `token_pattern` requires 2+ chars (drops single-char words like "I"). `cv.get_feature_names_out()` – returns vocab sorted alphabetically. `X.toarray()` – converts sparse matrix to dense.

Bag of N-grams

Count frequencies of n consecutive words. Introduces **local context**. n=1: unigram (=BoW), n=2: bigram, n=3: trigram. Larger n = more context but **much** larger feature space. Choice of n by trial and error. `ngram_range=(2,2)` – bigrams only; `(1,2)` – uni+bigrams. Ex: "I love NLP" bigrams: [I love, love NLP]. "I am learning NLP" trigram: "am learning NLP".

TF-IDF

$$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)$$

$$\text{TF}(t,d) = \frac{\text{count of } t \text{ in } d}{\text{total terms in } d} \quad \text{IDF}(t) = \log_{10}\left(\frac{\text{total docs}}{\text{docs with } t}\right)$$

Rare words → **higher** weight (high IDF). Common words → **lower** weight (low IDF). `tfidf=TfidfVectorizer(); X=tfidf.fit_transform(docs)` – builds TF-IDF weighted matrix from docs. Originated from **info retrieval** (before NLP). Not suited for **small corpora** (IDF misleads). Zeros carry info (word absent = attribute). Ex: 1000 docs, 100-word doc, "Trump" appears 5×. TF= 5/100 = 0.05. If in 50 docs: IDF= log(1000/50) = 1.3, TF-IDF= 0.065. If in only 5 docs: IDF= log(1000/5) = 2.3, TF-IDF= 0.115 (rarer → higher weight). No polysemy: "bank" (financial) and "bank" (river) get **same** vector.

CountVec vs TF-IDF: CountVec = whole numbers (raw counts); TF-IDF = real numbers (weighted). TF-IDF weights "milk" (rare) higher than "hot" (common) → smarter similarity.

All frequency methods: no semantics, sparse, high dim, OOV problem. Still useful when frequency matters more than semantics (trade-off accuracy vs computation).

Text Similarity Metrics

Applications: plagiarism detection, search engines, machine translation, info retrieval, text classification. Used as **benchmark** to evaluate/compare representation techniques.

Jaccard: set overlap $J(A,B) = |A \cap B|/|A \cup B|$. **Hamming**: differing positions in equal-length strings. **Levenshtein**: min single-char edits (insert, delete, sub). "kitten"→"sitting" = 3 (k→s, e→i, insert g). "intention"→"execution" = 5. **Euclidean**: $d = \sqrt{\sum(a_i - b_i)^2}$. Measures straight-line **distance** (magnitude only, not direction). **Cosine Similarity** (most used in NLP): measures **angle** between vectors (magnitude + direction). $\cos(\vec{A},\vec{B}) = \frac{\vec{A}\cdot\vec{B}}{\|\vec{A}\|\,\|\vec{B}\|}$

cos =1: identical. cos =0: no similarity. cos = −1: opposite. Ex: $\vec{A}$=[1, 2, 1], $\vec{B}$=[2, 1, 1]. $\vec{A}\cdot\vec{B} = 5$, $\|\vec{A}\| = \|\vec{B}\| = \sqrt{6}$. cos = 5/6 ≈ 0.833 (high). Ex: "I love NLP" $\vec{a}$=[1, 1, 0, 1] vs "I love you" $\vec{b}$=[1, 1, 1, 0]. cos = 2/3 ≈ 0.667.

As benchmark: represent → compute cosine → high score = good representation. TF-IDF + cosine > BoW + cosine (weights rare words higher). `cosine_similarity(matrix)` from `sklearn.metrics.pairwise` – pairwise cosine between all row vectors.

Choosing Representation

Simple classif/info retrieval → TF-IDF. Semantic understanding → word embeddings. Deep contextual → Transformers. Always consider accuracy vs computation trade-off.

Ethical concerns: false similarity/dissimilarity (e.g., plagiarism detection errors), term/topic bias (popular topics overweighted, minority topics underrepresented), language bias (English dominance on internet).

L4: Word Embedding

Motivation: Count-based: sparse, high-dim, no semantics, OOV. Goal: dense vectors with semantic meaning. **Distributional hypothesis**: words in similar contexts are similar. **Semantic field**: words covering a domain (restaurant → food, menu, waiter, chef). Embedding must capture word semantics + contextual relations.

WordNet: Lexical database (Princeton). Synsets (synonyms), gloss (definition). **Hypernym**: broader term ("animal"→"dog"). **Hyponym**: specific term ("dog"→"animal"). Also: meronym (part-of), antonym (opposite). Synset: {car, automobile, motorcar}. Holonym (whole): car→wheel. Troponym: "run"→"move". Entailment: "snore"→"sleep". Used for **query expansion**: query → keywords → WordNet synonyms → expanded search. Not computational, static, English-only. `wordnet.synsets("car")` – all synsets for word. `.definition()` – gloss. `.examples()` – usage examples.

Word Embedding: Maps words to dense vectors: $f : V \to \mathbb{R}^D$. Prediction-based (not counting). Dim D is hyperparameter (50–300), does **not** need to equal vocab size. Higher dim = more features but more computation. **Self-supervised**: data provides its own labels, no human annotation. Dense: every position has a meaningful value (no zeros). **Analogy**: $\vec{king} - \vec{man} + \vec{woman} \approx \vec{queen}$. Also: walk:walking::swim:swimming, Spain:Madrid::Italy:Rome. **Transfer learning**: pre-trained models reused across tasks without retraining.

Word2Vec (Google, 2013)

CBOW: context words → predict center word. Window $2K{+}1$ (odd). Ex: "The cat sat on the mat" K=2: [The,cat,on,the]→"sat". **Skip-gram**: center word → predict context words. Boundary: padding for missing positions. Architecture: one-hot $\times W$ selects row from $V \times D$ weight matrix → softmax. After training, W = embedding matrix. **SGNS**: Skip-gram + negative sampling. Positive pairs (target, context) vs. random negative pairs. Uses **logistic regression** to distinguish them. Learned weights = embeddings.

	CBOW	**Skip-gram**
Input	Context	Target
Output	Target	Context
Speed	Faster	Slower
Best for	Freq. words	Rare words

Gensim: `Word2Vec(sg=0/1, vector_size, window, negative)` – trains Word2Vec. `sg=0`=**CBOW**, `sg=1`=**Skip-gram**. `vector_size`=dim (default 100). `window`=context size. `negative`=# neg samples. Pre-trained: `word2vec-google-news-300` (3M words, 100B tokens).

`model.similarity("king","queen")` – cosine similarity between two words. `model.most_similar("cup", topn=3)` – 3 nearest words by cosine. Analogy: `most_similar(positive= ["king","woman"], negative=["man"])` – word closest to king – man + woman. Train own: `Word2Vec(sentences=corpus, vector_size=100, window=5, sg=1, negative=5)` – trains skip-gram w/ neg sampling. `model.wv["word"]` – get embedding vector.

As features: tokenize → lookup embedding per token → average all vectors → single feature vector → classifier.

GloVe (Stanford, 2014): Prediction + **global co-occurrence matrix**. X_{ij} = how many times words i, j appear together in window across entire corpus. $\vec{x_i} \cdot \vec{x_j} \approx \log(X_{ij})$. $J = \sum f(X_{ij})(\vec{w_i}^T \vec{w_j} + b_i + b_j - \log X_{ij})^2$ $f(X_{ij})$: weighting fn, prevents frequent co-occurrences from dominating. Steps: 1. build co-occurrence matrix, 2. minimize J, 3. vectors capture local + global patterns. Pre-trained: Wiki+Gigaword 6B, CommonCrawl 42B/840B, Twitter 27B. Dims: 50–300. Comes as **text file** (word + vector values per line).

FastText (Facebook, 2016)

Character n-grams (3–6 chars). Word vector = sum of n-gram vectors. "cities" → ⟨ci, cit, iti, tie, ies, es⟩. **Handles OOV**: shared n-grams give approx vectors for unseen words. Captures morphological info. Multi-language. Same researcher (Mikolov) created Word2Vec and FastText. Two-step: center word = n-grams + word vector; context words = word vector only. `FastText(sentences, vector_size=100, window=5, sg=1)` – trains FastText. OOV: `model.wv["unseenword"]` – approx vector via char n-grams.

OOV Handling: 1. Default zero vector (spaCy: `doc[0].has_vector` → False). 2. Synonym fallback (WordNet). 3. Train on own corpus. Limitation: "I love school" vs "I hate school" = high similarity (doesn't capture opposition).

Evaluation: **Intrinsic**: word similarity, word analogy. **Extrinsic**: downstream task performance. Limitations: Limited context (window only), bias in training data ("doctor" closer to "man" than "woman"), **static**: one fixed vector per word regardless of context (no polysemy), dim choice, resource intensive. → **Contextualized representations** (ELMo, BERT, ULMFiT): word vectors change based on surrounding context.

L5: Language Models

Data Collection

Social media APIs (X/Twitter, free tier ∼11.5K tweets/day), web scraping (BeautifulSoup, lxml, html5lib), PDF files (PyPDF2). BeautifulSoup: `requests.get(url)` → `BeautifulSoup(resp.text, "html.parser")` → `.find_all("p")` → tokenize → save.

Probability Review

$P(A,B,C) = P(A) \times P(B) \times P(C)$ (independent). $P(X|Y) = P(X,Y)/P(Y)$ (conditional). In corpus: $P(w_2|w_1) = C(w_1 w_2)/C(w_1)$. **Chain rule**: $P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|X_1, ..., X_{i-1})$

Language Model

Predicts next word: $P(w_{t+1}|w_1, ..., w_t)$. Prob distribution over vocab; highest prob = prediction. Must come from model's vocabulary (seen during training). Everyday: WhatsApp text completion, Google autocomplete, email suggestions.

N-gram LM (Statistical)

$P(w_n|w_1, ..., w_{n-1}) = \frac{C(w_1,...,w_n)}{C(w_1,...,w_{n-1})}$. **Markov assumption**: w_t depends only on previous $n{-}1$ words. Ex: "students opened their __" (trigram). C("students opened their")=1000, C(…"books")=400, C(…"exams")=100. P(books)= 400/1000 = 0.4 (winner). **Text generation**: seed → predict next (highest prob) → append → repeat. Often **not coherent** (small context). Training: corpus → sliding window (L→R, **order matters**) → extract n-grams → count frequencies → probability table = trained model. Limitations: choosing n (larger = more context but exponential computation), **sparsity** (unseen → zero prob), limited context. **Backoff**: fall back to smaller n (4-gram → trigram → bigram).

Neural Network LM

Pipeline: one-hot → **embedding lookup** → concat/avg → **hidden layer** → **softmax** → prediction. Still needs **fixed window** (fixed input size = fixed # of features). **Averaging problem**: "food was good, not bad" vs "food was bad, not good" → same avg vector, opposite meaning. Treats words as **independent**.

Deep Neural Networks

UAT (Cybenko 1989): NN approximates any continuous function. **Hinton 2006**: revived deep NNs. Traditional ML hits **performance ceiling** with more data; DNNs **keep improving**. CNN: images. **RNN: sequences** (stateful).

RNN Language Model

Stateful: $h_t = f(W_x x_t + W_h h_{t-1} + b)$. vs Standard NN: words fed **one at a time**, hidden state at **each** step, info propagates step 1→T. Variable length. Shared weights across time steps. **Self-supervised**: next word is known target (no annotation). Loss = **neg log prob** of correct next word. $L_{total} = \sum_{t=1}^{T} L_t$. **BPTT**: backprop through time. Weight update: $w_{new} = w_{old} - \eta \cdot \nabla L$. **Learning rate** η: too low → slow; too high → diverge. **Hallucination**: model can't find correct data, produces unrealistic output.

Vanishing Gradient

$\frac{\partial L}{\partial h_1} = \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_4}{\partial h_3} \cdot \frac{\partial L}{\partial h_4}$. Small derivatives multiply → gradient **vanishes** (→ 0, not explodes). Early tokens don't learn; tokens near **end** learn more. Cannot capture **long-range dependencies**. Ex: "tried to print her **ticket**…long passage…finally printed her __" → can't remember "ticket".

LSTM (Hochreiter & Schmidhuber, 1997)

Adds **cell state** c_t (long-term) + hidden state h_t (short-term). Three **gates** (sigmoid: near 1 = keep, near 0 = forget): **Forget**: $f_t = \sigma(W_f[h_{t-1}, x_t]{+}b_f)$. **Input**: $i_t = \sigma(W_i[h_{t-1}, x_t]{+}b_i)$. **Candidate**: $\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t]{+}b_c)$. **Cell update**: $c_t = \underbrace{f_t \odot c_{t-1}}_{\text{filter old}} + \underbrace{i_t \odot \tilde{c}_t}_{\text{add new}}$. **Output**: $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$. **Hidden**: $h_t = o_t \odot \tanh(c_t)$. c_t = **long-term**. h_t = **short-term**. Sigmoid: 0–1 (gate). Tanh: −1 to 1. ⊙: element-wise multiply.

Perplexity

$\text{PPL}(W) = P(w_1, ..., w_N)^{-1/N}$. Inverse prob normalized by N words. **Low** = predicts well. **High** = confused/surprised.

GRU

Simplified LSTM: combines forget+input into single **update gate**, merges cell+hidden state. 2 gates (reset, update). Fewer params, faster.

Keras Code

`Sequential([Embedding(vocab, dim), LSTM(128), Dense(vocab, 'softmax')])` – LSTM LM: embed → LSTM → softmax over vocab. GRU: replace `LSTM(128)` with `GRU(128)`. Multi-layer: `LSTM(128, return_sequences=True)` – output full sequence to next layer; then `LSTM(64)` – last hidden state only.

Evolution

N-gram → NN LM → RNN → LSTM/GRU → Transformer.

L6: RNN, LSTM, Seq2Seq, Attention

RNN Recap

$h_t = f(W_x x_t + W_h h_{t-1} + b)$, f = activation (typically **tanh**). Sequential input, hidden state propagates info. Variable length. Self-supervised (next word = known target). Training: feed tokens one-by-one L→R. Each step: one-hot → embedding → hidden state → loss. vs Standard NN: all words at once, one shared hidden layer, no info propagation, fixed window.

BPTT & Vanishing Gradient

Gradient: derivative of loss w.r.t. weights (rate of change of loss). $W_{new} = W_{old} - \alpha \cdot \nabla L$ (α = learning rate). Chain rule: $\frac{\partial L}{\partial h_1} = \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_4}{\partial h_3} \cdot \frac{\partial L}{\partial h_4}$. Small derivatives multiply → gradient **vanishes** (→ 0, not explodes). Early tokens don't learn; tokens near **end** learn more. → **LSTM** adds cell state + gates.

LSTM Gates Summary

vs RNN: LSTM input adds c_{t-1}; output adds updated c_t. Gates = vectors (0–1 via σ), dynamically control info flow.

Forget	σ	what to discard from c_{t-1}
Input	σ	what new info to add
Candidate	tanh	proposed new content
Cell update	–	$f_t \odot c_{t-1}$ (filter old) $+ i_t \odot \tilde{c}_t$ (add new)
Output	σ	what to output
Hidden	tanh	$o_t \odot \tanh(c_t)$

Sequence Problem Types

1-to-1	Single→Single	Image classif.
1-to-many	Single→Seq	Image caption
Many-to-1	Seq→Single	Sentiment, stock
Many-to-many	Seq→Seq	Translation

Bidirectional LSTM

Standard RNN/LSTM = forward only (past context). Bi-LSTM adds **both** directions. Forward ($\overrightarrow{h_t}$) + backward ($\overleftarrow{h_t}$) LSTMs. $h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$. **Separate weights** (not shared). Concatenated h_t passed to next layers. Ex: "terribly exciting" – forward-only misreads "terribly" as negative; Bi-LSTM sees "exciting" too.

Multi-layer RNN/LSTM

Stack layers: hidden states from layer i → inputs to layer $i{+}1$. Learns increasingly abstract representations. Keras: `LSTM(128, return_sequences=True)` (pass full sequence to next layer) then `LSTM(64)` (final layer).

Seq2Seq (Encoder-Decoder)

Used for: machine translation, summarization, chatbots. Input/output can be **different lengths**. **Encoder**: reads entire input L→R → fixed-length **context vector** (final hidden state). **Decoder**: context vector as initial hidden state, starts with START token → output sequence token by token. Each generated word fed back as input to next step. Conditional LM: $P(y_1, ..., y_T|x_1, ..., x_S)$. Training: end-to-end backprop, $J = \frac{1}{T}\sum_{t=1}^{T} J_t$ (avg neg log prob). Testing: argmax at each decoder step.

Bottleneck Problem

Entire input compressed to single fixed-length vector → information loss for long sequences. Longer input = harder to compress = worse performance.

Attention Mechanism

Solution to bottleneck. **Attention** = weighted average over inputs. Decoder looks at **all** encoder hidden states at each step: 1. **Dot product**: decoder state vs. each encoder state → scores. 2. **Softmax**: scores → attention weights (**probability distribution** over encoder states). 3. **Weighted sum**: weights × encoder states → attention output. 4. **Concat**: attention output + decoder state → prediction.

Benefits: variable-length, long-range dependencies, focus on relevant parts. Ex: "il a m' entarté" → decoder step 1 attends mostly to "il"→"he", step 3 to "m'"→"me".

GRU vs LSTM

GRU: **update gate** z_t (combines forget+input), **reset gate** r_t (controls how much past to forget). Merges cell state into hidden state.

	LSTM	**GRU**
Gates	3	2 (reset, update)
States	$h_t + c_t$	h_t only
Params	More	Fewer
Speed	Slower	Faster

Path to Transformers

N-gram → NN LM → RNN → LSTM/GRU → Seq2Seq + Attention → **Transformer** (2017, "Attention Is All You Need"). **No recurrence** (no RNN/LSTM), relies **entirely on self-attention**. Handles long-range dependencies. **LSTM limitations**: sequential processing (slow, can't parallelize), still struggles with very long sequences, resource intensive → Transformer overcomes all. Modern LLMs don't use RNN/LSTM directly, but Transformer **built on top of** these ideas. Each step improved on previous limitations.

Pre-trained embeddings (GloVe) as embedding layer → more semantic, more coherent output. More **epochs** → better text generation. Keras: `Sequential([Embedding(..), LSTM(128), Dense(vocab, 'softmax')])` – same LSTM LM architecture. `Bidirectional(LSTM(n))` – wraps LSTM to process input in both directions. **LSTM for classification**: also works for sentiment, not just generation. `loss='categorical_crossentropy'` – multi-class loss; `optimizer='adam'` – adaptive learning rate optimizer.