Machine Translation – Advanced Topics

Vincent Vandeghinste

2026

Contents

Chapter 1

Introduction to Machine Translation

1.1 What is Machine Translation?

Machine Translation (MT) is commonly defined as *the use of computers to translate text from one natural language into another*. This definition is used, for example, by the European Association for Machine Translation (European Association for Machine Translation, 2024).

MT is one of the oldest ambitions in computer science. The history of the field is discussed in Section 1.5. Although fully automatic, high-quality translation remains an unsolved problem, current MT systems can produce useful output for many domains and tasks.

MT is now embedded in web applications, computer-aided translation (CAT) tools, multilingual information retrieval, and speech-to-speech translation. Studies show that neural MT can substantially increase translator productivity, as shown in Figure 1.1 (Plitt and Masselot, 2010; CrossLang, 2025). This course is mainly focused on neural MT.

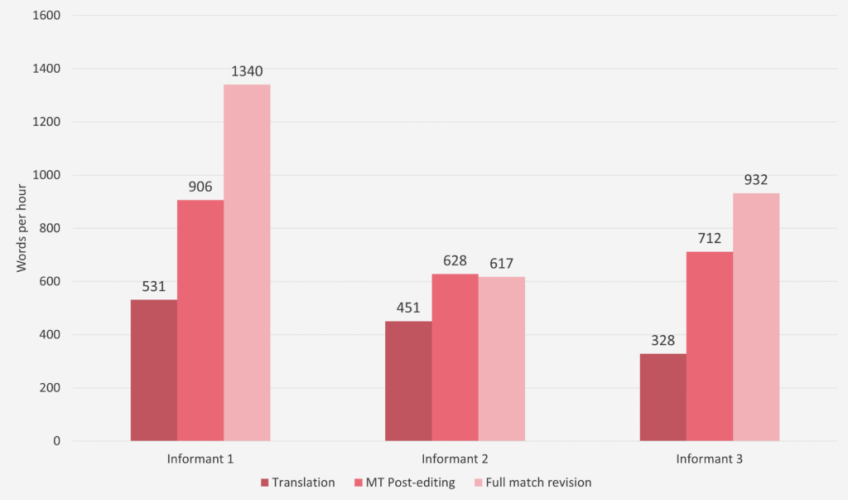


Figure 1.1: Productivity increase from three translators when using MT (CrossLang, 2025).

1.2 Goal of the course Machine Translation: Advanced Topics

The course *Machine Translation: Advanced Topics* consists of two main sections:

1. The *Machine Translation* part, for which these are the course notes, and which will introduce the MT research field and the current state-of-the-art in MT. The course will allow to gain hands-on experience in the experimental paradigm in MT research and see the effects on translation quality of the major breakthroughs that have happened in the field of Neural Machine Translation, since around 2013.

2. The *Post-editing* part, in which students will take a critical look at the post-editing process, again through hands-on experience. This part is out of the scope of the current notes.

1.3 Machine Translation in the Context of Natural Language Processing and Artificial Intelligence

As shown in Figure 1.2, Machine Translation (MT) is a core task within the broader field of Natural Language Processing (NLP), which itself is situated within Artificial Intelligence (AI).

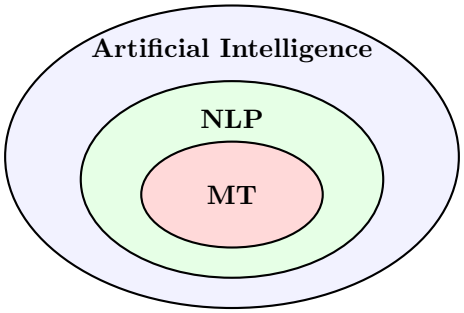


Figure 1.2: MT as a subfield of Natural Language Processing and Artificial Intelligence.

1.3.1 Artificial Intelligence (AI)

Artificial Intelligence is the study and development of computational systems that can perform tasks generally associated with human intelligence. These tasks include reasoning, decision making, perception, planning, learning, and language understanding.

In the context of MT, AI provides the conceptual and technical foundations for building models that can learn patterns from data, represent linguistic structures, and make predictions about translation quality. Modern MT belongs to the subfield of *machine learning*, where systems improve their performance based on experience.

1.3.2 Natural Language Processing (NLP)

Natural Language Processing is the scientific and engineering discipline that focuses on enabling computers to analyse, generate, and interact using human language. NLP covers a broad range of tasks such as part-of-speech tagging, syntactic parsing, sentiment analysis, question answering, information extraction, and also machine translation. Because natural language is richly ambiguous, variable, and context-dependent, NLP must integrate knowledge about linguistic structure with statistical or neural models that handle uncertainty.

NLP is an inherently interdisciplinary area, drawing on insights and methods from computer science, engineering, linguistics, and logic and mathematics (Jurafsky and Martin, 2025), as illustrated in Figure 1.3.

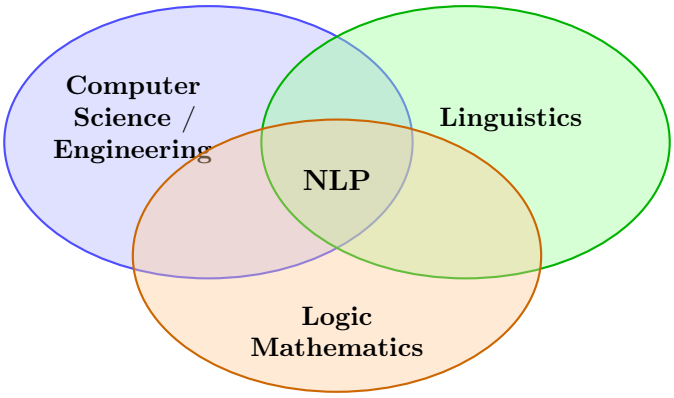


Figure 1.3: NLP as an interdisciplinary field.

A classic example of linguistic ambiguity is the sentence *I made her duck*. This short sentence allows multiple interpretations involving lexical, syntactic, and semantic ambiguity. MT systems must therefore learn to resolve or manage these ambiguities based on contextual cues, training data, and probabilistic modelling. When uncertainty cannot be resolved deterministically, modern systems consider multiple possible translations and rank them using probabilistic scoring functions (Koehn, 2020; Sennrich, 2018).

Illustrative ambiguity

I made her duck.

This short sentence has several distinct readings:

1. **Culinary reading (create + dish)**
   "I cooked a duck for her."
   make = 'prepare (food)', *duck* = noun (the animal / dish), *her* = indirect object ('for her').

2. **Culinary reading (create + possession)**
   "I cooked the duck that belongs to her."
   make = 'prepare', *duck* = noun, *her* = possessive determiner ('her duck').

3. **Causative reading (cause to move)**
   "I caused her to lower her head (to duck)."
   make = causative verb, *duck* = verb, *her* = direct object ('I made her [duck].').

4. **Transformative reading (turn into a duck)**
   "I turned her into a duck."
   make = 'cause to become', *duck* = noun (new state), *her* = direct object ('I made her [a duck].').

5. **Magical Creative reading**
   "I created the duck which belongs to her" *make* = 'create', *duck* = noun, *her* = possessive deteriminer ('her duck').

1.3.2.1 Computer Science and Engineering

Computer Science and Engineering contribute the algorithmic, mathematical, and software foundations needed to design and implement MT systems. This includes: efficient data structures, ma-

chine learning algorithms, neural network architectures, optimisation methods, high-performance computing, distributed training environments, and engineering best practices for building real-world translation applications. In addition, knowledge from software engineering is essential for designing robust, scalable, and maintainable MT pipelines that can process large amounts of multilingual data.

1.3.2.2 Linguistics

Linguistics provides the theoretical models and descriptive frameworks needed to understand how languages are structured and how meaning is conveyed. MT systems benefit from linguistic knowledge in areas such as morphology, syntax, semantics, pragmatics, discourse structure, and typology. Understanding linguistic variation, cross-linguistic generalisations, and translation equivalences helps inform model design, error analysis, and data selection. Even in highly data-driven modelling paradigms like neural MT, linguistic insights remain crucial for interpreting system behaviour and addressing persistent challenges such as word sense disambiguation, anaphora resolution, and structural divergence between languages.

Word sense disambiguation concerns selecting the intended meaning of a polysemous word from context, anaphora resolution involves identifying which earlier expression a pronoun or referring element points to, and structural divergence refers to systematic differences in how languages encode the same meaning through syntax, morphology, or word order.

In addition, insights from Translation Studies complement linguistic theory by foregrounding translation as a communicative and socio-cultural activity rather than a purely formal mapping between languages. Concepts such as equivalence, adequacy versus fluency, translation shifts, norms, and functionalist approaches (e.g. Skopos theory) help frame what constitutes a "good" translation in different contexts. Skopos theory (Vermeer, 1989) is a functionalist approach to translation which holds that the primary criterion for translation decisions is the *purpose (skopos)* of the target text in its intended communicative context, rather than strict formal or semantic equivalence to the source text.

For MT, these perspectives are particularly relevant for data selection and annotation, evaluation practices, and the interpretation of system errors, highlighting that translation quality is context-dependent and cannot be fully captured by surface-level correspondences alone.

1.3.2.3 Logic and Mathematics

Logic contributes formal methods for representing meaning, reasoning with symbolic structures, and specifying the correctness of algorithms. Historically, logical formalisms played a central role in early rule-based MT systems. Although contemporary neural MT relies heavily on statistical methods, logical foundations remain important in areas such as semantic representation, inference, and the specification of constraints in hybrid systems. Mathematical foundations—including probability theory, linear algebra, statistics, and optimisation—underlie all modern NLP and MT models, especially neural networks and probabilistic models.

Together, these disciplines shape the field of NLP, within which Machine Translation evolved as one of the most ambitious and challenging tasks. MT integrates nearly all aspects of linguistic and computational modelling: from handling ambiguity to managing uncertainty, from processing large-scale text corpora to learning complex mappings between languages.

1.4 The Machine Translation Research Community

Machine Translation is embedded within a vibrant and collaborative research ecosystem supported by several international associations and conferences. These organisations promote open scientific communication across the global MT landscape, define evaluation practices, coordinate shared tasks and curate benchmark datasets.

1.4.1 Professional Associations and Regional Conferences

The field is anchored by three major regional MT associations, each of which organises its own regional conference and contributes to the biennial MT Summit.

EAMT (European Association for Machine Translation)[1] supports MT research and development across Europe, with a strong emphasis on collaboration between academia, industry, and public institutions.

Its annual conference, the **EAMT Conference**, is the primary European MT forum, featuring research papers, system demonstrations, and practitioner-focused tracks. It happens every year, unless when the MT Summit takes place in Europe.

AMTA (Association for Machine Translation in the Americas)[2] promotes MT activities in North and South America, bridging academic research with commercial and governmental applications.

The **AMTA Conference** is held biennially and includes dedicated research, industry, and government tracks, reflecting the region's diverse MT use cases and applied focus.

AAMT (Asia-Pacific Association for Machine Translation)[3] serves the linguistically diverse Asia-Pacific region, supporting MT for a typologically rich set of languages and scripts.

Its main event, the **AAMT Conference** (formerly hosted under the name *Asia-Pacific MT Conference*), provides a venue for emerging MT technologies and region-specific challenges such as translation for low-resource and agglutinative languages.

Together, these three associations form the **International Association for Machine Translation (IAMT)**[4] and jointly oversee the **MT Summit**,[5] which rotates across the three regions. MT Summit is the flagship MT conference, combining a scientific track, user and industry sessions, and workshops aimed at both researchers and professional translators.

1.4.2 Major International Conferences and Shared Tasks

Beyond the regional conferences and the MT Summit, several global venues play a central role in shaping MT research:

WMT (Workshop/Conference on Machine Translation)[6] is a large yearly MT event, known primarily for its shared tasks, including news translation, low-resource translation, metrics evaluation, quality estimation, and data filtering. WMT has become the standard benchmarking venue for MT systems.

[4] `https://eamt.org/international-association-for-machine-translation/`

[5] The website of the MT Summit 2025 edition can be found here: `https://mtsummit2025.unige.ch/`.

IWSLT (International Workshop on Spoken Language Translation)[7] focuses on speech-to-text and speech-to-speech translation, IWSLT provides open benchmarks for lecture translation, simultaneous translation, and end-to-end multimodal systems.

ACL (Association for Computational Linguistics)[8] organizes several generic NLP conferences, where a substantial part of presentations are about machine translation.

LREC (Language Resources and Evaluation Conference)[9] is a biennial conference mainly focussing on datasets and evaluation practices, often with descriptions of datasets for smaller languages.

Shared tasks are community-driven evaluation campaigns in which multiple research groups address the same well-defined problem using common datasets, training conditions, and evaluation protocols, allowing systems to be compared in a fair and reproducible manner. In machine translation, shared tasks typically provide standardized training, development, and test sets, specify evaluation metrics, and culminate in a coordinated evaluation and workshop where results are reported and analyzed. Beyond benchmarking system performance, shared tasks play a crucial role in advancing the field by identifying open challenges, encouraging methodological rigor, fostering resource sharing, and creating reference points that guide future research and system development.

WMT and IWSLT host shared tasks that provide publicly available datasets, open access tools and standardized evaluation protocols, ensuring reproducibility and enabling meaningful comparison across MT systems.

1.4.3 Open Access and Knowledge Dissemination

The MT community is strongly committed to open scientific communication. The **ACL Anthology**[10] plays a central role by providing open access to thousands of peer-reviewed papers in NLP and MT. Most MT-related venues publish their proceedings through the Anthology, promoting long-term preservation of research outputs, transparent access to methods, datasets, and evaluation results, and a shared foundation for benchmarking and cumulative scientific progress.

The combination of active regional associations, global evaluation campaigns, and open-access publication has facilitated rapid advances in Machine Translation, supporting both foundational research and industrial innovation.

1.5 MT History: From Early Rule-Based Systems to Neural and Large Language Models

This section presents a historically organised overview of machine translation, structured around the major paradigms that have shaped the field. Rather than treating paradigms and history separately, we introduce each approach at the point where it emerged, highlighting how successive paradigms responded to the limitations of earlier ones.

[10] `https://aclanthology.org/`

1.5.1 Pre-digital Concepts and Early Visions

Ideas of universal languages and interlingual representations predate digital computing by centuries, notably in the work of Descartes (1596-1650) and Wilkins (1614–1672). Early mechanical aids to translation appeared in the 1930s, such as Artsrouni's *mechanical brain* for bilingual dictionary lookup (Figure 1.4). These systems anticipated later rule-based MT by emphasising explicit lexical correspondences and structured representations.

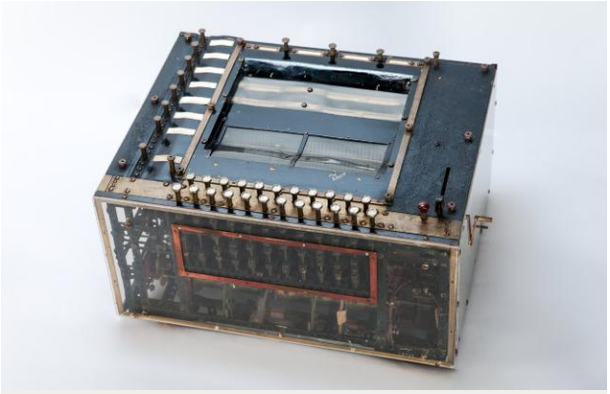


Figure 1.4: Le Cerveau Méchanique

A decisive conceptual step was taken in Weaver (1949)'s memorandum, which framed translation as a decoding problem inspired by wartime cryptography. This idea strongly influenced the first generation of digital MT systems.

1.5.2 1950s–1970s: Rule-Based Machine Translation

The first operational MT system was demonstrated in the 1954 Georgetown–IBM experiment (Hutchins, 2004), using hand-crafted rules and dictionaries in a restricted domain. Despite exaggerated early optimism, this experiment marked the beginning of **Rule-Based Machine Translation (RBMT)**, which dominated MT research for several decades.

RBMT systems model translation as a sequence of deterministic transformations based on explicitly encoded linguistic knowledge, including morphological analysers, grammars, bilingual lexicons, and transfer rules.

A central conceptual framework for understanding RBMT architectures is the *Vauquois triangle* (Vauquois, 1968) (Figure 1.5) on which RBMT approaches can be mapped.

The Vauquois triangle represents the source language on the left and the target language on the right. Moving upward in the triangle corresponds to increasing levels of linguistic abstraction, reducing the gap that needs to be bridged between the source and the target language. RBMT systems can be positioned at different heights in this space.

1.5.2.1 Direct translation (surface level)

At the bottom of the Vauquois triangle, translation operates directly on the lexical/surface level. Rules are applied to words or short sequences without constructing an explicit syntactic or semantic representation. This is called *direct translation*.

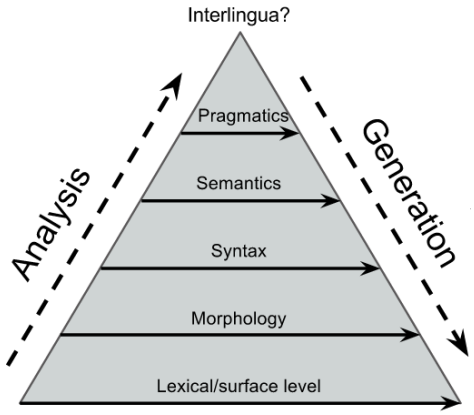


Figure 1.5: The Vauquois triangle, illustrating translation strategies from direct word-level translation to deep interlingual representations By Mihkelkohava - Own work, CC BY-SA 4.0, `https://commons.wikimedia.org/w/index.php?curid=46284016`

Direct Translation Example

Step	**Operation**
Input	`she eats apples`
Morphological analysis	`eats = eat + 3rd person singular`
	`apples = apple + plural`
Lexical lookup	`she → zij`
	`eat → eten`
	`apple → appel`
Morphological generation	`eten + 3rd person singular = eet`
	`appel + plural = appels`
Output	`zij eet appels`

No syntactic structure is built. Translation proceeds via word-by-word substitution and simple morphological rules. Agreement and word order are assumed to be identical.

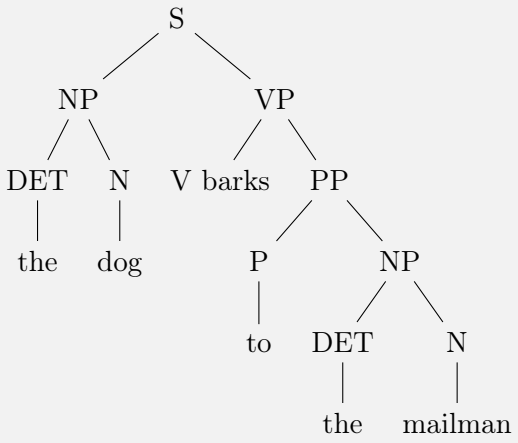
1.5.2.2 Transfer-based translation (syntactic and semantic levels)

In the middle of the Vauquois triangle, RBMT systems analyse the source sentence into an intermediate linguistic representation and apply explicit transfer rules to convert it into a target-oriented structure, distinguishing three phases: *analysis, transfer, generation.*

Analysis In the analysis phase, the source sentence is syntactically (and maybe also semantically) parsed: through a grammar, the sentence is analysed and a tree structure analysis (or equivalent) is the result of this process.

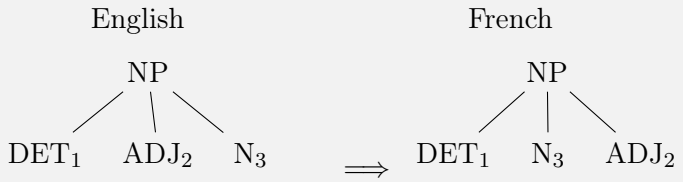
Example Syntactic Analysis

Here is an example syntactic analysis for the sentence `the dog barks to the mailman`, in the form of a *constituent tree.*



Syntactic transfer rules as a synchronous tree substitution grammar Many systematic syntactic divergences between languages are more naturally captured using a *synchronous tree substitution grammar* (STSG), also called *tree transduction grammars* in which pairs of subtrees are synchronised via variable correspondences between their frontier nodes. They model patterns of how to do transfer in source language analysis subtrees.

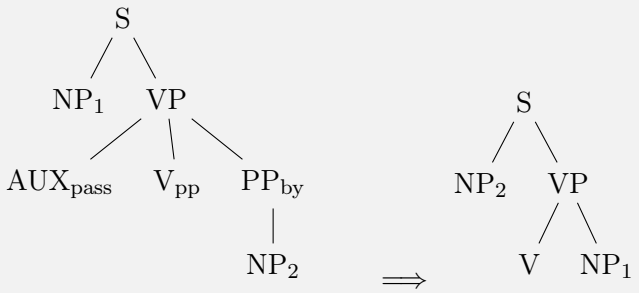
Exampel transfer rule between English and French



This rule shows that in English noun phrases (NPs) that contain a determiner, adjective and noun, when transferred to French, the adjective and noun change places. (This is often true, but there are many exceptions).

Example:
$\texttt{DET}_1\texttt{[the] ADJ}_2\texttt{[blue] N}_3\texttt{[sky]} \rightarrow \texttt{DET}_1\texttt{[le] N}_3\texttt{[ciel] ADJ}_2\texttt{[bleu]}$

A more complex transfer example: Passive ↔ active



This transfer rule shows how to convert from a passive structure in the source language to an active structure in the target language.

Example:
$\texttt{NP}_1\texttt{[de man] AUX}_{pass}\texttt{[wordt] V}_{pp}\texttt{[gebeten] PP}_{by}\texttt{[door NP}_2\texttt{[de hond]]} \rightarrow$
$\texttt{NP}_2\texttt{[the dog] V[bites] NP}_1\texttt{[the man]}$

These synchronous tree pairs make explicit how constituent structure and grammatical roles can be systematically reconfigured across languages.

Semantic transfer At a higher point in the triangle, transfer rules operate on predicate–argument structures rather than surface syntax.

Semantic Transfer Rules

Predicate mapping :

```
eat(x,y) → eten(x,y)
```

Argument Reversal :

```
miss(x,y) → manquer(y,x)
```

Role preservation :

```
AGENT(eat,x) → AGENT(eten,x)
PATIENT(eat,y) → PATIENT(eten,y)
```

Such rules abstract away from word order and allow the system to handle alternations such as passivisation or argument reordering more robustly.

Semantic Transfer Example

Source Analysis :

```
eat(
  AGENT = she,
  PATIENT = apples,
  TENSE = present
)
```

Transfer rules :

```
eat(x,y)  → eten(x,y)
she       → zij
apples    → appel[PL]
```

Target generation :

```
S → NP V NP
NP(AGENT) = zij
V = eet
NP(PATIENT)[PL] = appels
```

Generation The final phase in transfer rule-based MT is the generation phase, in which the output of the transfer process (which can be a syntax tree) needs to be turned into a surface sentence. This may require resolving target language specific issues such as agreement between subjects and participles.

1.5.2.3 Interlingua-based translation (deep semantic level)

At the top of the Vauquois triangle, translation proceeds via a language-independent semantic representation known as an *interlingua*. The source language is mapped to meaning, and the target language is generated from that meaning.

Interlingua Example

- **Interlingua representation**:

```
EVENT: EAT
  AGENT: FEMALE_PERSON
  PATIENT: APPLE (PLURAL)
  TENSE: PRESENT
```

- **Source analysis**:
  `she eats apples`
  ⇒ `EAT(AGENT=FEMALE_PERSON, PATIENT=APPLE[PL], TENSE=PRESENT)`
- **Target generation Dutch**:
  `EAT(x,y,PRESENT)`
  ⇒ `NP(x) + V_eat[PRESENT,3SG] + NP(y)`

  `NP(FEMALE_PERSON) → zij`
  `V(EAT, PRESENT, 3SG) → eet`
  `NP(APPLE, PLURAL) → appels`

  At this level, the source language no longer matters. The same interlingua could generate translations in any target language for which generation rules exist.

Interlingua-based systems maximise reuse across languages but require extremely rich and carefully designed semantic representations.

1.5.2.4 Typical RBMT components

Regardless of their position in the triangle, most RBMT systems include some combination of the following components:

- **Morphological analysis and generation**, handling inflection, agreement, and derivation;
- **Syntactic analysis**, using phrase-structure or dependency grammars;
- **Lexical transfer**, mapping source-language lexemes to target-language equivalents, often enriched with part-of-speech or sense information;
- **Structural transfer rules**, describing systematic cross-linguistic differences in word order, argument structure, or grammatical realisation.

1.5.2.5 Strengths and limitations

RBMT systems can be highly precise and linguistically interpretable, and they perform well in controlled domains with stable syntax and terminology. However, they require immense manual effort to develop and maintain, are brittle in the face of unexpected input, and scale poorly to many language pairs or new domains. These limitations motivated the shift toward data-driven approaches, such as statistical and neural MT, discussed in the following sections and chapters.

A notable exception in terms of scalability is offered by *interlingua-based* systems. Because translation proceeds via a single, language-independent meaning representation, an interlingua architecture avoids the need to design separate transfer components for each language pair. In a system with N languages, this reduces the number of required components from N^2 pairwise transfer modules to N analysis and generation modules. In principle, this makes interlingua approaches particularly attractive for multilingual machine translation and for systems targeting many low-resource language pairs.

In practice, however, the theoretical scalability advantages of interlingua systems are offset by the difficulty of defining a sufficiently expressive, language-neutral semantic representation. Capturing subtle lexical, syntactic, and pragmatic distinctions across diverse languages has proven extremely challenging, which has limited the practical adoption of interlingua-based RBMT systems despite their conceptual appeal.

A further fundamental limitation of RBMT systems is *error percolation* in pipeline architectures. RBMT typically decomposes translation into sequential stages of analysis, transfer, and generation. Errors introduced at early stages, such as incorrect morphological analysis or syntactic parsing, tend to propagate unchanged through subsequent components and often become amplified in later stages. Because later modules generally assume well-formed input from earlier ones, they have limited ability to recover from upstream mistakes. This lack of robustness to partial or noisy analyses contrasts sharply with later data-driven approaches, which model uncertainty more explicitly and can often produce reasonable output even when internal representations are imperfect.

1.5.3 1980s: Transfer-Based Systems and the Emergence of Example-Based MT

Despite the funding setbacks following the ALPAC report (Pierce et al., 1966), RBMT research continued in restricted domains, notably in systems such as SYSTRAN and MÉTÉO. During this period, transfer-based architectures became increasingly prominent.

At the same time, dissatisfaction with large hand-written rule sets motivated new data-oriented ideas. Nagao (1984) proposed **Example-Based Machine Translation (EBMT)**, introducing the idea of *translation by analogy*. Rather than encoding linguistic knowledge explicitly, EBMT systems retrieve similar sentences from a parallel corpus and adapt their translations to new inputs.

EBMT is the first approach toward MT which is *data-driven*, i.e. in which information for translation is not hand-made but automatically induced from the data.

The basic operations are:

1. retrieve example sentences similar to the input,
2. identify correspondences between example and input phrases,
3. recombine translated fragments.

Example-based MT

Assume the parallel corpus contains:

Example 1:
EN: `I like green apples.`
NL: `Ik hou van groene appels`

Example 2:
EN: `I like red pears.`
NL: `Ik hou van rode peren`

To translate the new sentence: `I like green pears.` the EBMT system performs:

- match `I like` ↔ `Ik hou van`,
- find the translation of `green` from Example 1,
- find the translation of `pears` from Example 2,
- recombine: `Ik hou van groene peren`

Despite its conceptual appeal, EBMT suffers from several inherent limitations. Its performance depends heavily on the coverage and diversity of the example corpus: if no sufficiently similar sentence is found, the system fails to produce a translation. Even when examples exist, identifying reliable correspondences between source and target fragments is non-trivial and often requires heuristic alignment procedures. EBMT also struggles to generalise beyond observed patterns, making it brittle in the face of lexical variation, syntactic reordering, or longer sentences. As a result, EBMT systems scale poorly to open-domain translation and were gradually superseded by more robust statistical approaches that offered better generalisation from data.

1.5.4 1990s–2000s: Statistical Machine Translation

The increasing availability of parallel corpora and computational resources in the 1990s enabled a decisive shift toward probabilistic, data-driven methods. Building on earlier work in word alignment and language modelling, **Statistical Machine Translation (SMT)** replaced hand-crafted rules with probability distributions learned from data (Brown et al., 1993).

SMT systems combine translation models, language models, and additional features within a log-linear framework, selecting translations via approximate decoding.

Translation was formulated as a probabilistic inference problem, in which the goal is to select the target sentence that maximises the probability given a source sentence. Canonical SMT systems are built around several core components, each addressing a different aspect of the translation task.

Word alignment Word alignment models estimate which words or short segments in a source sentence correspond to which words in the target sentence. Alignment algorithms and models such as the IBM models, applied on large parallel corpora, provide the statistical foundation for learning translation units from data. Although alignments are imperfect and often ambiguous, they enable SMT systems to move beyond word-for-word substitution.

Phrase-based translation Phrase-based SMT extends word-level translation by allowing variable length sequences of words, known as *phrases*, to be translated as units. Phrase tables are extracted automatically from aligned corpora and associate each source phrase with one or more target phrases and corresponding probabilities. This mechanism captures local reordering and

idiomatic expressions more effectively than purely word-based models.

Phrase-based SMT became popular due to the free and open source toolkit Moses (Koehn et al., 2007), which made it relatively easy to build your own MT system, given that you had the data to train it on.

Language modelling Fluency in SMT is enforced through an explicit target-side language model, typically an n-gram model trained on large monolingual corpora. The language model assigns higher probability to well-formed target-language sequences and penalises ungrammatical or unnatural word orders. This separation between translation adequacy and target-language fluency is a defining characteristic of SMT architectures.

Log-linear model and decoding The different knowledge sources in SMT (translation probabilities, language models, and additional features such as reordering or length penalties) are combined in a log-linear framework, meaning that each feature is weighted, and decoding (i.e. actual translation) amounts to searching for the target sentence with the highest overall score. This provides flexibility and extensibility, but also leads to complex decoding algorithms and extensive parameter tuning.

An SMT decoder constructs candidate translations by combining phrase translations and ranks them using a weighted scoring function, selecting the highest-scoring output. The language model ensures that the most fluent option is preferred over less natural alternatives.

SMT example

For the sentence: `She eats apples.`
a simplified **phrase table** might contain entries such as:

Source phrase	Target phrase	Probability
`she`	`zij`	0.92
`eats`	`eet`	0.87
`apples`	`appels`	0.94
`apples`	`de appels`	0.04
`eats apples`	`eet appels`	0.76

During **decoding**, the SMT system incrementally constructs partial translation hypotheses and maintains a beam of the highest-scoring candidates. At each step, hypotheses are expanded by applying compatible phrase translations and updating their scores using a weighted combination of translation-model probabilities, language-model probabilities, and other features.

Step 0: Initial state

- Covered source words: none
- Hypothesis: empty
- Score: 0.0

Step 1: Translating `she`
From the phrase table, the decoder can apply:

- `she` → `zij`

This yields the following beam:

Partial translation	Covered source	Score (log)
`zij`	`she`	−0.8

The score reflects the translation probability for `she` and an initial language-model contribution.

Step 2: Expanding with `eats`
The hypothesis `zij` can be expanded by translating `eats`:

- `eats` → `eet`

Resulting beam:

Partial translation	Covered source	Score (log)
`zij eet`	`she eats`	−1.2
`zij`	`she`	−2.0

The hypothesis `zij eet` is preferred because it both covers more source words and forms a fluent target-language sequence according to the language model.
The hypothesis `zij` receives an additional penalty because it does not extend coverage of the source sentence at this step; SMT decoders penalise hypotheses that fail to make progress.

Step 3: Translating `apples`
From the state `zij eet`, the decoder can apply either a single-word or a multi-word phrase translation:

- `apples` → `appels`
- `apples` → `de appels`
- `eats apples` → `eet appels`

This yields competing hypotheses:

Partial translation	Covered source	Score (log)
`zij eet appels`	`she eats apples`	−1.5
`zij eet de appels`	`she eats apples`	−2.3
`zij eet`	`she eats`	−2.6

Although `de appels` is a possible translation of `apples`, the language model assigns a higher probability to the fluent sequence `eet appels` than to `eet de appels`. As a result, the hypothesis `zij eet appels` receives the best overall score.

After all source words are covered, the highest-scoring hypothesis in the beam is selected as the final translation.

SMT represented a major advance in translation quality and scalability compared to rule-based systems, but its reliance on local phrase translations and n-gram language models limited its ability to capture long-distance dependencies and global sentence coherence. These limitations ultimately motivated the transition toward neural machine translation.

1.5.5 2010s: Neural Machine Translation

Neural MT emerged from advances in neural language modelling (Bengio et al., 2003) and sequence-to-sequence learning (Sutskever et al., 2014). The introduction of attention (Bahdanau et al., 2015) and the Transformer architecture (Vaswani et al., 2017) led to substantial improvements in fluency, long-distance dependencies, and overall translation quality.

By the mid-2010s, neural MT had largely replaced SMT in both research and industry. Subword segmentation, back-translation, and multilingual training further extended neural MT to lower-resource settings.

From chapter 4 onward, we dive into the details of neural MT systems.

1.5.6 2020s–present: Large Multilingual Models and LLMs

Recent years have seen the rise of large multilingual encoder–decoder models (e.g. mBART, mT5, NLLB) as well as general-purpose decoder-only language models such as GPT-style systems, LLaMA, and Qwen.

These models embed translation within broader pretrained architectures, enabling zero-shot and few-shot translation, multilingual transfer, and tighter integration with other language understanding tasks. At the same time, they raise new challenges regarding controllability, consistency, and evaluation.

Today's MT landscape therefore consists of specialised encoder–decoder systems (Chapter **??**) and with general-purpose LLMs (Chapter 8).

1.6 The Experimental Paradigm in Machine Translation

Machine Translation research follows a well-established *experimental paradigm* that is shared with much of empirical natural language processing. Rather than asking whether a single system is "good", MT research aims to understand whether a *specific design choice* leads to a measurable improvement in translation quality, efficiency, or robustness. To this end, experiments are designed to isolate the effect of one controlled change, while keeping all other aspects of the system identical.

This paradigm allows researchers to draw causal conclusions: if two systems differ in only one respect, any consistent difference in performance can be attributed to that change rather than to uncontrolled variation.

1.6.1 Minimal Pairs of Experimental Conditions

Central to MT experimentation is the comparison of *minimal pairs of experimental conditions*. Each condition corresponds to a complete MT system, and conditions are constructed so that they differ in exactly one controlled variable. Typical manipulations include changing a single architectural parameter, modifying the data representation, or adjusting the amount or type of training data.

For example, a researcher might compare a Transformer model with six encoder layers to an otherwise identical model with twelve encoder layers, or contrast a word-level system with a subword-based system (see Section 6.1 using byte-pair encoding (BPE). Crucially, all other components—tokenisation, preprocessing, optimisation settings, training budget, and evaluation procedure—must remain unchanged. In this sense, the notion of a minimal pair is precisely the practical mechanism by which all non-target variables are held constant.

Maintaining minimal differences between conditions ensures that observed differences in translation quality scores, training stability, or inference speed can be reliably attributed to the manipulated variable rather than to confounding factors. This requirement is particularly important for neural MT systems, which are sensitive to seemingly minor implementation choices.

In practice, enforcing minimal pairs requires strict experimental discipline. Random seeds must be fixed so that differences in parameter initialisation do not obscure the effect of the experimental manipulation. Tokenisation and preprocessing pipelines must be identical, as even small differences in normalisation or segmentation can affect vocabulary statistics and training dynamics. Similarly,

optimisation settings such as learning rate, batch size, scheduler, and total training budget must be kept constant across conditions.

Careful construction of minimal pairs is what distinguishes a scientifically valid MT experiment from an informal system comparison.

1.6.2 Controlled Data Splits

Another cornerstone of the experimental paradigm is the strict separation of data into training, validation, and test sets. The training set is used to learn model parameters, the validation set guides hyperparameter tuning and early stopping, and the test set is reserved exclusively for final evaluation.

While common split ratios such as 80/10/10 or 90/5/5 are often reported, the exact proportions are less important than consistency. All experimental conditions must use *exactly the same* data partitions. If one system is trained or evaluated on different sentences, the comparison is no longer valid.

Key principle

The test set is used *once*, after all design decisions have been made.

Violating this principle, by tuning on the test set or by changing the validation data across experiments, leads to overly optimistic results and undermines the scientific validity of the evaluation.

An implementation for data splitting is given in Section 2.8. Some commonly used datasets (such as the shared task datasets) come with a predefined split which should be reused in order to allow result comparison.

1.6.3 Evaluation and Significance Testing

After training, systems are evaluated on the held-out test set using automatic evaluation metrics such as BLEU, chrF, COMET, or BLEURT.

In modern MT research, two additional principles are widely accepted. First, results should be reported using multiple metrics, as each metric captures different aspects of translation quality. Second, observed differences between systems should be tested for statistical significance. Techniques such as bootstrap resampling are commonly used to determine whether a reported improvement is likely to reflect a genuine difference rather than random variation.

Without significance testing, small numerical differences can easily be overinterpreted, especially when comparing strong neural models.

Both the evaluation metrics and significance testing will be extensively discussed in Chapter 3.

1.6.4 Interpretation and Reporting

The final stage of the experimental paradigm concerns interpretation and reporting. Researchers analyse not only whether a change improved translation quality, but also whether it affected training cost, inference speed, or robustness across language pairs and domains. Quantitative evaluation is often complemented by qualitative analysis, in which example translations are inspected to identify systematic improvements or new types of errors.

A complete experimental report includes detailed information about the data, preprocessing steps, model architecture, hyperparameters, training procedure, evaluation metrics, and known limitations. This level of transparency is essential for reproducibility and allows other researchers to verify, replicate, and build upon the reported findings.

In sum, the MT experimental paradigm relies on careful control, minimal differences between systems, transparent evaluation, and replicability. These principles enable researchers to make scientifically grounded claims about what improves machine translation, and why.

1.7 The State of MT Today

The current machine translation landscape is marked by rapid technological change, expanding multilingual coverage, and deep integration into professional translation and localisation workflows. Industrial surveys and market reports (e.g. Intento, 2025) consistently document steady gains in translation quality, particularly for high-resource language pairs, alongside a growing acceptance of MT output as a standard component of computer-assisted translation (CAT) environments. Rather than replacing human translators, MT has become an infrastructure technology that supports, accelerates, and reshapes professional translation practice.

1.7.1 Coexistence of NMT and LLM-based MT

Despite the emergence of large multilingual language models, **Neural Machine Translation (NMT)** remains the backbone of most production-grade MT systems. Carefully trained and domain-adapted NMT models—typically based on the Transformer architecture—continue to deliver strong performance in sentence-level translation tasks, especially in settings where quality, predictability, and low latency are critical. In many industrial applications, dedicated NMT systems still outperform general-purpose LLMs when evaluated on standard MT benchmarks.

At the same time, LLMs have broadened the MT technology space by introducing new capabilities that extend beyond traditional sentence-level translation. LLM-based systems tend to be more robust when faced with noisy, informal, or non-canonical input, and they are better equipped to incorporate wider contextual information, including document-level structure or explicit user instructions. Their ability to adapt to new domains, registers, or styles through prompting or parameter-efficient fine-tuning has made them attractive in scenarios where flexibility and interaction are more important than strict consistency or throughput.

As a result, hybrid architectures are increasingly common. In such setups, a conventional NMT engine provides fast and reliable base translations, while an LLM is employed for downstream tasks such as post-editing, stylistic rewriting, terminology enforcement, explanation, or quality estimation. This division of labour reflects a pragmatic response to the complementary strengths and weaknesses of the two paradigms.

1.7.2 Integration in Professional Workflows

Machine translation is now firmly embedded in professional translation pipelines. Modern CAT tools routinely integrate MT engines that operate in real time and adapt dynamically to user feedback. Beyond sentence-level translation, many systems support document-level processing, enabling improved consistency, terminology control, and coherence across longer texts.

Professional users increasingly expect MT systems to be customisable. This may take the form

of user-defined glossaries, domain-adapted models, or interactive prompting mechanisms that allow translators to influence tone, register, or style. In parallel, automated quality estimation tools are used to predict the reliability of MT output and to prioritise segments for human post-editing. As a consequence, the boundary between fully automatic translation and human translation has become increasingly fluid, with growing attention to usability, cognitive effort, and human–machine interaction.

1.7.3 Evaluation Trends

Evaluation practices in MT are also evolving. While surface-level metrics such as BLEU or chrF remain widely used, both research and industry are placing greater emphasis on evaluation beyond surface-level metrics and isolated sentences. Document-level assessment now plays a more prominent role, capturing phenomena such as coherence, consistency, and discourse structure that are invisible to sentence-based metrics.

In addition, there is increasing interest in context- and task-aware evaluation, including measures of factual consistency, adequacy for downstream tasks, and risk in sensitive application domains. LLM-based evaluators and reference-free metrics are being explored as scalable alternatives to traditional reference-based evaluation, though their reliability remains an active research question. Despite these advances, human evaluation continues to be indispensable, particularly for low-resource languages and high-stakes domains where automatic metrics may fail to capture critical errors.

1.7.4 Challenges and Risks

Despite impressive progress, contemporary MT systems face a number of unresolved challenges. Ensuring controllability remains difficult, especially when users require strict adherence to terminology, style guides, or constrained output formats. Reliability is another concern, as both NMT and LLM-based systems may produce omissions, hallucinations, or overconfident translations that are hard to detect automatically.

Linguistic coverage also remains uneven, with many low-resource and typologically diverse languages still underrepresented in training data and benchmarks. At the same time, broader issues of data governance have become central, including questions of privacy, copyright, and transparency about training data sources. Finally, concerns about bias and fairness in multilingual models have prompted renewed attention to the social and ethical dimensions of MT technology.

A clear understanding of the principles, assumptions, and limitations of both NMT and LLM-based approaches is therefore essential for the responsible deployment of machine translation in professional, societal, and multilingual communication contexts.

Chapter 2

Data Preparation

High-quality data preparation is essential for any machine translation model. Neural models are sensitive to noise and inconsistencies in the training data: even small preprocessing problems can degrade translation quality or hinder reproducibility.

This chapter introduces a practical data preparation workflow using parallel data from the **OPUS-Tatoeba** corpus (English–Dutch). This is an example workflow and the provided steps are useful to prepare other datasets as well for their usage in MT.

Google Colab

The implementation of this chapter can be found HERE.

2.1 The OPUS Collection

OPUS[1] (Tiedemann, 2012) is a large open collection of parallel corpora compiled from multilingual resources on the web. The project, initiated by Jörg Tiedemann, aggregates data from sources such as movie subtitles, parliamentary proceedings, software documentation, religious texts, TED talks, localisations, and many others.

OPUS provides:

- unified formats (e.g. Moses (see Section 1.5.4, TMX),
- automatic sentence alignment,
- language identification and metadata,
- downloadable parallel sentence pairs for thousands of language pairs,
- stable, versioned releases for reproducible experiments.

Because OPUS handles alignment and cleaning, it is widely used for MT training and evaluation.

[1] `https://opus.nlpl.eu`

2.2 The Tatoeba Project

Tatoeba[2] is a volunteer-driven project that collects example sentences translated into many languages. It began in 2006 and has grown into a large multilingual collection. The original dataset is organised as a many-to-many graph of translations, not as a clean parallel corpus.

OPUS releases a processed version of Tatoeba that:

- extracts verified translation pairs,
- normalises encoding,
- provides sentence-aligned files per language pair.

2.3 Overview of the Pipeline

We follow the standard MT data preparation workflow in the following sections.

2.3.1 Downloading OPUS-Tatoeba (EN–NL)

We use the Moses-format release for English–Dutch from OPUS-Tatoeba:

```
!wget https://object.pouta.csc.fi/OPUS-Tatoeba/v2023-04-12/moses/en-nl.txt.zip
!unzip *.zip
```

- The exclamation mark `!` tells Google Colab that this is a linux command and not a python command.
- **wget https://...** is the linux command to download a file from a url.
- **unzip *filename*** is the linux command to unzip a file.

2.3.2 Loading the Data

```
source = "Tatoeba.en-nl.en"
target = "Tatoeba.en-nl.nl"
```

OPUS always names files in the pattern `Corpus.src-tgt.src`. The lines in both files are aligned: line i in the English file is the translation of line i in the Dutch file.

We load the aligned files into a pandas DataFrame.

```
import pandas as pd
import csv

df_source = pd.read_csv(
    source, names=["Source"], sep="\0",
    quoting=csv.QUOTE_NONE, engine="python"
)
```

[2] https://tatoeba.org

```
df_target = pd.read_csv(
    target, names=["Target"], sep="\0",
    quoting=csv.QUOTE_NONE, engine="python"
)

df = pd.concat([df_source, df_target], axis=1)
```

- `import pandas as pd`
  We import the `pandas` library, which provides high-level data structures for working with tabular data. By convention, it is imported under the short alias `pd`.
- `import csv`
  We import Python's built-in `csv` module in order to access predefined constants that control how text files are parsed, such as how quotation marks are handled.
- `df_source = pd.read_csv(...)`
  We read the source-language file into a pandas DataFrame. Each line of the file corresponds to one sentence and becomes one row in the DataFrame.
- `source`
  This variable contains the path to the source-language text file.
- `names=["Source"]`
  Because the input file does not contain a header row, we explicitly assign a column name. The single column is named `Source`.
- `sep="\0"`
  We specify a null character as the separator. Since this character does not occur in normal text, pandas treats each entire line as a single field, even if the sentence itself contains spaces or punctuation.
- `quoting=csv.QUOTE_NONE`
  We disable quotation handling entirely. This ensures that quotation marks inside sentences are treated as normal characters rather than as delimiters.
- `engine="python"`
  We explicitly select the Python parsing engine, which supports custom separators such as the null character.
- `df_target = pd.read_csv(...)`
  We repeat the same procedure for the target-language file, storing the result in a separate DataFrame.
- `names=["Target"]`
  The single column of this DataFrame is named `Target`, corresponding to the target-language sentence.
- `df = pd.concat([df_source, df_target], axis=1)`
  We concatenate the source and target

2.4 Basic Cleaning

We apply simple but essential cleaning steps.

```
df = df.dropna()
df = df.drop_duplicates()
df = df[df["Source"] != df["Target"]].reset_index(drop=True)
```

- `df = df.dropna()`
  We remove all rows that contain missing values (`NaN`) in either the source or target column. In a parallel corpus, incomplete sentence pairs are not useful for training and must be discarded.
- `df = df.drop_duplicates()`
  We remove duplicate rows from the DataFrame. This eliminates repeated source–target sentence pairs, which could otherwise bias the model during training by over-representing certain examples.
- `df = df[df["Source"] != df["Target"]].reset_index(drop=True)`
  We filter out sentence pairs where the source and target sentences are identical. Such pairs typically indicate noise (e.g. untranslated sentences) and provide no useful learning signal for a translation model. After filtering, we reset the DataFrame index and drop the old index to ensure that row numbering remains consecutive.

Next, minimal markup removal (rare in Tatoeba, but included for consistency):

```
import re

clean_re = r"<.*?>|&?(amp|nbsp|quot);"

df["Source"] = (
    df["Source"].replace(clean_re, " ", regex=True)
                .replace(r"  +", " ", regex=True)
                .str.strip()
)

df["Target"] = (
    df["Target"].replace(clean_re, " ", regex=True)
                .replace(r"  +", " ", regex=True)
                .str.strip()
)
```

- `import re`
  We import Python's built-in `re` module, which provides support for regular expressions. Regular expressions allow us to search for and remove structured patterns such as markup and encoded symbols.
- `clean_re = r"<.*?>|&?(amp|nbsp|quot);"`
  We define a regular expression pattern that matches common types of markup and HTML entities:
  - `<.*?>` matches anything that looks like an HTML or XML tag.
  - `&?(amp|nbsp|quot);` matches common HTML entities such as `&`, ` `, and `&quot;`.
- `df["Source"] = (...)`
  We apply a sequence of text-cleaning operations to the source-language column of the DataFrame.

- `replace(clean_re, " ", regex=True)`
  All substrings that match the regular expression pattern are replaced by a single space. This removes markup while preserving word boundaries.
- `replace(r" +", " ", regex=True)`
  We collapse multiple consecutive spaces into a single space. This prevents the introduction of irregular spacing as a side effect of the previous replacement step.
- `str.strip()`
  We remove leading and trailing whitespace from each sentence.
- `df["Target"] = (...)`
  We repeat the same sequence of cleaning operations for the target-language column. Applying identical preprocessing to both languages is important for consistency in a parallel corpus.
- `print("After markup cleaning:", df.shape)`
  We print the size of the DataFrame after markup cleaning. The number of rows should remain unchanged, confirming that this step modifies sentence content but does not remove sentence pairs.

2.5 Sentence-Level Filtering

Even after basic cleaning, parallel corpora may still contain problematic sentence pairs. Some typical issues are:

- one side being much longer than the other (misalignment),
- unusually long sentences that are hard for the model to handle,
- residual noise that slipped through earlier steps.

Tatoeba sentences are generally short, but we include filters here both for good practice and for demonstration. We implement two common types of filters:

1. a **maximum length** filter, which removes sentences that are too long on either side;
2. a **length ratio** filter, which removes sentence pairs where one side is much longer than the other.

Let $|s|$ be the number of tokens on the source (English) side and $|t|$ the number of tokens on the target (Dutch) side. We define a maximum allowed ratio r and discard pairs that violate

$$\frac{\max(|s|, |t|)}{\min(|s|, |t|)} \leq r.$$

In practice, we approximate $|s|$ and $|t|$ using the number of spaces plus one (assuming one space between tokens), and we set $r =$ `max_ratio`.

```
max_spaces = 50
max_ratio  = 2.5

# Approximate token counts as number of spaces + 1
src_len = df["Source"].str.count(" ") + 1
tgt_len = df["Target"].str.count(" ") + 1
```

```
mask = (
    (src_len > tgt_len * max_ratio) |
    (tgt_len > src_len * max_ratio) |
    (src_len > max_spaces)          |
    (tgt_len > max_spaces)
)

df = df[~mask].reset_index(drop=True)
```

- `max_spaces` enforces an absolute upper bound on sentence length;
- `max_ratio` enforces that source and target lengths are not too different (by more than a factor of 2.5 in either direction);
- the boolean mask selects all sentence pairs that violate these constraints, and we remove them with `df = df[~mask]`.

This step removes many potential misalignments and extreme cases, resulting in a cleaner and more homogeneous training corpus.

2.6 Shuffling the Corpus

```
df = df.sample(frac=1.0, random_state=42).reset_index(drop=True)
```

- `df = df.sample(frac=1.0, random_state=42).reset_index(drop=True)`
  We randomly shuffle the rows of the DataFrame. Shuffling ensures that sentence pairs are presented to the model in a random order, which helps prevent undesired ordering effects during training.
- `frac=1.0`
  The parameter `frac` specifies the fraction of rows to sample. A value of `1.0` means that all sentence pairs are included, but in a randomized order.
- `random_state=42`
  We fix the random seed used for shuffling. This makes the shuffling process reproducible: running the code again with the same seed will result in the same order of sentence pairs.
- `reset_index(drop=True)`
  After shuffling, we reset the DataFrame index and discard the old index. This ensures that row indices remain consecutive and do not reflect the original ordering of the corpus.

2.7 Tokenisation

We apply the Moses tokeniser to both sides. Moses tokenization makes the effect of preprocessing immediately visible. For example, the raw English sentence `‘‘I can’t believe it’s true!’’` is transformed into `‘‘I ca n’t believe it ’s true !’’`, where punctuation is separated and contractions are split into consistent subunits. Similarly, Dutch-specific rules handle quotation marks and punctuation in a language-aware way. Such normalization reduces sparsity in the training data by ensuring that the same linguistic patterns are represented consistently across sentences.

Historically, Moses tokenization originates from the Moses statistical machine translation toolkit (Koehn et al., 2007) and became a de facto standard in MT research for many years. Even in neural machine translation, Moses-style tokenization remains widely used as a baseline because it is robust, language-aware, and well understood.

```
!pip install -q sacremoses
from sacremoses import MosesTokenizer

tok_en = MosesTokenizer(lang="en", escape=False)
tok_nl = MosesTokenizer(lang="nl", escape=False)

df["Source_Tok"] = df["Source"].apply(lambda s: tok_en.tokenize(s,
↪  return_str=True))
df["Target_Tok"] = df["Target"].apply(lambda s: tok_nl.tokenize(s,
↪  return_str=True))
```

- `!pip install -q sacremoses`
  We install the `sacremoses` package, which is a Python reimplementation of the tokenization and preprocessing tools from the Moses statistical machine translation toolkit. The `-q` flag suppresses verbose installation output.
- `from sacremoses import MosesTokenizer`
  We import the `MosesTokenizer` class, which provides language-aware tokenization rules commonly used in machine translation.
- `tok_en = MosesTokenizer(lang="en", escape=False)`
  We create a tokenizer configured for English. This tokenizer applies English-specific rules for splitting text into tokens, handling punctuation, contractions, and special symbols. `escape=False` indicates that apostrophes should not be changed into `&apos;`.
- `tok_nl = MosesTokenizer(lang="nl", escape=False)`
  We create a tokenizer configured for Dutch. Using a language-specific tokenizer ensures that tokenization respects conventions of the target language.
- `df["Source_Tok"] = df["Source"].apply(...)`
  We apply the English tokenizer to each source-language sentence in the DataFrame and store the tokenized result in a new column called `Source_Tok`.
- `lambda s: tok_en.tokenize(s, return_str=True)`
  For each sentence `s`, we tokenize it using the Moses tokenizer. The parameter `return_str=True` ensures that the output is a single space-separated string of tokens rather than a list.
- `df["Target_Tok"] = df["Target"].apply(...)`
  We repeat the same procedure for the target-language sentences, storing the tokenized output in the column `Target_Tok`.
- `lambda s: tok_nl.tokenize(s, return_str=True)`
  Each Dutch sentence is tokenized according to Dutch-specific rules, ensuring consistent and language-appropriate preprocessing.

2.8 Train/Dev/Test Split

Splitting the data into training, development, and test sets is a fundamental principle of experimental machine translation. The training set is used to learn model parameters, while the development set provides an unbiased signal for monitoring progress, tuning hyperparameters, and applying techniques such as early stopping. Crucially, the test set is kept completely separate and is only used once the model design is finalized. This separation prevents overfitting and ensures that reported evaluation scores reflect the model's ability to generalize to unseen data rather than its ability to memorize the training corpus. By fixing random seeds and clearly defining the splits, we also ensure reproducibility, allowing experiments to be repeated and compared in a scientifically sound manner.

We extract small dev/test sets; the remainder becomes training data.

```
num_dev = 1000
num_test = 1000

df_dev = df.sample(n=num_dev, random_state=1)
df_train = df.drop(df_dev.index)

df_test = df_train.sample(n=num_test, random_state=2)
df_train = df_train.drop(df_test.index)

print("Train/dev/test sizes:",
      len(df_train), len(df_dev), len(df_test))
```

- `num_dev = 1000`
  We define the number of sentence pairs to be used for the development (validation) set. This set is used during training to monitor model performance and tune hyperparameters.
- `num_test = 1000`
  We define the number of sentence pairs to be used for the test set. This set is kept separate and is only used for the final evaluation of the trained model.
- `df_dev = df.sample(n=num_dev, random_state=1)`
  We randomly sample `num_dev` sentence pairs from the full DataFrame to create the development set. Fixing the random seed ensures that the same sentences are selected each time the code is run.
- `df_train = df.drop(df_dev.index)`
  We remove the development sentences from the original DataFrame. The remaining sentence pairs form a temporary training pool.
- `df_test = df_train.sample(n=num_test, random_state=2)`
  From the remaining data, we randomly sample `num_test` sentence pairs to create the test set. A different random seed is used to ensure an independent selection from the development set.
- `df_train = df_train.drop(df_test.index)`
  We remove the test sentences from the training pool. The remaining sentence pairs constitute the final training set.
- `print("Train/dev/test sizes:", ...)`
  We print the number of sentence pairs in each split. This allows us to verify that the data has been correctly partitioned and that the splits are mutually exclusive.

2.9 Saving the Final Output Files

2.9.1 Saving files locally in Colab

We write the cleaned output files locally

```
corpus_prefix="tatoeba-en-nl."

def write_split(df_split, srcfile, tgtfile):
    with open(srcfile, "w", encoding="utf-8") as sf:
        sf.write("\n".join(df_split["Source_Tok"]) + "\n")
    with open(tgtfile, "w", encoding="utf-8") as tf:
        tf.write("\n".join(df_split["Target_Tok"]) + "\n")
    print(f"Wrote {srcfile}, {tgtfile}")

write_split(df_train, corpus_prefix+"train.en", corpus_prefix+"train.nl")
write_split(df_dev,   corpus_prefix+"dev.en",   corpus_prefix+"dev.nl")
write_split(df_test,  corpus_prefix+"test.en",  corpus_prefix+"test.nl")
```

- `def write_split(df_split, srcfile, tgtfile):`
  We define a function that writes a single data split (training, development, or test) to disk as two separate text files: one for the source language and one for the target language.
- `df_split`
  This argument contains a DataFrame representing one data split. Each row corresponds to one parallel sentence pair.
- `srcfile, tgtfile`
  These arguments specify the filenames for the source-language and target-language output files.
- `with open(srcfile, "w", encoding="utf-8") as sf:`
  We open the source-language output file in write mode using UTF-8 encoding, which ensures correct handling of multilingual text.
- `sf.write("\n".join(df_split["Source_Tok"]) + "\n")`
  We write all tokenized source-language sentences to the file, one sentence per line. Sentences are joined using newline characters, and a final newline is added to comply with common MT data format conventions.
- `with open(tgtfile, "w", encoding="utf-8") as tf:`
  We open the target-language output file in the same way.
- `tf.write("\n".join(df_split["Target_Tok"]) + "\n")`
  We write the tokenized target-language sentences to the target file, again using one sentence per line and preserving sentence alignment with the source file.
- `write_split(df_train, corpus_prefix+"train.en", corpus_prefix+"train.nl")`
  We write the training split to disk, producing the files `tatoeba-en-nl.train.en` and `tatoeba-en-nl.train.nl`.
- `write_split(df_dev, corpus_prefix+"dev.en", corpus_prefix+"dev.nl")`
  We write the development split to disk.
- `write_split(df_test, corpus_prefix+"test.en", corpus_prefix+"test.nl")`

We write the test split to disk.

2.9.2 Copying the files to Google Drive

Now the files have been written in our local directory on Google Colab. In order to keep them for reuse later, we have to copy them to our Google Drive.

We first need to connect to our Google Drive. Then we need to make sure we create a dedicated target directory.

```
!mkdir /content/drive/MyDrive/MTAT/
!cp *.en /content/drive/MyDrive/MTAT/
!cp *.nl /content/drive/MyDrive/MTAT/
```

- `mkdir path` creates the path if it doesn't exist yet;
- `cp *.en target_path` copies all files ending with `.en` to `path`
- `cp *.nl target_path` does the same for the files ending with `.nl`.

2.10 Discussion and Variants of the Pipeline

The data preparation steps presented in this chapter correspond to a *standard and widely used MT preprocessing pipeline*. Variants of this pipeline (with minor modifications) have been used for many years in both statistical and neural machine translation research and remain a solid default choice for controlled experiments.

The core principles illustrated here are broadly applicable:

- careful cleaning of parallel data,
- removal of noise and misalignments,
- consistent preprocessing on source and target sides,
- explicit and reproducible train/dev/test splits.

At the same time, it is important to be aware that several steps in the pipeline are *design choices* rather than absolute requirements. Two common variations are discussed below.

2.10.1 Lowercasing

In many traditional MT pipelines, all text is converted to lowercase during preprocessing. Lowercasing reduces vocabulary size and sparsity by collapsing word forms such as *"House"* and *"house"* into a single token. This was particularly important in phrase-based and early neural MT systems, where vocabulary size had a strong impact on model capacity and training stability.

Lowercasing can still be useful when:

- training models on very limited data,
- working with highly noisy text,
- prioritising robustness over orthographic fidelity.

However, modern neural MT systems are generally able to handle case distinctions well, especially when trained on sufficient data. For this reason, many contemporary MT pipelines preserve the original casing, as we have done in this chapter. Case preservation allows models to learn proper capitalization, sentence-initial casing, and named entity conventions directly from the data.

2.10.2 Tokenisation vs. Subword Segmentation

We used Moses tokenization in this chapter because it:

- makes preprocessing steps explicit and transparent,
- provides language-aware normalization,
- is easy to inspect and understand for educational purposes.

In current neural MT practice, however, explicit word-level tokenization is often replaced—or complemented—by *subword segmentation* methods such as Byte Pair Encoding (BPE) or SentencePiece (unigram or BPE models). Subwording addresses the open-vocabulary problem by decomposing rare or unseen words into smaller units, allowing models to generalize better to new word forms.

When subwording is used, the pipeline typically changes as follows:

- minimal normalization is applied (often just cleaning and optional lowercasing),
- Moses tokenization may be skipped entirely,
- a shared or language-specific subword model is learned on the training data and applied consistently to train, dev, and test sets.

In such setups, subword segmentation implicitly performs much of the work that tokenization used to handle, including punctuation separation and morphological variation. As a result, modern MT systems often operate directly on subword units rather than on word tokens.

2.11 Exercise: Process your own dataset

For a language pair you are familiar with, perform the following steps:

- Select a dataset from Opus
- Check the data manually: does it contain full sentences
- Check the size of the dataset:
  - if it is smaller than Tatoeba, it may be too small
  - if it is bigger than a few million sentences per language, it may be too big for practical processing, i.e. it may be too big to be read into the memory of your Google colab session, or it may take a very long time to process. On the other hand, it may lead to better translations, once your model is trained on it
- Process it with the steps as provided in the colab session.
- Write the final version to your Google drive
- Write a small report on your dataset processing, containing information on the raw dataset size and the cleaned up dataset size, and the exact cleaning procedure that you applied.

Chapter 3

Machine Translation Evaluation

Machine Translation evaluation addresses one of the central questions in MT: *How good is a translation?* Because many different translations for a sentence may be equally acceptable, MT evaluation is inherently challenging.

This chapter introduces the main paradigms for evaluating MT, incorporating standard academic practice and evaluation methods from international shared tasks.

3.1 Why MT Evaluation Is Difficult

Example of Translations

Chinese : 这个 机场 的 安全 工作 由 以色列 方面 负责 .

Translations :

- Israeli officials are responsible for airport security.
- Israel is in charge of the security at this airport.
- The security work for this airport is the responsibility of the Israel government.
- Israeli side was in charge of the security of this airport.
- Israel is responsible for the airport's security.
- Israel is responsible for safety work at this airport.
- Israel presides over the security of the airport.
- Israel took charge of the airport security.
- The safety of this airport is taken charge of by Israel.
- This airport's security is the responsibility of the Israeli security officials.

This example from Koehn (2025) illustrates why evaluation is hard: many syntactic choices (e.g. topic–comment order, passive restructuring, temporal adverbial placement) must be handled correctly, and yet multiple very different English translations may all be equally acceptable.

3.2 Human Evaluation

Human evaluation of translation is considered the most reliable method but also the most costly. Note that this can as well apply to the evaluation of human translations as to the evaluation of machine translations.

Human evaluation is standard in machine translation shared tasks at WMT and IWSLT.

3.2.1 Rating-Based Evaluation

3.2.1.1 Adequacy and Fluency

Human raters commonly assess two dimensions of translations, i.e. *adequacy* and *fluency*.

Adequacy Given the source sentence and a translation, the human assessor is asked to answer the question: *How well does the translation preserve the meaning of the source?* This requires a bilingual assessor who compares the source sentence with the translation.

As it may be hard to get bilingual assessors, an alternative approach is to present the assessor with a reference translation instead of the source, so a monolingual assessor can compare the reference with the translation that needs assessment.

The adequacy question often needs to be answered on a five point Likert scale, as shown in Table 3.1a

Adequacy	
5	all meaning
4	most meaning
3	much meaning
2	little meaning
1	no meaning

(a) Adequacy Likert scale

Fluency	
5	flawless target language
4	good target language
3	non-native target language
2	disfluent target language
1	incomprehensible

(b) Fluency Likert scale

Table 3.1: Likert scales used for MT evaluation

Fluency For fluency, the assessor does not need to compare the translation with the source or a reference, but only needs to answer the question *How natural, grammatical, and idiomatic is the translation?*

Exercise: Assess Translation Quality (Koehn, 2025)

Rank according to adequacy and fluency on a 1-5 scale as in Tables 3.1a and 3.1b.

Source : L'affaire NSA souligne l'absence totale de d´ebat sur le renseignement

Reference : NSA Affair Emphasizes Complete Lack of Debate on Intelligence

System1 : The NSA case underscores the total lack of debate on intelligence

System2 : The case highlights the NSA total absence of debate on intelligence

System3 : The matter NSA underlines the total absence of debates on the piece of information

3.2.1.2 Likert scales and their limitations

It is important to note that adequacy and fluency scores are typically collected using a Likert scale, such as the 1–5 scale shown above. Likert scales are *ordinal* scales: the numbers indicate an order (e.g. a score of 4 is better than a score of 3), but they do not guarantee that the difference between consecutive scores is the same. In other words, the perceived difference between scores 1 and 2 may not be comparable to the difference between scores 4 and 5.

For this reason, treating Likert scores as if they were numerical measurements on an interval scale can be misleading. In particular, computing average scores implicitly assumes equal distances between scale points, an assumption that is not justified for ordinal data. As a result, analyses based solely on mean adequacy or fluency scores should be interpreted with caution, and alternative summaries such as score distributions or median values are often more appropriate.

3.2.1.3 Inter-rater Agreement

When human evaluation involves multiple annotators, an important question is how consistently different raters judge the same translation output. System outputs may receive very different adequacy and fluency scores even if they share much lexical material.

Figure 3.1 shows the distribution of answers on the adequacy scale for the different human evaluators in the WMT 2006 task.

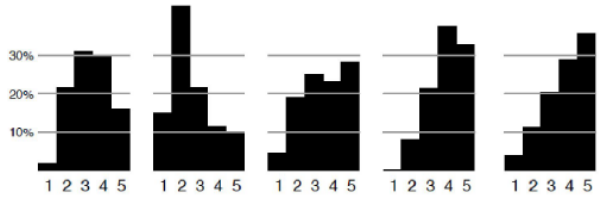


Figure 3.1: Histograms of adequacy judgments by different human evaluators. Figure from Koehn (2025).

The consistency is known as *inter-rater agreement*. High inter-rater agreement means that different annotators tend to assign similar scores to the same translation, while low agreement indicates substantial disagreement between raters.

In machine translation evaluation, achieving high inter-rater agreement is notoriously difficult. Judgments of adequacy and fluency rely on subjective interpretation, and annotators may differ in linguistic background, tolerance for errors, familiarity with the domain, or expectations of translation quality. For example, one rater may penalize minor grammatical issues strongly, while another may focus primarily on meaning preservation. As a result, the same system output can receive very different scores from different annotators.

Low inter-rater agreement poses serious methodological problems. If annotators do not agree, it becomes unclear whether observed differences between systems reflect genuine quality differences or merely annotator variability. This noise reduces the reliability of evaluation results and makes it harder to draw strong conclusions from human scores. While statistical measures such as Cohen's κ or Krippendorff's α can be used to quantify agreement, their values are often low for adequacy and fluency ratings, highlighting the limitations of coarse ordinal scales.

These challenges motivated the search for alternative evaluation methods that reduce subjectivity and improve consistency across annotators. One influential response within the WMT community

has been the adoption of Direct Assessment, which replaces discrete category judgments with continuous rating scales and carefully controlled annotation procedures.

3.2.1.4 Rater Effects, Training, and Normalization

As discussed in the previous section, disagreement between human raters is a central challenge in machine translation evaluation. Part of this disagreement stems not from the translations themselves, but from systematic differences between raters. As noted by Jurafsky and Martin (2025), crowdworkers and human annotators frequently conflate adequacy and fluency, apply rating scales inconsistently, and vary widely in how strict or lenient their judgments are.

These *rater effects* introduce noise into human evaluation results. For example, some raters may consistently assign higher scores than others, or use only a narrow portion of the available scale. Without correction, such effects can distort system comparisons and reduce the reliability of conclusions drawn from the data.

To mitigate these problems, common practice in large-scale evaluations includes both rater filtering and score normalization. Annotators whose ratings are highly inconsistent or clearly deviate from expected behavior can be treated as outliers and excluded. In addition, scores are often *z-normalized per rater*, transforming each rater's scores to have zero mean and unit variance. This procedure removes individual differences in scale usage while preserving relative judgments within each rater's annotations.

These normalization techniques play a crucial role in modern human evaluation frameworks and form an important methodological bridge between traditional ordinal judgments and the continuous rating schemes used in Direct Assessment.

3.2.1.5 Direct Assessment

In recent years, the WMT shared tasks have moved away from traditional discrete scales toward continuous rating scales, most commonly implemented as sliders ranging from 0 to 100 (Graham et al., 2013). This approach is known as *Direct Assessment* and was introduced to provide more fine-grained human quality judgments than coarse ordinal scales. Direct Assessment generally improves inter-annotator agreement compared to 1–5 Likert scales.

Direct Assessment has been the official human evaluation metric in the WMT General MT task since its introduction in 2017 and has been used, often with guidelines to stabilize scoring, in subsequent annual shared tasks (e.g., in WMT22 and later) as an alternative to discrete, ordinal adequacy and fluency ratings.

3.2.2 Ranking-Based Evaluation

One way to reduce scale interpretation problems is to avoid absolute scores altogether and instead ask annotators to directly compare system outputs.

In ranking-based evaluation, annotators are not asked to assign absolute quality scores to translations. Instead, they are presented with two or more system outputs for the same source sentence and asked to directly compare them. A typical annotation task is formulated as:

Ranking Translations Example

Is translation A better than, worse than, or equal to translation B?

Translation A Israeli officials are responsible for airport security.

Translation B Israel is in charge of the security at this airport.

This approach reframes evaluation as a *relative judgment* task. Rather than deciding how good a translation is in isolation, annotators only need to decide which of two alternatives is better. Such pairwise comparisons are often easier for humans to make and reduce the cognitive burden associated with interpreting numerical rating scales.

Ranking-based evaluation was widely used in early WMT shared tasks because it tends to yield higher inter-rater agreement than adequacy and fluency scoring (Callison-Burch et al., 2007). Since annotators are comparing translations of the same source sentence, many sources of variability—such as individual scale usage, overall strictness, or differences in score interpretation—are reduced. As a result, annotators are more likely to agree on relative preferences even when they disagree on absolute quality levels.

However, ranking-based evaluation also has important limitations. It does not provide an absolute notion of translation quality, only relative preferences between systems. In addition, the number of comparisons grows rapidly with the number of systems being evaluated, making large-scale evaluation costly. These practical constraints motivated later WMT evaluations to move toward continuous-scoring approaches such as Direct Assessment, which combine fine-grained judgments with improved statistical reliability.

3.2.3 MQM Error Annotation

MQM (*Multidimensional Quality Metrics*) is a human evaluation framework that approaches translation quality through systematic error analysis rather than holistic judgment (Lommel et al., 2014). Instead of asking annotators to rate or compare translations, MQM requires them to explicitly identify and categorize errors in the translation output according to a predefined taxonomy.

The MQM framework (see also Figure 3.2) organizes errors along multiple dimensions, including:

- **Accuracy**, covering omissions, mistranslations, untranslated words, and additions;
- **Fluency**, including grammatical errors, word order issues, and spelling;
- **Terminology**, capturing incorrect or inconsistent term usage;
- **Style**, such as register mismatches;
- **Locale conventions**, including punctuation, formatting, and cultural norms.

Each identified error is assigned a severity level (typically minor, major, or critical). These severities are weighted and aggregated to produce a penalty-based score, often normalized by sentence length to yield a penalty-per-word metric.

Compared to adequacy and fluency ratings, MQM offers greater diagnostic power. While holistic scores indicate *that* a translation is poor, MQM reveals *why* it is poor by pinpointing specific error types. This makes MQM particularly valuable for system development, error analysis, and quality assurance workflows. However, MQM annotation is substantially more expensive and time-consuming, requiring trained annotators and detailed guidelines to ensure consistency.

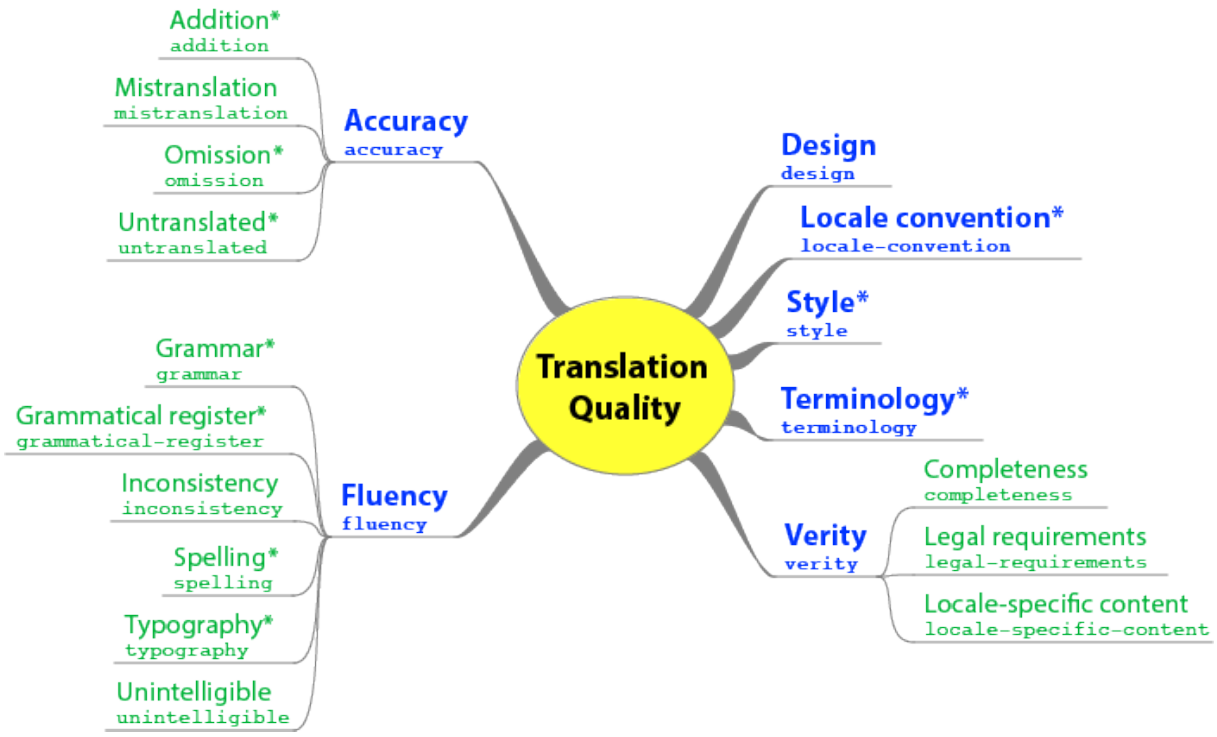


Figure 3.2: The core error categories proposed by the MQM guidelines. From `https://sites.middlebury.edu/runyul/2018/03/04/translation-quality-assessment-mqm-multidimensional-quality-metrics/`.

In contrast to ranking-based evaluation, which captures relative system preferences, MQM provides fine-grained, interpretable feedback at the level of individual errors. Ranking is efficient and yields higher inter-rater agreement, but it offers limited insight into linguistic weaknesses. MQM trades off scalability for analytical depth.

Finally, unlike Direct Assessment, which aims to produce reliable numerical quality estimates at scale, MQM does not primarily target system-level ranking. Instead, it is designed to support qualitative analysis and informed decision making. As a result, MQM is often used in complementary fashion: Direct Assessment is employed to compare overall system quality, while MQM is applied to smaller samples to understand error patterns and guide further system improvements.

3.2.4 Post-editing as Evaluation

Post-editing-based evaluation approaches translation quality indirectly by *measuring the effort required to transform machine translation output into a fully correct and acceptable human translation*. Instead of assigning scores or annotating errors, human annotators are asked to *edit* the system output until it meets predefined quality standards, such as adequacy, fluency, and terminological correctness.

The intuition behind this approach is that better translations require less human effort to correct. Quality can therefore be quantified by comparing the original *raw* MT output to its post-edited version using edit-distance-based measures.

HTER calculation HTER (Human-targeted Translation Edit Rate) (Snover et al., 2006) quantifies post-editing effort by measuring the minimal number of editing operations required to

transform the raw MT output into a human post-edited version. The allowed operations are:

- **Insertion**: adding a missing word or phrase,
- **Deletion**: removing an incorrect or superfluous element,
- **Substitution**: replacing an incorrect word or phrase,
- **Shift**: moving a contiguous sequence of words to a different position in the sentence.

HTER is computed as the total number of edits divided by the length of the post-edited sentence:

$$\text{HTER} = \frac{\#\text{edits}}{\#\text{words in post-edited translation}}$$

HTER Calculation Example

Consider the following English–Dutch translation:
Source Yesterday, the minister announced the decision.

MT output	De	minister	kondigde	de	beslissing	aan	gisteren
Post-edited	Gisteren	kondigde	de	minister	de	beslissing	aan

To obtain the post-edited translation, the annotator performs **two shift** operations. First, the adverb *gisteren* is moved from the sentence-final position to the sentence-initial position. Second, the noun phrase *de minister* is shifted from a preverbal position to a postverbal position. No insertions, deletions, or substitutions are required.
The post-edited sentence contains seven words. The HTER score is therefore:

$$\text{HTER} = \frac{2}{7} \approx 0.29$$

Why this requires two edits. Each shift operation in HTER moves a single contiguous block of words to a new position. Although multiple word positions change as a result of reordering, only explicitly shifted blocks are counted as edits. In this example, moving *gisteren* alone does not yield the post-edited word order; an additional shift of *de minister* is required. HTER therefore counts two shift operations, reflecting the minimal number of reordering actions needed to obtain the human-targeted translation.

Compared to adequacy and fluency ratings, post-editing provides a more objective signal, as it is grounded in observable editing actions rather than subjective judgments. However, post-editing effort still depends on individual annotator behavior, such as editing style, tolerance for minor issues, and familiarity with the domain. As a result, rater effects and normalization may still be necessary.

In contrast to ranking-based evaluation, post-editing yields an absolute, sentence-level quality measure rather than a relative preference. Unlike MQM, which focuses on identifying and categorizing errors, post-editing captures the *cumulative impact* of all errors on human effort, without explicitly distinguishing their types. This makes it particularly attractive for industrial settings, where the cost of human correction is a central concern.

Post-editing-based evaluation is therefore best viewed as a bridge between human-centered evaluation and real-world translation workflows. While it is less scalable than Direct Assessment and less diagnostically informative than MQM, it provides a pragmatic measure of translation quality that closely aligns with professional translation practice.

3.3 Automatic Evaluation Metrics

Automatic evaluation metrics aim to approximate human judgments of translation quality at a much lower cost, higher speed and higher consistency. Given a machine translation output and one or more human reference translations, these metrics compute a numerical score intended to reflect translation quality. Historically, the development of automatic metrics mirrors the evolution of machine translation itself, progressing from surface-based string matching to increasingly semantic, learned approaches.

3.3.1 Surface oriented metrics

3.3.1.1 Precision and Recall as Evaluation Concepts

Many automatic evaluation metrics for machine translation are based on measuring overlap between a system output and a reference translation. Two fundamental concepts underlying such overlap-based evaluation are *precision* and *recall*. These notions help clarify different types of translation errors, such as unnecessary additions versus missing content.

Precision. Precision measures how much of the system output is correct. In machine translation, it answers the question: *To what extent does the translation avoid introducing incorrect or unwarranted content?* A translation with high precision contains few words or expressions that are not supported by the reference.

Recall. Recall measures how much of the reference content is recovered by the system output. In MT terms, it answers the question: *To what extent does the translation cover the content that should be present?* A translation with low recall omits information that appears in the reference.

Formal definition. Let M be the number of matched units (M for matches) between the system output and the reference, H the number of units in the system output (H for hypothesis), and R the number of units in the reference. Precision and recall are defined as:

$$\text{Precision} = \frac{M}{H} \qquad \text{Recall} = \frac{M}{R}$$

The definition of a "unit" depends on the metric and may correspond to words, n-grams, or other representational elements.

Combining precision and recall. Because precision and recall capture complementary aspects of translation quality, they are often combined into a single score using a weighted harmonic mean, known as the F-score:

$$F_\beta = \frac{(1+\beta^2)\,\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}$$

The parameter β controls the relative importance of recall versus precision. Values $\beta > 1$ place greater emphasis on recall, reflecting the intuition that missing essential content is often more harmful than introducing minor additional material.

Example: Precision, Recall, and F-measure (word overlap)

System A:
`Israeli officials responsibility of airport safety`

Reference:
`Israeli officials are responsible for airport security`

Assume a simple word-overlap matching where the following three words are counted as correct matches: `Israeli`, `officials`, `airport`

- **correct** $M = 3$.
- **output-length** $H = 6$ (`Israeli, officials, responsibility, of, airport, safety`)
- **reference-length** $R = 7$ (`Israeli, officials, are, responsible, for, airport, security`)

$$\text{Precision} = \frac{\text{correct}}{\text{output-length}} = \frac{3}{6} = 0.50 = 50\%.$$

$$\text{Recall} = \frac{\text{correct}}{\text{reference-length}} = \frac{3}{7} \approx 0.43 = 43\%.$$

F-measure (harmonic mean)

$$F_1 == \frac{2 \cdot 0.50 \cdot 0.43}{0.50 + 0.43} \approx 0.46 = 46\%.$$

(Equivalent form used on the slides)

$$F_1 = \frac{P \cdot R}{(P + R)/2} = \frac{0.50 \cdot 0.43}{(0.50 + 0.43)/2} \approx 0.46 = 46\%.$$

The following example demonstrates why precision and recall are not very good in measuring MT quality.

Limitation of Precision and Recall: Insensitivity to Word Order

Reference:
`Israeli officials are responsible for airport security`

System A:
`Israeli officials responsibility of airport safety`

System B:
`airport security Israeli officials are responsible`

Under a simple word-overlap evaluation, all content words produced by System B appear in the reference. As a result, System B achieves *perfect precision* and *perfect recall*, and therefore an F-measure of 100%.

However, System B is clearly not a well-formed translation: although it contains the correct words, they appear in an unnatural order and fail to express the intended syntactic relations. Precision and recall count which words are present, but they do not account for *where* those words occur or how they are structured in the sentence.

This example illustrates a fundamental limitation of overlap-based precision and recall: they do not penalize incorrect reordering. Consequently, a translation can receive a perfect score despite being ungrammatical or difficult to understand, as long as it contains the right words.

3.3.1.2 Edit-distance and Word Error Rate

One of the earliest ideas for automatic evaluation is to measure how many edits are required to transform a system output into a reference translation. This approach is based on the Levenshtein distance, which counts word insertions, deletions, and substitutions. When normalized by the length of the reference, this yields the *Word Error Rate* (WER).

While intuitive, WER has important limitations for translation evaluation. It penalizes legitimate reordering heavily and treats all word mismatches equally, even when they do not affect meaning. As a result, WER correlates poorly with human judgments for many translation tasks.

WER computes the minimum number of word-level edits needed to transform the MT output into the reference.

WER is a *normalized* error measure rather than a raw edit count. After computing the minimum number of edit operations (insertions, deletions, and substitutions), this number is divided by the total number of words in the reference translation. Formally, if E is the number of edits and N is the reference length, then

$$\text{WER} = \frac{E}{N}.$$

Normalization is necessary because longer sentences naturally allow for more errors than shorter ones. Dividing by the reference length ensures that WER scores are comparable across sentences of different lengths.

In this example synonymous words are treated as errors because only exact matches count.

Word Error Rate Example

Ref	Has	France	benefited	from	information	provided	by	the	NSA	?
MT	Did	France	profit	from	information	supplied	by	the	NSA	?
Type	S	M	S	M	M	S	M	M	M	M

Each column corresponds to a word position after optimal alignment. Exact word matches (M) incur no cost, while substitutions (S) contribute one edit operation.
Even though *benefited* and *profit*, as well as *provided* and *supplied*, are semantically equivalent, they are treated as substitutions. This illustrates why WER often over-penalizes acceptable translations.
In the example there are three substitutions and the reference contains ten tokens. The resulting Word Error Rate is therefore

$$\text{WER} = \frac{3}{10} = 0.30.$$

This value can be interpreted as saying that, on average, 30% of the reference words would need to be edited to transform the MT output into the reference.

3.3.1.3 BLEU: N-gram Precision

A major milestone in MT evaluation was the introduction of BLEU (Papineni et al., 2002). BLEU measures the overlap of word n-grams between the machine translation output and one or more reference translations. Precision is computed for n-grams of length one to four, and the scores are combined using a geometric mean. To discourage overly short translations, BLEU includes a brevity penalty.

BLEU is typically computed at the corpus level rather than for individual sentences. Its simplicity, efficiency, and language independence made it the de facto standard metric in MT research for

many years. However, BLEU relies on exact word matching, does not explicitly model recall, and is often difficult to interpret intuitively. As a result, BLEU scores should be used primarily for system comparison rather than as absolute indicators of translation quality.

BLEU Example: N-gram Precision and Brevity Penalty

Reference: Has France benefited from information provided by the NSA ?
MT output: Did France profit from information supplied by the NSA ?

Exact n-gram matches

Reference	Has	France	benefited	from	information	provided	by	the	NSA	?
MT output	Did	France	profit	from	information	supplied	by	the	NSA	?
Unigram	×	✓	×	✓	✓	×	✓	✓	✓	✓
Bigram	×	×	×	✓	×	×	✓	✓	✓	–
Trigram	×	×	×	×	×	×	✓	✓	–	–
4-gram	×	×	×	×	×	×	×	✓	–	–

Each row indicates whether an n-gram starting at that position in the MT output also occurs in the reference. Only *exact* matches count. Synonymous expressions such as *benefited* vs. *profit* and *provided* vs. *supplied* are treated as mismatches.

Step 1: n-gram precision The MT output contains:

- 10 unigrams, of which 7 match the reference:
$$p_1 = \frac{7}{10}$$
- 9 bigrams, of which 4 match:
$$p_2 = \frac{4}{9}$$
- 8 trigrams, of which 2 match:
$$p_3 = \frac{2}{8}$$
- 7 four-grams, of which 1 matches:
$$p_4 = \frac{1}{7}$$

Step 2: Geometric mean of precisions BLEU combines the n-gram precisions using the geometric mean:

$$\mathrm{GM} = \sqrt[4]{p_1 \cdot p_2 \cdot p_3 \cdot p_4} = \sqrt[4]{\frac{7}{10} \cdot \frac{4}{9} \cdot \frac{2}{8} \cdot \frac{1}{7}}.$$

The geometric mean strongly penalizes low precision at higher n-gram orders: even a small number of mismatches early in the sentence breaks many longer n-grams.

Step 3: Brevity penalty The MT output and reference have the same length:

$$c = r = 10.$$

The brevity penalty is therefore:

$$\mathrm{BP} = \min\left(1, \frac{c}{r}\right) = 1.$$

Final BLEU score The final BLEU score is obtained by multiplying the geometric mean by the brevity penalty:

$$\mathrm{BLEU} = \mathrm{BP} \times \mathrm{GM}.$$

Because BLEU relies on exact n-gram overlap and does not model recall, it penalizes paraphrasing and synonymy much more strongly than WER or chrF, even when the translation is semantically adequate.

BLEU can be computed with respect to multiple reference translations for the same source sentence. In this setting, the machine translation output is compared against all available references, and an n-gram match is counted if the n-gram occurs in *any* of the references. This design reflects the fact that multiple translations may be equally valid and helps reduce the penalty for legitimate lexical or syntactic variation. To prevent systems from being unfairly rewarded for repeating frequent words, BLEU applies *clipped counts*: for each n-gram in the system output, the maximum number of matches is limited to the highest count observed across the reference set. When computing the brevity penalty, BLEU uses the reference length that is closest to the system output length, rather than averaging over all references. Together, these mechanisms allow BLEU to accommodate translation variability while maintaining a conservative precision-based evaluation.

Limitations of BLEU Despite its historical importance and widespread use, BLEU has several well-known limitations that affect its interpretability and correlation with human judgments:

- **Lack of recall sensitivity.** BLEU measures only n-gram precision and does not explicitly model recall. As a result, translations that omit important content may still obtain reasonable BLEU scores as long as the remaining output matches the reference well.
- **Unreliable at the sentence level.** BLEU was designed as a corpus-level metric. When applied to individual sentences, higher-order n-gram counts become sparse, leading to unstable and difficult-to-interpret scores.
- **Over-penalization of legitimate variation.** Because BLEU relies on exact word n-gram matching, it penalizes valid paraphrases, synonym substitutions, and alternative word orders, even when meaning is preserved.
- **Sensitivity to morphological variation.** In morphologically rich languages, small inflectional differences can break multiple word n-grams, disproportionately lowering BLEU scores despite minimal semantic impact.
- **Dependence on tokenization.** BLEU scores are strongly affected by tokenization and preprocessing choices, which can inflate or deflate scores and complicate comparison across systems or studies.

3.3.1.4 Translation Edit Rate

Translation Edit Rate (TER) (Snover et al., 2006) extends word-level edit-distance metrics such as WER by allowing the movement of contiguous blocks of words (shifts) to count as a single edit operation. TER computes the minimum number of insertions, deletions, substitutions, and shifts required to transform the MT output into a reference translation, normalized by the reference length.

HTER (*Human-targeted TER*) uses the same edit operations and normalization as TER, but differs in the choice of reference. The concrete calculation of TER / HTER, including shift operations, is illustrated in Section 3.2.4.

While TER compares the MT output to an independently produced human reference, HTER computes the edit distance with respect to a post-edited version of the system output. As a result, TER measures similarity to a fixed reference, whereas HTER is more closely aligned with the human effort required to correct the translation.

3.3.1.5 Character-based Metrics: chrF

Character-based metrics such as chrF (Popovic, 2015) were introduced to overcome several limitations of word-based overlap metrics, most notably BLEU. By operating on *character n-grams* rather than word n-grams, chrF is less sensitive to tokenization choices, inflectional morphology, and minor orthographic variation. This makes chrF particularly well suited to morphologically rich languages and to translation scenarios where word segmentation is ambiguous.

Like BLEU, chrF measures surface overlap between the MT output and a reference translation. However, unlike BLEU, chrF explicitly models both *precision* and *recall*, allowing it to penalize missing content more directly.

Definition and normalization chrF computes overlap over *character n-grams of multiple orders*, rather than relying on a single n. In practice, character n-grams are extracted for $n = 1$ up to a maximum order N (by default, $N = 6$).

For each n, two quantities are computed:

- **character precision ($\mathbf{chrP}_n$):** the proportion of character n-grams in the MT output that also occur in the reference;
- **character recall ($\mathbf{chrR}_n$):** the proportion of character n-grams in the reference that are covered by the MT output.

As with BLEU, normalization is essential: precision normalizes by the length of the MT output, while recall normalizes by the length of the reference. This ensures comparability across sentences of different lengths.

Precision and recall are first computed *separately for each n-gram order* and then *aggregated across all* $n = 1 \dots N$ (typically by micro-averaging). The resulting overall precision and recall are finally combined into a single score using a weighted F-measure:

$$\mathrm{chrF}_\beta = \frac{(1+\beta^2)\,\mathrm{chrP} \cdot \mathrm{chrR}}{\beta^2 \cdot \mathrm{chrP} + \mathrm{chrR}}.$$

The parameter β controls the relative importance of recall versus precision. Following Popovic (2015), most MT evaluations use $\beta = 2$, which reflects the intuition that omitting translated content (low recall) is more harmful than producing extra or slightly noisy content (lower precision).

chrF Example: Character n-gram Precision, Recall, and F-score

Reference: `witness for the past`
Hypothesis 1: `witness of the past`
Hypothesis 2: `past witness`

As in standard chrF computation, spaces are removed prior to character n-gram extraction.

Reference string: `witnessforthepast`
Hypothesis 1 string: `witnessofthepast`

Step 1: Character n-gram overlap by order

Step 1: Character n-gram overlap by order (with counts) Spaces are removed prior to character n-gram extraction.

Reference string: `witnessforthepast` (length 17)
Hypothesis 1 string: `witnessofthepast` (length 16)

For each n, let $|G_{\text{MT}}^{(n)}|$ and $|G_{\text{ref}}^{(n)}|$ be the total number of character n-grams (including multiplicity), and let $|G_{\cap}^{(n)}| = |G_{\text{MT}}^{(n)} \cap G_{\text{ref}}^{(n)}|$ be the total overlap count (multiset intersection).

n	$\lvert G_{\text{MT}}^{(n)}\rvert$	$\lvert G_{\text{ref}}^{(n)}\rvert$	$\lvert G_{\cap}^{(n)}\rvert$	chrP_n	chrR_n
1	16	17	16	$16/16 = 1.000$	$16/17 = 0.941$
2	15	16	14	$14/15 = 0.933$	$14/16 = 0.875$
3	14	15	13	$13/14 = 0.929$	$13/15 = 0.867$
4	13	14	12	$12/13 = 0.923$	$12/14 = 0.857$
5	12	13	10	$10/12 = 0.833$	$10/13 = 0.769$
6	11	12	9	$9/11 = 0.818$	$9/12 = 0.750$

Step 2: Micro-averaged precision and recall (summing counts) Micro-averaging sums the overlap counts and the total counts across all $n = 1 \dots 6$:

$$\sum_{n=1}^{6} |G_{\cap}^{(n)}| = 16 + 14 + 13 + 12 + 10 + 9 = 74,$$

$$\sum_{n=1}^{6} |G_{\text{MT}}^{(n)}| = 16{+}15{+}14{+}13{+}12{+}11 = 81, \qquad \sum_{n=1}^{6} |G_{\text{ref}}^{(n)}| = 17{+}16{+}15{+}14{+}13{+}12 = 87.$$

The micro-averaged precision and recall are therefore:

$$\text{chrP} = \frac{\sum_n |G_{\cap}^{(n)}|}{\sum_n |G_{\text{MT}}^{(n)}|} = \frac{74}{81} \approx 0.914, \qquad \text{chrR} = \frac{\sum_n |G_{\cap}^{(n)}|}{\sum_n |G_{\text{ref}}^{(n)}|} = \frac{74}{87} \approx 0.851.$$

Step 3: Weighted F-score Using $\beta = 2$:

$$\text{chrF}_2 = \frac{(1 + 2^2) \cdot \text{chrP} \cdot \text{chrR}}{2^2 \cdot \text{chrP} + \text{chrR}} = \frac{5 \cdot 0.914 \cdot 0.851}{4 \cdot 0.914 + 0.851} \approx 0.863.$$

Contrastive case Hypothesis 2 (`past witness`) preserves the same content words but changes their order substantially. While many character unigrams still overlap, higher- order character n-grams—especially those spanning word boundaries—are disrupted. This sharply reduces recall of reference n-grams and results in a much lower chrF score ($\text{chrF}_2 \approx 0.62$).

Interpretation This example illustrates how chrF:

- tolerates small lexical and morphological variation,
- penalizes missing or displaced content through recall,
- degrades more smoothly than BLEU under paraphrasing.

Contrastive example chrF thus occupies an intermediate position between word-based n-gram metrics such as BLEU and edit-distance-based metrics such as TER, combining surface overlap with greater linguistic robustness.

3.3.2 Neural Evaluation Metrics

More recent evaluation metrics leverage pretrained neural language models to capture semantic similarity beyond surface overlap. BERTScore (Zhang et al., 2020) computes similarity between contextualized token embeddings of the system output and the reference, allowing semantically similar words to receive partial credit even in the absence of exact matches.

BLEURT (Sellam et al., 2020) and COMET (Rei et al., 2020) go a step further by training neural models directly to predict human judgment scores. These metrics combine information from the source sentence, the system output, and the reference translation, and are fine-tuned on large collections of human ratings from WMT evaluation campaigns.

Neural metrics consistently outperform traditional overlap-based metrics in terms of correlation with human judgments. However, they are more computationally expensive, may be sensitive to domain and quality drift, and require careful interpretation when applied outside their training conditions.

Embedding-based metrics overcome the surface-matching limitations of BLEU/chrF.

3.3.2.1 BERTScore

BERTScore (Zhang et al., 2020) is an embedding-based evaluation metric that measures semantic similarity between a system output and a reference translation using contextualized word representations from a pretrained language model.

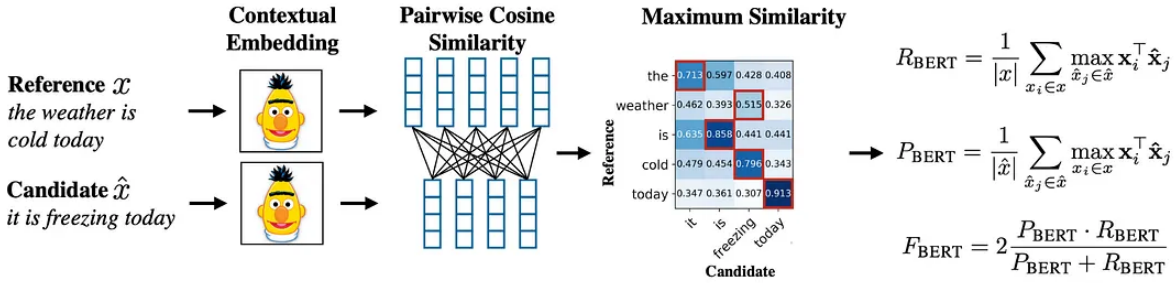


Figure 3.3: BERTScore: each token is represented by a contextual embedding, pairwise cosine similarities are computed between reference and system tokens, and each token is aligned to its most similar counterpart. Figure from `https://medium.com/data-science/bertscore-evaluating-text-generation-with-bert-beb7b3431300`.

As illustrated in Figure 3.3, BERTScore works in three steps:

1. Each word in the reference and the system output is mapped to a **contextual embedding** using a pretrained language model.
2. All words are compared using **cosine similarity**, producing a similarity matrix.
3. Each word is aligned to the *most similar* word in the other sentence.

From these alignments, BERTScore computes:

- **Recall**: how well the meaning of the reference sentence is covered by the system output.
- **Precision**: how much of the system output is semantically relevant to the reference.
- **F-score**: a single score combining precision and recall.

Because it relies on contextual embeddings rather than exact word matching, BERTScore can reward semantically equivalent translations even when the wording differs, making it more robust than surface-based metrics such as BLEU or chrF.

BERTScore Example: Semantic Matching Beyond Exact Overlap

Reference: *The boy is riding a bicycle*
System output: *The child is riding a bike*

Although several words differ lexically (*boy* vs. *child*, *bicycle* vs. *bike*), their contextual embeddings are highly similar. BERTScore therefore aligns these tokens and assigns them high cosine similarity.

- *boy* $\leftrightarrow$ *child* (high semantic similarity)
- *bicycle* $\leftrightarrow$ *bike* (near-synonyms)
- Function words such as *the* and *is* align almost perfectly

As a result, both precision and recall are high, leading to a high F_{BERT} score, even though a word-based metric like BLEU would penalize the lack of exact matches.

3.3.2.2 BLEURT

BLEURT is an automatic evaluation metric for natural language generation that is designed to correlate well with **human judgments of translation quality**. Unlike surface-based metrics such as BLEU or chrF, and unlike similarity-based metrics such as BERTScore, BLEURT is a **learned metric**: it predicts a quality score rather than computing similarity using a fixed formula.

The key idea behind BLEURT is that translation quality is not explicitly defined by hand. Instead, the metric learns what humans consider a good or bad translation based on examples of human evaluations.

All figures in this section are taken from
`https://research.google/blog/evaluating-natural-language-generation-with-bleurt/`.

Human judgments and evaluation Human evaluation is often considered the gold standard for assessing translation quality. Human annotators are typically asked to rate translations according to criteria such as adequacy or overall quality, as explained in Section 3.2.1. These judgments are expensive to obtain, but they provide valuable information about what users consider a good translation.

BLEURT is trained to *imitate* such human judgments, allowing automatic evaluation to approximate human evaluation more closely.

Pretraining BLEURT Before being trained on human judgments, BLEURT is first pretrained on large amounts of synthetic data. Starting from high-quality reference sentences, artificial errors are automatically introduced, such as word deletions, substitutions, and reordering, as shown in Figure 3.4.

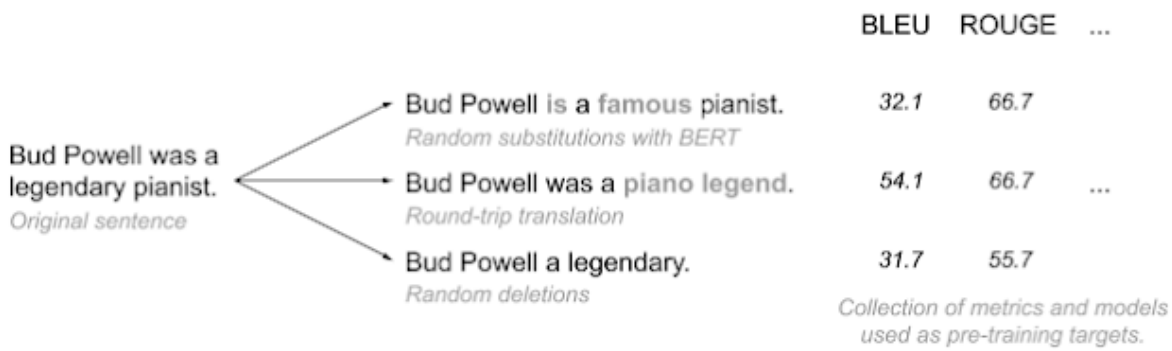


Figure 3.4: Examples of synthetic data used to pretrain BLEURT.

During this stage, the model learns to distinguish well-formed and semantically correct sentences from degraded or incorrect ones. This allows BLEURT to acquire a broad notion of language quality before seeing any human ratings.

Training BLEURT The overall training process of BLEURT is illustrated in Figure 3.5. After pretraining on synthetic data, the model is fine-tuned using human evaluation data.

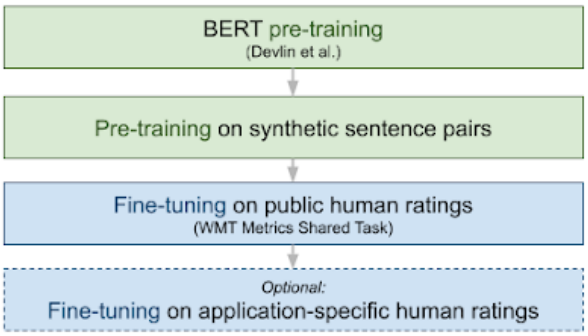


Figure 3.5: Overview of the main training steps used to build BLEURT.

During fine-tuning, the input consists of a reference translation together with a system output, and the target is a human quality score. The model learns which types of errors humans consider serious and which differences are relatively minor.

BLEURT therefore combines two sources of information:

- large-scale synthetic training data;
- smaller but high-quality human judgments.

Positioning BLEURT among evaluation metrics BLEURT can be situated among other automatic evaluation metrics as follows:

- **BLEU / chrF**: measure surface overlap between system output and reference;
- **BERTScore**: measures semantic similarity using contextual embeddings;
- **BLEURT**: predicts translation quality based on learned human judgments.

By learning directly from human evaluations, BLEURT goes one step further than similarity-based metrics and aims to approximate how humans assess translation quality. At the same time, BLEURT should be regarded as a black-box metric: it produces a score, but does not explain *why* a translation is good or bad.

BLEURT Example: Penalizing Fluent but Incorrect Translations

Reference: *The doctor prescribed the patient a new medication*
System output: *The doctor described the patient a new medication*

The system output is fluent and shares substantial lexical overlap with the reference, which may result in a relatively high BLEU or chrF score. However, the verb *described* is semantically incorrect in this context.
BLEURT has been trained on examples of subtle semantic errors and therefore assigns a lower score to this translation, reflecting the loss of adequacy as a human evaluator would perceive it.

3.3.2.3 COMET

COMET (Crosslingual Optimized Metric for Evaluation of Translation) (Rei et al., 2020) is an automatic evaluation metric for machine translation that is designed to correlate strongly with **human judgments of translation quality**. Like BLEURT, COMET is a **learned metric**: it predicts a quality score using a neural model rather than relying on surface overlap or fixed similarity formulas.

A key difference with BLEURT is that COMET explicitly incorporates the **source sentence** in addition to the reference translation and the system output. This allows COMET to assess adequacy more directly by checking whether the meaning of the source is preserved in the translation.

Human judgments and evaluation COMET is trained using human evaluation data, where annotators judge the quality of machine translations. These judgments typically reflect adequacy, fluency, or overall quality, and serve as the target signal during training.

As with BLEURT, the goal of COMET is to learn how humans assess translation quality, so that automatic evaluation scores better reflect human preferences.

Input representations COMET models typically take as input:

- the **source sentence**;
- the **system output**;
- optionally, a **reference translation**.

These inputs are encoded using pretrained multilingual language models. By including the source sentence, COMET can detect meaning shifts or omissions even when the system output is fluent and lexically similar to the reference.

Training COMET COMET is trained in a supervised setting. During training, the model receives examples consisting of a source sentence, a system output, and a reference, and learns to predict a human quality score.

Through this process, the model learns which translation errors humans consider serious, such as mistranslations or missing content, and which variations are acceptable, such as paraphrasing.

Positioning COMET among evaluation metrics COMET can be positioned alongside other automatic evaluation metrics as follows:

- **BLEU / chrF**: measure surface overlap between system output and reference;

- **BERTScore**: measures semantic similarity between system output and reference;
- **BLEURT**: predicts translation quality based on learned human judgments;
- **COMET**: predicts translation quality using the source sentence, system output, and reference.

Because it explicitly conditions on the source sentence, COMET is particularly effective at detecting adequacy errors. Like BLEURT, it should be regarded as a black-box metric: it produces a score that correlates with human judgments, but does not provide an explicit explanation for the score.

COMET Example: Using the Source Sentence to Detect Adequacy Errors

Source: *He did not attend the meeting*
Reference: *Il n'a pas assisté à la réunion*
System output: *Il a assisté à la réunion*

The system output is fluent and closely matches the reference except for the missing negation. Surface-based metrics may still assign a moderate score due to high overlap. Because COMET explicitly models the relationship between the source and the translation, it detects the contradiction introduced by dropping the negation and assigns a substantially lower score, in line with human adequacy judgments.

3.3.2.4 Which Metric Should Be Reported?

No single automatic metric fully captures translation quality. In contemporary MT research, it is therefore common practice to report multiple complementary metrics.

- **BLEU** remains widely reported for historical continuity and comparability with earlier work, especially in shared tasks and benchmarking studies.
- **chrF** is often reported alongside BLEU, particularly for morphologically rich languages, as it is less sensitive to tokenization and inflectional variation.
- **Neural metrics** such as BERTScore, BLEURT, and COMET provide better correlation with human judgments and are increasingly used as primary evaluation measures.

In recent WMT evaluations, **COMET** has become the preferred automatic metric for system ranking, while BLEU and chrF are typically retained as auxiliary metrics for transparency and comparability. When reporting results, it is good practice to clearly state which metric is used for model selection and which are reported for reference.

3.3.3 Statistical Significance Testing

Automatic evaluation metrics such as BLEU and chrF are typically reported as single-point scores. However, small numerical differences between systems may arise purely due to sampling variability in the test set rather than reflecting a genuine improvement in translation quality. Statistical significance testing is therefore essential to determine whether an observed difference is likely to be meaningful.

Following Koehn (2004) and the recommendations implemented in `sacreBLEU` (Post, 2018), we use **bootstrap resampling**, a non-parametric method that makes minimal assumptions about the distribution of metric scores.

3.3.3.1 Bootstrap Resampling

Given a test set of n sentence pairs, bootstrap resampling constructs many pseudo-testsets by sampling sentences *with replacement*. Each pseudo-testset has the same size as the original test set but may contain duplicate sentences and omit others.

The procedure is as follows:

1. Sample n sentence pairs with replacement from the original test set to create a pseudo-testset.
2. Compute the evaluation metric (e.g. BLEU or chrF) on this pseudo-testset.
3. Repeat this process many times (typically 1,000–10,000 iterations), yielding a distribution of metric scores.
4. Sort the scores and discard the lowest and highest 2.5%.

The remaining interval corresponds to a **95% confidence interval** for the metric. If confidence intervals for two systems do not substantially overlap, this provides evidence that their performance difference is not due to random variation.

3.3.3.2 Paired Bootstrap for System Comparison

When comparing two MT systems A and B, bootstrap resampling is applied in a *paired* manner: for each pseudo-testset, the same sampled sentences are used to evaluate both systems. This controls for sentence-level difficulty and ensures a fair comparison.

For each bootstrap sample, we compute the difference:

$$\Delta = \text{metric}(A) - \text{metric}(B).$$

Statistical significance is then estimated as:

$$p = \Pr(\text{metric}(A) > \text{metric}(B)),$$

i.e. the proportion of bootstrap samples for which system A outperforms system B. Under this definition, values of $p > 0.95$ indicate statistical significance at the 5% level. This is equivalent to the classical hypothesis-testing convention of reporting $p < 0.05$, but expressed as the probability that one system outperforms another under bootstrap resampling.

3.3.3.3 Interpretation and Practical Considerations

Statistical significance does not imply practical relevance: a difference may be statistically significant but too small to matter in real-world translation use. Conversely, lack of significance may simply indicate that the test set is too small to reliably detect differences.

Bootstrap resampling is most commonly applied to corpus-level metrics such as BLEU and chrF. While neural metrics such as COMET and BLEURT can also be used with bootstrap testing, their higher variance and sensitivity to domain shift require careful interpretation. For this reason, significance testing is often reported alongside traditional metrics even when neural metrics are used for primary system ranking.

3.3.4 Evaluating Evaluation Metrics

Automatic evaluation metrics are only useful insofar as they correlate with human judgments of translation quality. Since metrics differ in what aspects of quality they capture (e.g. fluency, adequacy, semantic faithfulness), their own performance must be evaluated through systematic comparison against human annotations.

For this reason, the WMT conference series organizes annual **Metrics Shared Tasks**, in which automatic metrics are assessed based on their agreement with human evaluations. In recent years, these human judgments are typically collected using **MQM**, a fine-grained error annotation framework that distinguishes error types such as mistranslation, omission, addition, and grammatical errors, that was already discussed in section 3.2.3.

3.3.4.1 Correlation with Human Judgments

Metric quality is commonly measured by computing statistical correlation between metric scores and human ratings. Depending on the evaluation setting, different correlation measures are used:

- **Pearson correlation** measures linear correlation between metric scores and human scores.
- **Spearman correlation** measures rank correlation, focusing on whether metrics correctly order systems by quality.
- **Kendall's** τ is often used in system-level ranking tasks and is robust to small score differences.

High correlation indicates that a metric tends to agree with human judgments, but it does not guarantee that the metric captures all relevant aspects of translation quality.

3.3.4.2 System-level vs. Segment-level Evaluation

Metrics can be evaluated at different granularities:

- **System-level evaluation** compares average metric scores across MT systems. Most metrics achieve higher correlation with human judgments at this level.
- **Segment-level evaluation** compares scores at the sentence level. This is a more challenging setting, as human judgments are noisier and metrics must be sensitive to fine-grained errors.

Neural metrics such as COMET and BLEURT consistently outperform traditional overlap-based metrics in both settings, although the performance gap is particularly pronounced at the segment level.

3.3.4.3 Implications for Metric Choice

Results from metric shared tasks show that no single metric is universally optimal across languages, domains, and evaluation goals. Metrics trained on news-domain data may perform less reliably on specialized or low-resource domains, and strong system-level correlation does not necessarily imply reliable sentence-level feedback.

As a result, metric choice should be guided by the evaluation context:

- Use system-level metrics (e.g. BLEU, chrF, COMET) for benchmarking and model comparison.

- Use neural, source-aware metrics (e.g. COMET) when adequacy and meaning preservation are critical.
- Treat all automatic metrics as proxies for human judgment rather than definitive measures of translation quality.

Ultimately, automatic evaluation complements but does not replace human evaluation, particularly in high-stakes or application-driven translation settings.

3.4 Hands-on Automatic Translation Evaluation

Google Colab

The implementation of the metrics and techniques in this section can be found HERE.

In this hands-on session we will see how we can send sentences to online translation engines, and how we can automatically evaluate the results, provided that we have human translated reference translations at our disposal.

3.4.1 Accessing Translation engines in Python

We can connect python to existing online translation engines by using their so-called APIs: Application Programming Interface.

We first need to install the python module `translators`, which allows easy access to several online engines. For more info on this module check `https://pypi.org/project/translators/`.

For python, we install modules with the `pip install` command. We tell Google Colab that the code cell does not concern python code, but contains a linux command by prepending it with a `!`.

```
!pip install translators --upgrade
```

To show how it works, we will use a set of Dutch sentences created as the development set of the Tatoeba English-to-Dutch parallel data.

We download the testset using `!wget`.

```
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/data/
↪  tatoeba-en-nl/dev.en
```

We then read in the files in python, in the list `sourcelines`.

```
sourcefile = "dev.en"
sourcelines=open(sourcefile,'r').readlines()
sourcelines = [l.strip() for l in sourcelines]
```

The next code block shows how we loop over the sentences of the article and use the online translation engines. We print out the results per sentence, so we can easily compare the different engines.

Note that the engines are real engines, and that it is possible that they stop offering the service of accepting sentences through an API. As far as I've tested, it seems to work for google and

bing, and only sometimes for deepl. These APIs are only intended for research and teaching, so don't overuse them!

The engines apply, by default, automatic source language identification. The target language defaults to English. As we want to translate to Dutch we have to set the target language to `nl`.

We first define a function that takes a sentence, an engine and a target language as input arguments and that strips the translated sentence of potential newline characters at the end of the line. The function also catches the case where the engine does not return anything, so the code will not crash.

```
import translators as ts
def translate_sentence(sentence, engine, targetlang):
    try:
        out = ts.translate_text(
            sentence,
            translator=engine,
            to_language=targetlang
        )
        return str(out).strip()
    except Exception:
        return "-"
```

Then we call the function on the MT engines bing and google, and keep the results in a python dictionary results, which contains one list per engine.

We limit the loop to the translation of the first 30 source lines [:30].

```
engines=['bing','google']
targetlang = 'nl'
results = {eng: [] for eng in engines}

for sentence in sourcelines[:30]:
    sentence = sentence.strip()
    if not sentence:
        continue
    print("SRC:",sentence)
    for eng in engines:
      result = translate_sentence(sentence, eng,targetlang)
      print(eng,':',result)
      results[eng].append(result)
    print("\n")
```

As such, we get a side-by-side translation of bing and google for these 30 sentences.

3.4.1.1 DeepL

In order to automatically translate with DeepL, you first need to create a DeepL account at `https://www.deepl.com/en/pro-api`, and then go to Account → API keys and limits → Create key.

Then we click on the key symbol in the left pane and add a secret key with the name `deepl`. These keys are not visible to people you share the Colab session with.

We get the values of these keys using `userdata.get('deepl')` and can use that value to get access to the DeepL web service.

```
from google.colab import userdata
auth_key = userdata.get('deepl')
```

Then we install the deepl python library.

```
!pip install deepl
```

Now we translate the first 30 lines and add them to the results dictionary

```
import deepl


translator = deepl.Translator(auth_key)

translations = translator.translate_text(sourcelines[:30], target_lang="NL")
results['deepl']=[]

for t in translations:
    print(t.text)
    results['deepl'].append(t.text)
```

We can now loop over our dictionary and source sentences again and compare the three engines

```
engines=['google','bing','deepl']
for (index,source) in enumerate(sourcelines[:30]):
  print(f'SRC : {source}')
  for engine in engines:
    print(f'{engine} : {results[engine][index]}')
  print("")
```

3.4.1.2 Save the outputs to files

Now that we've translated 30 sentences with 3 different MT engines, we store the results, so we can reuse and evaluate them later.

```
for engine, outputs in results.items():
    with open(f"tatoeba-{engine}.txt", "w") as f:
        for sent in outputs:
            f.write(sent + "\n")
```

The above made local copies. We need to copy them to a path in our Google Drive, using `!cp`.

```
!cp tatoeba-*.txt drive/MyDrive/MTATM/.../tatoeba/
```

3.4.2 Automatic metrics

In this section we'll demonstrate how to calculate several evaluation metrics.

3.4.2.1 Using MATEO (Vanroy et al., 2023)

In this hands-on section, we focus on automatic evaluation: computing quantitative scores that summarise how close a system output is to one or more reference translations.

A practical way to do this is MATEO (MAchine Translation Evaluation Online) available at `https://mateo.ivdnt.org/`, a web-based interface that lets you upload MT outputs and obtain scores from a battery of established metrics in a single, consistent evaluation pipeline. Tools like MATEO make it easier to compare systems, settings, or datasets without re-implementing evaluation code each time—while also reminding us that metric scores are proxies that must be interpreted with care.

The online version will only work for relatively small files, not for test files with thousands of sentences.

3.4.2.2 Exercise: Evaluate the three engines with MATEO

Use the `tatoeba-engine.txt` files that you just created to evaluate the engines with MATEO. Remember that you need to create a version of only 30 sentences for the source and the reference files. (You can use the `!head` command for that).

The reference file is found here: `https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/data/tatoeba-en-nl/dev.nl`

Interpret the results.

3.4.2.3 SacreBLEU: BLEU scores / chrF / TER

There is a python library that allows us to calculate BLEU scores easily. We first need to install it.

```
!pip install sacrebleu
```

Now we can use SacreBleu from within python, but we can also use it as a command on the commandline.

In Python First we need to import the metrics

```
from sacrebleu.metrics import BLEU, CHRF, TER
```

Then we need a list of references. Be aware that this should be a list of lists, as we can have multiple references per source sentence.

And of course we also need system output.

```
refs = [ # First set of references
        ['The dog bit the man.', 'It was not unexpected.', 'The man bit him
        ↪  first.'],
         # Second set of references
        ['The dog had bit the man.', 'No one was surprised.', 'The man had
        ↪  bitten the dog.'],
       ]
```

```
sys = ['The dog bit the man.', "It wasn't surprising.", 'The man had just
↪  bitten him.']
```

Then we can calculate the scores like this:

```
bleu = BLEU()
bleu.corpus_score(sys, refs)
```

```
chrf = CHRF()
chrf.corpus_score(sys, refs)
```

```
ter = TER()
ter.corpus_score(sys,refs)
```

Command line To test this, we can download some automatically translated sentences from Romanian into Portuguese, automatically transcribed from a speech file. These sentences come from an experiment at the European Parliament where three anonymous commercial companies performed speech translation on a speech from Romanian to Portuguese. It is only a very short fragment.

```
!wget
↪  https://github.com/VincentCCL/MTAT/raw/refs/heads/main/data/ep/RO_PT_AV.zip
!unzip RO_PT_AV.zip
```

This gives us the results of the automatic translation by three MT engines of the same snippet of speech in the European Parliament. It also includes a reference translation.

Now we can calculate BLEU, TER and chrF in a single command:

```
!sacrebleu RO_PT_AdinaValean_REF.txt.sent -i
↪  RO_PT_AdinaValean_A.txt.mt.tok.align -m bleu ter chrf
```

Exercise

Take the bing, google and deepl translations we made for the first 30 tatoeba sentences and calculate their BLEU, TER and chrF scores.

3.4.2.4 Significance levels for BLEU / chrF / TER

Sacrebleu also allows you to calculate whether the difference in scores is significant. In order to calculate this we need to give sacrebleu the output of (at least) two MT systems, and add the `-paired-bs` option.

```
!sacrebleu reference_file -i mt-output-file_baseline mt-output-file_system2
↪  mt-output-file_systenm3 ... -m metrics --paired-bs --format-text
```

3.4.3 BERTScore

3.4.3.1 Hugging Face access token for BERTScore

BERTScore relies on pretrained Transformer models that are downloaded from the Hugging Face Hub. To access these models, you need a Hugging Face account and a read-only access token.

If you do not yet have a Hugging Face account:

1. Go to `https://huggingface.co`
2. Create a free account (or log in if you already have one).
3. Navigate to **Access Tokens**.
4. Create a new token with role Read and copy the token (it starts with `hf_...`).

In Google Colab, this token must be stored as a secret:

1. Click the Secrets (Key) icon in the left sidebar.
2. Add a new secret with:

   ```
   Name: HF_TOKEN
   Value: your Hugging Face access token
   ```

3. Save the secret.

Once the token is stored as a secret, it is automatically made available to Python libraries in the notebook. No additional authentication code is required when running BERTScore from Python.

If the token is missing or invalid, the model download will fail. After the model has been downloaded once, it is cached and reused for the remainder of the Colab session.

To make the token available for command-line commands, execute this code:

```python
from google.colab import userdata
from huggingface_hub import login

hf_token = userdata.get("HF_TOKEN")
login(hf_token)
```

3.4.3.2 Running BERTScore

We first need to instal the appropriate module.

```
!pip install bert-score
```

And we have a bit of data in this format:

```
refs_bertscore = [
    'The dog bit the man. ||| The dog had bit the man.',
```

```
    'It was not unexpected. ||| No one was surprised.',
    'The man bit him first. ||| The man had bitten the dog.',
]

sys = ['The dog bit the man.', "It wasn't surprising.", 'The man had just bitten
↪  him.']
```

Then we can calculate BERTScore Precision, Recall and F-score with the following command. Note that you need to set the target language explicitly.

```
from bert_score import score
(P, R, F), hashname = score(sys, refs_bertscore, lang="en", return_hash=True)
print(
    f"{hashname}: P={P.mean().item():.6f} R={R.mean().item():.6f}
    ↪  F={F.mean().item():.6f}"
)
```

With the command-line on the previously downloaded files from the European Parliament:

```
!bert-score \
  -r RO_PT_AdinaValean_REF.txt.sent \
  -c RO_PT_AdinaValean_A.txt.mt.tok.align \
  --lang pt # We change the Target language to Portuguese\
  --m xlm-roberta-large
```

3.4.3.3 Exercise

Calculate BERT-score on the 30 sentences from the corpus we've translated for Bing, Google and Deepl.

3.4.4 BLEURT

First, we install:

```
!pip install -q git+https://github.com/google-research/bleurt.git
!pip install -q huggingface_hub
```

Then, we download a TF BLEURT-20 checkpoint mirror from Hugging Face.

```
from huggingface_hub import snapshot_download

ckpt_dir = snapshot_download(
    repo_id="BramVanroy/BLEURT-20",
    local_dir="bleurt-20",
    local_dir_use_symlinks=False
)
print("Checkpoint in:", ckpt_dir)
```

And this is how we run Bleurt.

```
# 3) Score
from bleurt import score
references = ["This is a test."]
candidates = ["This is the test."]

scorer = score.BleurtScorer("bleurt-20")
scores = scorer.score(references=references, candidates=candidates)
print("BLEURT mean:", sum(scores)/len(scores))
```

3.4.5 COMET

COMET Colab

To get COMET to work requires us to start a new colab session, as it is not compatible with the things we've previously installed. You can find it HERE

First we need to install some specific packages

```
!pip -q install "transformers==4.41.2" "huggingface-hub==0.23.4"
!pip -q install unbabel-comet
```

Then it is really important to **RESTART THE RUNTIME**, or you will get an error.

And here is the python code for a single sentence. Note that COMET expects a list of sentence-level dictionaries, one dictionary per sentence. So for multiple sentences, you just have multiple dictionaries in the list.

```
import torch
from comet import download_model, load_from_checkpoint

ckpt = download_model("Unbabel/wmt22-comet-da")
model = load_from_checkpoint(ckpt)

gpus = 1 if torch.cuda.is_available() else 0

data = [
    {"src": "She eats apples.", "mt": "Zij eet appels.", "ref": "Zij eet appels."}
]

out = model.predict(data, batch_size=8, gpus=gpus)
print("\n",out.system_score)
```

And here is an example on how to calculate the COMET score on the bing/google/deepl translations that we ran in Section 3.4.1.

We have stored our bing / google / deepl translations of the 30 first sentences of the Tatoeba-en-nl dataset in our google drive.

We download the full source file 'dev.en' and the full reference file 'dev.nl' (1000 sentences each)

```
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/data/
↪  tatoeba-en-nl/dev.en
```

```
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/data/
↪  tatoeba-en-nl/dev.nl
```

We use the ‘head‘ command to create new files containing only the first 30 lines.

```
!head -n 30 dev.en > dev30.en
!head -n 30 dev.nl > dev30.nl
```

Then we download a python script that calculates the COMET score.

```
!wget
↪  https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/code/comet_score_files.py
```

And then we run the script with three arguments:

1. the source file `dev30.en`
2. the mt output, which is stored on our google drive
3. the reference file `dev30.nl`

```
!python comet_score_files.py dev30.en \
/content/drive/MyDrive/MTAT/.../tatoeba-bing.txt \
dev30.nl
```

which gives us the COMET score.

3.5 Quality Estimation (QE)

Quality Estimation (QE) aims to assess the quality of machine translation output *without access to reference translations*. Instead of comparing system output to a gold-standard reference, QE models predict quality scores directly from the source sentence and its translation, optionally enriched with system- internal signals or linguistic features. QE is particularly relevant in real-world settings where references are unavailable, expensive to obtain, or impractical at scale.

QE can be formulated at different levels of granularity, including sentence- level scoring, word-level error tagging, and document-level assessment. Early approaches relied on hand-crafted features such as fluency indicators, language model scores, and alignment statistics. More recent methods leverage neural models and multilingual pre-trained representations to predict human judgments, post-editing effort (e.g. HTER), or categorical quality labels.

Despite its practical importance, QE is methodologically distinct from the reference-based evaluation approaches discussed in this chapter. It requires separate training data annotated with human quality judgments or post-editing information and introduces additional challenges related to domain transfer, calibration, and interpretability. A full treatment of QE therefore falls outside the scope of this course. We restrict our focus to reference-based automatic metrics and human-centered evaluation methods, which are sufficient to understand and reproduce standard MT evaluation practices in the research literature.

3.6 Task-Based Evaluation

While automatic metrics and human judgment scores provide useful aggregate signals, they do not always reflect how MT systems perform in real-world usage. **Task-based evaluation** assesses MT quality indirectly by measuring how well users can complete downstream tasks using MT output. This type of evaluation is particularly relevant in professional translation workflows and human–computer interaction settings.

3.6.1 Post-editing Productivity

Post-editing productivity is one of the most widely used task-based evaluation methods in professional MT settings. In this paradigm, translators are asked to post-edit MT output to an acceptable quality level, and their effort is measured.

Common productivity measures include:

- **Editing time**: the time required to post-edit a sentence or document.
- **Number of edits**: typically measured using Translation Edit Rate (TER or HTER), capturing the amount of modification needed to reach the final version.
- **Translation throughput**: often expressed as words per hour, combining speed and volume.

Higher MT quality generally leads to faster post-editing, fewer edits, and higher throughput. However, productivity gains are not always linear: very fluent but semantically incorrect translations may slow down post-editing due to increased cognitive effort and error detection costs.

3.6.2 Content Understanding

Another form of task-based evaluation focuses on **content understanding** rather than translation quality per se. In this setup, monolingual users are asked to answer comprehension questions based solely on MT output.

This evaluation paradigm is particularly relevant for scenarios such as:

- information access across language barriers,
- gisting and document triage,
- crisis response and humanitarian applications.

Although difficult to design—requiring carefully constructed questions and controlled experimental conditions—content understanding evaluations capture whether MT output is *useful* for end users, even when it is not perfectly fluent or stylistically polished.

3.7 Bias and Ethical Considerations

Evaluation is not value-neutral: MT systems can reflect and amplify societal biases present in their training data. These biases may manifest in systematic errors that are not captured by standard quality metrics but have serious ethical implications.

Prates et al. (2019) demonstrate gender stereotyping in MT systems when translating from gender-neutral languages such as Hungarian into English:

Hungarian (gender neutral)	English MT output
ő egy ápoló	she is a nurse
ő egy tudós	he is a scientist
ő egy vezérigazgató	he is a CEO
ő egy esküvőszervező	she is a wedding organizer

These examples illustrate how MT systems may assign gendered pronouns based on occupational stereotypes rather than linguistic evidence. Such behavior can lead to:

- systematic amplification of gender and social biases;
- unfair or misleading translations in sensitive domains such as healthcare, law, and immigration;
- erosion of user trust in MT systems.

Addressing these issues requires evaluation practices that go beyond aggregate quality scores. Possible mitigation strategies include bias-aware evaluation sets, explicit uncertainty modeling, confidence estimation, and abstention mechanisms when the system lacks sufficient evidence to make a safe prediction.

3.8 Summary

Machine translation evaluation is challenging because translation is not a single-answer task: multiple outputs can be equally acceptable, and quality depends on both meaning preservation and target-language naturalness. This chapter surveyed the main evaluation paradigms used in research and in shared tasks such as WMT.

Human evaluation remains the most reliable approach. Rating-based methods (adequacy and fluency) are intuitive but suffer from ordinal-scale limitations, low inter-rater agreement, and systematic rater effects, which motivates careful rater training and score normalization. Ranking-based evaluation reduces some scale issues by eliciting relative preferences, while MQM provides fine-grained, diagnostic insight into error types and severity at higher cost. Post-editing evaluation connects MT quality directly to professional workflow impact by measuring editing effort (e.g. HTER).

Automatic metrics provide fast, reproducible proxies for human judgment. Surface-oriented metrics such as WER, BLEU, chrF, and TER/HTER quantify overlap or edit operations, but they cannot reliably distinguish meaning-preserving paraphrases from genuine errors. Neural metrics address this limitation by capturing semantic similarity and by learning from human ratings: BERTScore measures embedding-based similarity, BLEURT predicts quality from output–reference pairs, and COMET further incorporates the source sentence to better detect adequacy errors. In practice, it is common to report multiple metrics (e.g. BLEU and chrF for comparability, plus COMET or BLEURT for closer alignment with human judgments).

Because metric differences can be small and test sets finite, significance testing is essential. Bootstrap resampling provides confidence intervals and paired comparisons that help determine whether an observed improvement is likely to reflect a real quality gain rather than sampling noise.

Finally, evaluation must go beyond aggregate scores. MT systems can amplify societal biases and may cause harm in sensitive domains. Responsible evaluation therefore includes task-based measures of usefulness, bias-aware test cases, and careful interpretation of both human and

automatic scores.

Chapter 4

Recurrent Neural Networks Language Modeling

4.1 What is a Recurrent Neural Network (RNN)

For many translation and language tasks, **order matters**. Recurrent Neural Networks (RNNs) are neural networks that are designed to work with **sequences**.

An RNN processes a sentence one word at a time. At each step it takes two inputs:

- the current word (or its vector),
- a memory of what it has seen so far (the **hidden state**).

It then produces:

- an updated hidden state,
- optionally an output (for example, a next word prediction).

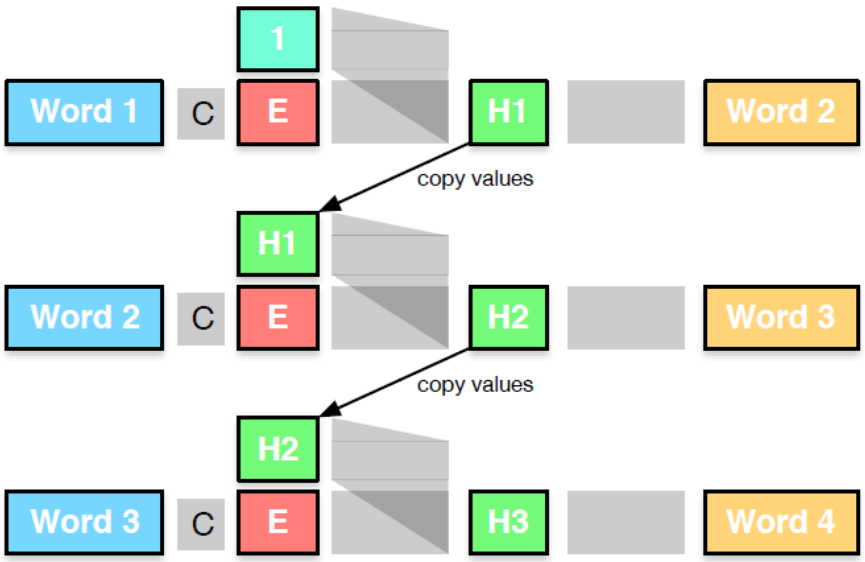


Figure 4.1: A simple recurrent neural network unrolled over time (from Koehn, 2020)

Figure 4.1 shows the same RNN cell applied to each word in the sequence. The green boxes (`H1, H2, H3`) are the hidden states that pass information from one time step to the next: they function as a kind of **context memory**. If there are only three words in the input sentence `H3` is the sentence representation.

Compared to other neural network types, RNNs have two main advantages for language data:

- **They can take word order into account.** Because the hidden state is passed along the sentence, the network knows not only which words appear, but also *in which order* they appeared.
- **They can handle variable-length sentences.** We do not need every sentence to have the same number of words; the RNN can simply run longer or shorter.

In practice, more advanced variants such as Long Short-Term Memories (LSTMs) and Gated Recurrent Units (GRUs) are often used, but the basic idea is the same: a network that reads a sequence step by step and remembers what it has seen.

4.2 Language Modeling

A *language model* (LM) is a system that learns how likely certain word sequences are in a language. In simple terms, a language model tries to answer:

> *If I have seen these words so far, what word is likely to come next?*

This idea is surprisingly powerful. If a model has seen many examples of English sentences, it may learn patterns such as:

- after `she likes`, the word `apples` is common;
- after `you are a`, words like `student` or `teacher` are likely;
- after `<sos>` (start of sentence), pronouns such as `i`, `you`, or `she` often appear.

A language model is therefore a kind of **predictive engine**. Given a partial sentence such as: `she likes` the model estimates which word(s) could reasonably follow. This estimation is not based on grammar rules but on patterns it has learned from data.

Most practically, a language model gives a score to every possible continuation of a sentence. For example, it should consider `she likes apples` more natural than: `she likes teachers` if the training data contains many examples of people liking apples and none of people liking teachers. The model does not "understand" these sentences in a human sense, but it recognises patterns from its observations.

Although we train the language model only on monolingual text, it plays a role in translation in several ways:

- A translation system must produce **fluent and natural** target-language sentences. A language model helps identify likely continuations.
- Encoder–decoder NMT systems can be viewed as combining:
  - an *encoder* that reads the source sentence, and
  - a *decoder* that acts as a **conditional language model**: it predicts the next target word given the previous ones and given the source.

Understanding simple language models helps grasp how modern MT systems generate words step-by-step during translation.

4.3 Implementing a toy RNN language model

Modern AI systems, such as large language models (LLMs), are extremely complex and trained on enormous datasets. However, their core behaviour still follows the same simple principle: *Generate the next word based on previous words.*

By building a small neural language model ourselves, using only a few lines of Python and a tiny dataset, we can clearly see:

- how text is converted into numbers;
- how a recurrent neural network processes a sentence from left to right;
- how the model learns to prefer some continuations over others;
- how the system generates text word by word.

These concepts carry directly over to neural machine translation, where the decoder predicts each target word in exactly the same incremental way.

In this section we build a small but complete *word-level recurrent neural language model* (RNN-LM) using Keras. The design mirrors the preprocessing pipeline used for a simple RNN-based NMT system:

- we work at the word level;
- we add `<sos>` and `<eos>` markers;
- we use `Tokenizer` from `keras.preprocessing`;
- we pad sequences and use `mask_zero=True` in the embedding;
- we train an RNN to predict the next word at each time step.

"Google Colab"

You can find the associated Google Colab session HERE.

As a running example we will track the sentence: `she likes apples` through all preprocessing steps.

We need to import the necessary modules.

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

4.3.1 Step 1: Defining a Toy Corpus

We start from a tiny toy corpus that combines simple copula sentences with a few `likes apples` sentences:

```
src_raw = [
    "i am a student",
    "i am a teacher",
    "you are a student",
    "you are a teacher",
    "she is a student",
    "he is a teacher",
    "we are students",
    "they are teachers",
    "i like apples",
    "you like apples",
    "she likes apples",
    "he likes apples",
]
```

On its own, this corpus is much too small for any serious language modelling, but it is ideal for illustrating all steps transparently.

4.3.2 Step 2: Adding <sos> and <eos>

Neural sequence models usually operate on sequences that are explicitly marked for their beginning and end. We therefore prepend <sos> (start-of-sentence) and append <eos> (end-of-sentence) to every sentence:

```
src = [f"<sos> {s} <eos>" for s in src_raw]
```

For the running example, this yields:

"she likes apples" ⟶ "<sos> she likes apples <eos>".

Including these markers in the tokenisation step ensures they become regular tokens with their own indices in the vocabulary.

So the full corpus becomes:

```
['<sos> i am a student <eos>',
 '<sos> i am a teacher <eos>',
 '<sos> you are a student <eos>',
 '<sos> you are a teacher <eos>',
 '<sos> she is a student <eos>',
 '<sos> he is a teacher <eos>',
 '<sos> we are students <eos>',
 '<sos> they are teachers <eos>',
 '<sos> i like apples <eos>',
 '<sos> you like apples <eos>',
 '<sos> she likes apples <eos>',
 '<sos> he likes apples <eos>']
```

4.3.3 Step 3: Tokenisation and Vocabulary

We now build a word-level vocabulary using Keras' Tokenizer. We disable the default filters so that no punctuation or special tokens (like <sos> and <eos>) are removed. We also set all tokens

to lowercase, setting the `lower` argument to `True`.

```
tokenizer_src = Tokenizer(
    filters="",   # don't strip punctuation, we want tokens as-is
    lower=True
)
tokenizer_src.fit_on_texts(src)
```

This is the step where the `Tokenizer` reads the entire corpus and *builds the vocabulary*. More precisely, it:

1. splits all sentences into lowercased individual words;
2. collects every unique word that appears anywhere in the corpus;
3. counts word frequencies;
4. assigns an integer ID to each word (the most frequent word receives ID 1);
5. stores dictionary internally: `word_index`: maps words to their integer IDs.

After this step, the tokenizer "knows" how to convert any sentence made of known words into a sequence of integers.

We can store the mapping in `src_word_index`:

```
src_word_index=tokenizer_src.word_index
```

which contains

```
{'<sos>': 1,
 '<eos>': 2,
 'a': 3,
 'are': 4,
 'apples': 5,
 'i': 6,
 'student': 7,
 'teacher': 8,
 'you': 9,
 'am': 10,
 'she': 11,
 'is': 12,
 'he': 13,
 'like': 14,
 'likes': 15,
 'we': 16,
 'students': 17,
 'they': 18,
 'teachers': 19}
```

So converting the example sentence: `<sos> she likes apples <eos>` → `[1, 11, 15, 5, 2]`

Now we convert all our sentences:

```
src_sequences = tokenizer_src.texts_to_sequences(src)
```

resulting in

```
[[1, 6, 10, 3, 7, 2],
 [1, 6, 10, 3, 8, 2],
 [1, 9, 4, 3, 7, 2],
 [1, 9, 4, 3, 8, 2],
 [1, 11, 12, 3, 7, 2],
 [1, 13, 12, 3, 8, 2],
 [1, 16, 4, 17, 2],
 [1, 18, 4, 19, 2],
 [1, 6, 14, 5, 2],
 [1, 9, 14, 5, 2],
 [1, 11, 15, 5, 2],
 [1, 13, 15, 5, 2]]
```

We also need the reverse table, to go from numbers back to words, and get the id numbers of the special tokens.

```
# id → word mapping (inverse vocabulary)
src_index_word = {idx: word for word, idx in tokenizer_src.word_index.items()}
src_index_word[0] = "<pad>"

sos_id = src_word_index["<sos>"]
eos_id = src_word_index["<eos>"]
```

4.3.4 Step 4: Padding to a Fixed Length

Neural networks process data in *batches*: several examples are grouped together and passed through the model at the same time. This is much faster and more stable than processing one sentence at a time. See section 4.4 for a more elaborate discussion on batches.

However, a batch must be a *rectangular* matrix: every example must have the same number of time steps. Natural sentences do not satisfy this: some sentences have three words, others four or five. After converting words to integer IDs, we obtain sequences of different lengths.

Such "ragged" sequences cannot form a proper matrix, and therefore cannot be processed in parallel. To solve this, we make all sequences the same length by adding padding tokens (value 0) at the end of shorter sentences:

```
[1, 10, 19,  6] ⟶ [1, 10, 19, 6, 0]
[2, 7, 14]      ⟶ [2, 7, 14, 0, 0]
```

Padding turns the variable-length sentences into a fixed-shape matrix suitable for batch processing. We then instruct the model to ignore the padding tokens using `mask_zero=True` in the embedding layer.

We find the length of the longest sentence and pad the others with zeros on the right (*post-padding*). Index 0 is reserved as the special padding ID.

```
max_src_len = max(len(seq) for seq in src_sequences)

src_sequences_padded = pad_sequences(
    src_sequences,
    maxlen=max_src_len,
    padding="post"  # pad with 0s on the right
)
```

This should result in

```
array([[ 1,  6, 10,  3,  7,  2],
       [ 1,  6, 10,  3,  8,  2],
       [ 1,  9,  4,  3,  7,  2],
       [ 1,  9,  4,  3,  8,  2],
       [ 1, 11, 12,  3,  7,  2],
       [ 1, 13, 12,  3,  8,  2],
       [ 1, 16,  4, 17,  2,  0],
       [ 1, 18,  4, 19,  2,  0],
       [ 1,  6, 14,  5,  2,  0],
       [ 1,  9, 14,  5,  2,  0],
       [ 1, 11, 15,  5,  2,  0],
       [ 1, 13, 15,  5,  2,  0]], dtype=int32)
```

4.3.5 Step 5: Building Input and Target Sequences

A language model learns to predict the *next* word given the previous context. For each padded sentence we create:

- an input sequence X containing all tokens except the last;
- a target sequence Y (what the model has to learn to predict) containing all tokens except the first.

Formally, for each sentence:

$$X = [w_0, w_1, \ldots, w_{T-2}], \qquad Y = [w_1, w_2, \ldots, w_{T-1}].$$

In code:

```
X_lm = src_sequences_padded[:, :-1]  # drop last token
y_lm = src_sequences_padded[:, 1:]   # drop first token

context_len = X_lm.shape[1]   # e.g. 5
num_src_tokens = len(tokenizer_src.word_index) + 1  # +1 for padding token 0
```

This results in X_{lm} =

```
array([[ 1,  6, 10,  3,  7],
       [ 1,  6, 10,  3,  8],
       [ 1,  9,  4,  3,  7],
       [ 1,  9,  4,  3,  8],
       [ 1, 11, 12,  3,  7],
```

```
       [ 1, 13, 12,  3,  8],
       [ 1, 16,  4, 17,  2],
       [ 1, 18,  4, 19,  2],
       [ 1,  6, 14,  5,  2],
       [ 1,  9, 14,  5,  2],
       [ 1, 11, 15,  5,  2],
       [ 1, 13, 15,  5,  2]], dtype=int32)
```

and Y_{lm} =

```
array([[ 6, 10,  3,  7,  2],
   [ 6, 10,  3,  8,  2],
   [ 9,  4,  3,  7,  2],
   [ 9,  4,  3,  8,  2],
   [11, 12,  3,  7,  2],
   [13, 12,  3,  8,  2],
   [16,  4, 17,  2,  0],
   [18,  4, 19,  2,  0],
   [ 6, 14,  5,  2,  0],
   [ 9, 14,  5,  2,  0],
   [11, 15,  5,  2,  0],
   [13, 15,  5,  2,  0]], dtype=int32)
```

At each position in a sentence, the model tries to guess which word is likely to come next, as shown in Table 4.1.

Input sequence seen so far	**Likely next word(s)**
<sos>	she, i, you, ...
<sos> she	likes, is
<sos> she likes	apples
<sos> she likes apples	<eos>

Table 4.1: Incremental next-word prediction in the RNN language model.

In short, the language model gradually learns to guess the next word based on the words it has seen so far. This is visualized in Figure 4.2.

Teacher Forcing in the Language Model

The way we train our language model uses *teacher forcing*. By shifting each sentence to create:

$$X = [w_0, w_1, \ldots, w_{T-1}], \qquad Y = [w_1, w_2, \ldots, w_T],$$

the model always sees the *correct* previous word and learns to predict the next one. At no point does it use its own predictions as input during training.
This is the same mechanism we will use later in the decoder of the NMT model.

4.3.6 Step 6: Defining the RNN Language Model

We now define the neural architecture. It consists of:

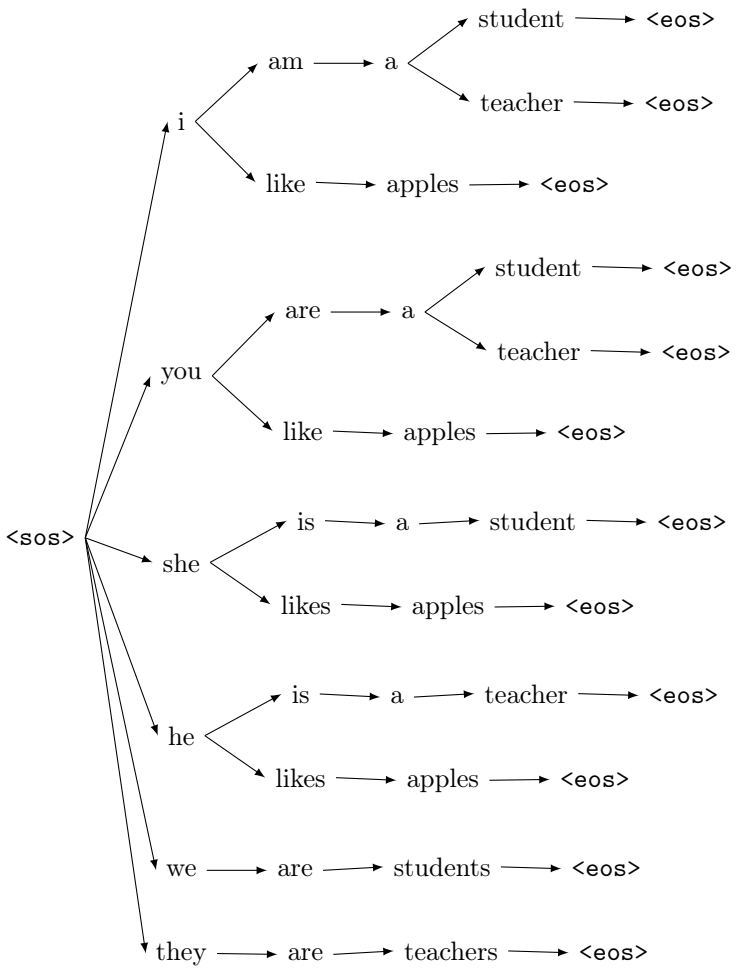


Figure 4.2: Trie built from the toy corpus with sentences such as `<sos> i am a student <eos>` and `<sos> she likes apples <eos>`.

1. **Input layer**. Shape: `(context_len,)` – a fixed-length sequence of word IDs.
2. **Embedding layer**. Converts each word ID into a dense vector. E.g., `13` (for "eat") might become `[0.1, -0.3, 0.7, ...]`. The dimensionality of this vector (length of the vector) is set in `output_dim`. We also set `mask_zero=True` in the embedding so that padding tokens (index 0) are ignored inside the RNN.
3. **SimpleRNN layer** with `return_sequences=True`. Processes the sequence step by step, maintaining a hidden state that summarizes all previous words. Because `return_sequences=True`, the RNN outputs a hidden state for every time step.
4. **Dense layer with softmax over the vocabulary**. For every time step, outputs a vector of `length num_src_tokens`, with probabilities for each possible next word.

```
embed_dim = 64
hidden_size = 128

inputs = keras.Input(shape=(context_len,), dtype="int32")

x = layers.Embedding(
        input_dim=num_src_tokens,
```

```
        output_dim=embed_dim,
        mask_zero=True    # ignore padding index 0
)(inputs)

x = layers.SimpleRNN(
        hidden_size,
        return_sequences=True
)(x)

outputs = layers.Dense(
        num_src_tokens,
        activation="softmax"
)(x)

model = keras.Model(inputs, outputs)
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.01),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)

model.summary()
```

Conceptually, the model processes a sentence one word at a time, from left to right:

- At each position it looks at two things:
  - the current word (in its embedded, vector form), and
  - its internal memory of everything it has seen so far (the *hidden state*).
- It combines these to update its hidden state: this is the model's evolving *representation* of the sentence prefix.
- A final layer then turns this hidden state into a set of scores for all words in the vocabulary.
- After applying a softmax, these scores become something like "preferences" or probabilities for each possible next word.

In more informal terms: at each step the RNN reads the next word, updates its memory of the sentence so far, and then asks:

> *Given everything I have seen up to this point, which word would most naturally come next?*

A visualisation of the network, as generated with this code, is presented in Figure 4.3

```
from tensorflow.keras.utils import plot_model

plot_model(
    model,
    to_file="model.png",
    show_shapes=True,
    show_layer_names=True,
)
```

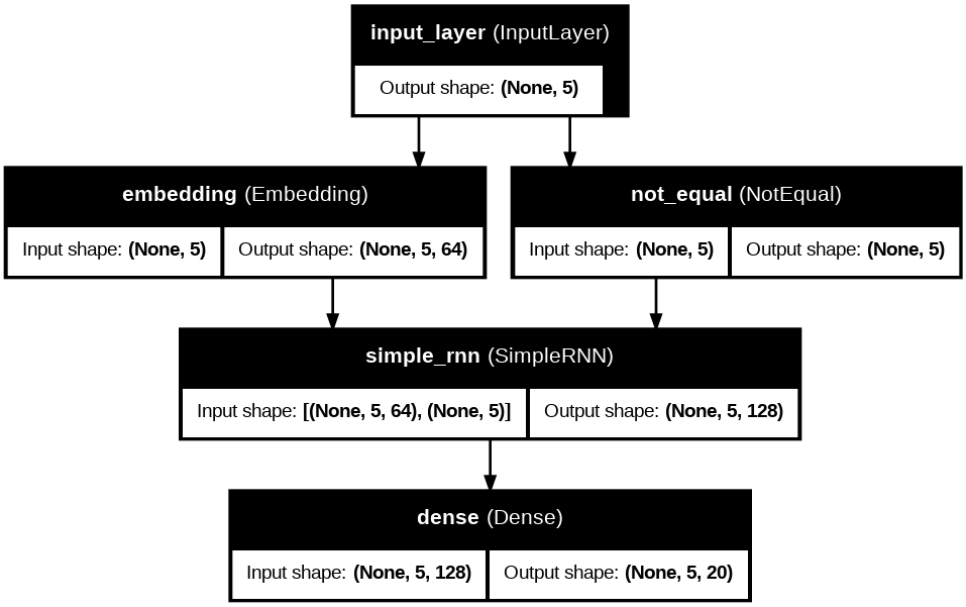


Figure 4.3: Visualisation of the RNN LM according to Keras

Understanding the Keras Model Diagram

The diagram generated by Keras shows the structure of our recurrent language model. Each box represents a layer, and each arrow shows how data flows from one layer to the next during processing. Below we explain each component in turn.

Input layer The input layer receives a batch of sentences, where each sentence is a sequence of five integer token IDs. The shape `(None, 5)` indicates that the batch size is flexible (`None`), and each sentence always has length 5 after padding.

Embedding layer The embedding layer converts each integer token into a dense 64-dimensional vector. After this layer, the model no longer works with word IDs but with continuous numerical representations. The output shape `(None, 5, 64)` means: for each of the 5 positions, we now have a 64-element embedding vector.

Masking (`NotEqual`) layer This layer appears automatically because we used `mask_zero=True` in the embedding. Its only purpose is to detect which positions in the input are padding tokens (value 0). It outputs a boolean mask such as:

$$[1, 10, 19, 18, 6] \rightarrow [\text{True}, \text{True}, \text{True}, \text{True}, \text{True}],$$

and

$$[1, 10, 19, 18, 0] \rightarrow [\text{True}, \text{True}, \text{True}, \text{True}, \text{False}].$$

The RNN uses this mask to avoid updating its state on padded positions.

SimpleRNN layer The recurrent layer processes the sentence one word at a time. At each position, it reads the corresponding embedding vector and updates its hidden state (size 128). Because we set `return_sequences=True`, the layer produces one output vector per time step, giving the output shape `(None, 5, 128)`. The RNN also receives the mask from the previous layer so that it can ignore padding.

Dense (output) layer The dense layer converts each RNN output vector into a score for every word in the vocabulary (here, 20 words including `<sos>` and `<eos>`). The resulting shape

`(None, 5, 20)` means that the model predicts a distribution over the 20 vocabulary items at each of the 5 positions. After the softmax (applied inside the loss function), these scores become the model's guesses for the next word.

The diagram visualises the flow:

token IDs → embeddings → RNN → vocabulary predictions

with masking handled automatically. This is the complete computation performed by our simple recurrent language model.

4.3.7 Step 7: Training the Model

We can now train the model by providing `X_lm` as input and `y_lm` as the expected output. The loss is computed at each time step. The goal is to (automatically) adjust weights so that the predicted next word matches the true next word as often as possible.

We call:

```
model.fit(X_lm, y_lm, ...)
```

- `X_lm` is shape `(num_sentences, context_len)`
- `y_lm` is shape `(num_sentences, context_len)`

At each time step, the loss compares:

- Predicted distribution over words
- True next word ID from `y_lm`

Over many epochs, the model learns patterns like:

- After `<sos> you are a` → `student` or `teacher`
- After `she` → `is` or `likes` After a content word like `apples` at the end of a sentence → `<eos>`

```
history = model.fit(
    X_lm,
    y_lm,
    batch_size=16,
    epochs=100,
    verbose=1
)

context_len = X_lm.shape[1]   # e.g. 5
num_src_tokens = len(tokenizer_src.word_index) + 1  # +1 for padding token 0
```

On this tiny corpus the model quickly overfits, but that is acceptable here: our goal is to understand the mechanics, not to build a robust LM.

4.3.8 Step 8: Generate text (sampling)

We want to use the trained model to produce new sequences, starting from a seed like `she likes`.

1. **Prepare the seed** `<sos> she likes`. We convert to IDs and, if needed, pad or truncate to length `context_len`.

   During training we used *post-padding*, adding zeros at the end of sentences, because all input positions were aligned across the batch. During generation, however, we may start from a short seed sentence. In that case we use *pre-padding* so that the most recent words are right-aligned and always occupy the final positions of the fixed-length context window. This ensures that the RNN processes the seed in the same relative positions as during training.

   Example:

   ```
   "<sos> she likes" → [1, 11, 15]
   → padded/truncated to length 5: [0, 0, 1, 11, 15]
   ```

2. **Feed into the model**. Give the whole context sequence to the model. The model outputs predictions for each position, but we only look at the last time step, corresponding to the last word `likes`.

3. **Sample the next word**. The last time step gives a probability distribution over the vocabulary:

   ```
   P(<sos> | context),
   P(they | context),
   P(eat | context),
   P(apples | context),
   P(teacher | context),
   P(<eos> | context),
   ...
   ```

   We sample one ID from this distribution (possibly using a temperature parameter to control randomness).

   Suppose it picks `apples`.

4. **Append and repeat** Add `apples` to the sentence. Slide the context window: we now use the last `context_len` tokens as the new input sequence. Ask the model again for the next word (maybe now it predicts `<eos>`). Stop when we hit `<eos>` or when we generated a maximum number of tokens.

So for `she likes` you might get:

- Input seed: `she likes`
- Model generates: `she likes apples <eos>`
- Final printed sentence: `she likes apples`

because we cut off everything after `<eos>` before printing.

We use *temperature* to control randomness: temperatures < 1 make the distribution sharper (more deterministic), while temperatures > 1 make it flatter (more random).

```python
def sample_next_token(probs, temperature=1.0):
    """Sample a token id from a probability distribution with temperature."""
    probs = np.asarray(probs).astype("float64")
```

```
    probs = np.maximum(probs, 1e-8)
    probs = np.log(probs) / temperature
    probs = np.exp(probs)
    probs = probs / np.sum(probs)
    return np.random.choice(len(probs), p=probs)

def generate_from_lm(seed_text, num_steps=10, temperature=0.8):
    """
    seed_text: string without <sos>/<eos>, e.g. "i like"
    num_steps: how many NEW tokens to generate
    """
    # Add <sos> so the model sees a proper start
    full_seed = f"<sos> {seed_text}"
    seed_seq = tokenizer_src.texts_to_sequences([full_seed])[0]

    # Start with at least context_len tokens (left-pad with 0 if needed)
    if len(seed_seq) < context_len:
        seed_seq = [0] * (context_len - len(seed_seq)) + seed_seq
    else:
        seed_seq = seed_seq[-context_len:]

    generated_ids = seed_seq[:]   # we keep the entire sequence of ids

    for _ in range(num_steps):
        # Use the last context_len ids as input
        context = np.array(generated_ids[-context_len:], dtype="int32")[None, :]  #
        ↪  (1, context_len)

        # Model outputs probs for each time step; we only need the LAST one
        preds = model.predict(context, verbose=0)[0]      # (context_len, vocab)
        next_probs = preds[-1]                            # distribution at last
        ↪  step

        next_id = sample_next_token(next_probs, temperature=temperature)
        generated_ids.append(next_id)

        # Stop early if we hit <eos>
        if next_id == eos_id:
          break

    id_to_word = {i: w for w, i in src_word_index.items()}
    id_to_word[0] = "<pad>"

    words = [id_to_word[i] for i in generated_ids if i != 0]

    # Drop initial <sos> in final display
    if words and words[0] == "<sos>":
      words = words[1:]

    # Cut at first <eos> if present
    if "<eos>" in words:
      eos_pos = words.index("<eos>")
      words = words[:eos_pos]  # everything before <eos>

    return " ".join(words)
```

And now we can try it.

```
print(generate_from_lm("she likes", num_steps=5, temperature=0.8))
print(generate_from_lm("i am", num_steps=5, temperature=0.8))
```

On this toy corpus, typical generations might be:

```
she likes apples
i am a student
```

reflecting patterns that the model has learned during training.

Temperature in Language Model Sampling

When the model generates text, it does not choose the next word deterministically. Instead, at each time step it produces a probability distribution over all possible next words. The *temperature* parameter controls how "confident" or "creative" the model is when sampling from this distribution.

Low temperature (e.g. 0.5) A low temperature makes the distribution sharper: the most likely words become even more likely. The model behaves conservatively and tends to choose safe, predictable continuations. This often leads to more repetitive output.

High temperature (e.g. 1.2) A high temperature makes the distribution flatter: less likely words become more accessible. The model behaves more creatively, occasionally choosing unexpected or unusual continuations. The output can be more varied, but also more error-prone.

Temperature of 1.0 A temperature of 1.0 means "no change" to the model's internal probabilities. This corresponds to the model's natural behaviour.

Intuition You can think of temperature as a creativity dial:

low temperature = safe choices, high temperature = more adventurous choices.

4.4 A Short Note on Batches

Before we train our language model, we need to understand a practical concept used in almost all neural network training: *batches*. This is not a linguistic idea but a computational one, and we keep the explanation informal.

4.4.1 Why not train on one sentence at a time?

Neural networks learn by adjusting their internal weights based on errors. If we showed the model only one sentence at a time, learning would be:

- very slow (the network updates after every single example),
- unstable (each tiny example may push the model in a different direction),
- inefficient on modern hardware.

4.4.2 Why not train on the whole dataset at once?

If we put all sentences into the model at the same time, learning would be:

- too memory-intensive, as all the sentences would need to be kept in memory,
- slow to update (the model would wait to see *all* data before making a single learning step),
- less flexible (we cannot shuffle or reorder the data as easily).

Batches: the practical compromise

Instead of using one sentence at a time or the entire dataset, we divide the data into **batches**. A batch is simply a small group of examples that the model processes together. For example:

$$\text{Batch size } = 16 \quad \Rightarrow \quad \text{the model processes 16 sentences at once.}$$

This has several advantages:

- the model can use parallel computation efficiently;
- the training signal becomes more stable than using one example at a time;
- memory usage stays manageable.

What happens during training?

Training proceeds in cycles:

1. Take the next batch of sentences (for example, 16 of them).
2. Run them through the model to obtain predictions.
3. Compare the predictions with the correct next words in each sentence.
4. Update the model's weights so that future predictions improve.

When all batches have been processed once, we call this an **epoch**. During training, we typically run several epochs.

4.5 RNN Language Modeling for Larger Texts

Google Colab for larger RNNs

There is a Google Colab session available HERE to test this on a larger text.

All the RNN LM code together in a python script can be downloaded and run.

```
!wget
↪ https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/code/rnn_lm.py
```

We first import this module and train it on a larger text, e.g. a file from gutenberg.org

```
import rnn_lm
import nltk
nltk.download('punkt_tab')

# Train on a local text file with one sentence per line
rnn_lm.train("https://www.gutenberg.org/cache/epub/28553/pg28553.txt",
             epochs=3)
```

And now we can sample from the corpus.

```
print(rnn_lm.sample("how does", num_steps=10, temperature=0.8))
print(rnn_lm.sample("the machine", num_steps=10, temperature=0.5))
```

Exercises: Larger Texts and Sampling

1. **Train on a different text (dataset swap).** Choose a different public-domain text (e.g. another Gutenberg book, a Wikipedia excerpt saved as a text file, or a collection of short stories). Train the language model for 3 epochs and sample from it.

   (a) Download or link to a new text file (URL or local file).

   (b) Train the LM for `epochs=3`.

   (c) Sample 3 continuations for each of these prompts: `"once upon"`, `"in the"`, `"he said"`.

   Deliverable: paste the three prompts and the generated continuations.

2. **Prompt sensitivity (same model, different seed).** Using the *same trained model*, compare generations from prompts that differ by only one word.

   Example prompt pairs:

   - `"she was"` vs. `"he was"`
   - `"in the house"` vs. `"in the street"`
   - `"i think"` vs. `"i know"`

   For each pair, generate 2 samples with the same temperature (e.g. `0.8`).

   Question: do the continuations diverge immediately, or only after a few words?

3. **Temperature experiment (sampling behaviour).** Train (or reuse) one model and generate continuations from the same prompt with different temperatures.

   Use one prompt, e.g. `"the machine"` (or choose your own), and generate output with:

   $$T \in \{0.2,\ 0.5,\ 0.8,\ 1.2\}.$$

   Deliverable: show the four outputs and answer:

   - Which temperature produces the most repetitive text?
   - Which temperature produces the most surprising/unpredictable text?
   - Which temperature gives the most "readable" continuation?

4. **Sampling diversity: generate multiple continuations.** Fix one prompt and one temperature (e.g. `"how does"`, `T=0.8`). Generate 5 different continuations by calling `sample()` five times.

   Question: do you get genuinely different continuations, or does the model keep returning very similar phrases? Give one plausible reason based on the training data.

Chapter 5

Neural Machine Translation with RNNs

Google Colab for RNN-MT

There is a Google Colab session available HERE, which implements the steps in this section.

In chapter 4 we built a recurrent language model (RNN-LM) that predicts the next word in a sentence given all previous words. Neural machine translation (NMT) with an *encoder–decoder* RNN introduces only one essential new idea:

- An **encoder RNN** reads the *source* sentence (e.g. English) and summarises it in its final hidden state.
- A **decoder RNN** is simply a *conditional language model*: it generates the *target* sentence (e.g. Dutch), one word at a time, starting from the encoder's final state.

Architecturally, the decoder behaves just like our RNN language model: it takes previous target words as input and predicts the next one. The only new element is that its initial hidden state comes from the encoder instead of being a fixed vector.

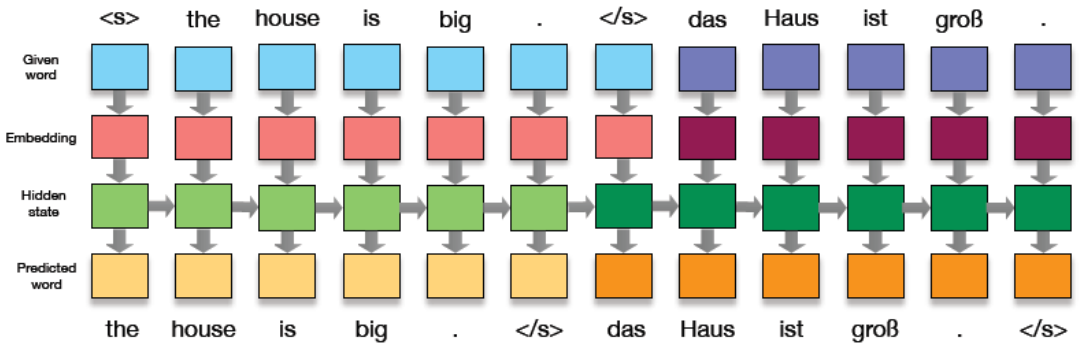


Figure 5.1: Sequence-to-sequence encoder-decoder model: Extending the language model, we concatenate the English input sentence the house is big with the German output sentence das Haus ist groß. The first dark green box (after processing the end-of-sentence token </s>) contains the embedding of the entire input sentence. Figure from Koehn (2020).

5.1 Implementation: Toy Example

To keep things transparent, we re-use the small toy corpus and add Dutch translations. Each English sentence has a Dutch counterpart.

```
src_raw = [
    "i am a student",
    "i am a teacher",
    "you are a student",
    "you are a teacher",
    "she is a student",
    "he is a teacher",
    "we are students",
    "they are teachers",
    "i like apples",
    "you like apples",
    "she likes apples",
    "he likes apples",
]

tgt_raw = [
    "ik ben een student",
    "ik ben een leraar",
    "jij bent een student",
    "jij bent een leraar",
    "zij is een student",
    "hij is een leraar",
    "wij zijn studenten",
    "zij zijn leraren",
    "ik lust appels",
    "jij lust appels",
    "zij lust appels",
    "hij lust appels",
]
```

As before, this dataset is far too small for any serious translation, but it is ideal for illustrating the mechanics of encoder–decoder training.

We largely repeat what we did for the language model, but now we have *two* languages.

5.1.1 Preprocessing steps

First we import the necessary libraries

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

5.1.1.1 Adding <sos> and <eos>

We start by importing the Keras preprocessing utilities and adding explicit start/end markers to every sentence on both sides.

```
# Add <sos> and <eos> on both sides
src = [f"<sos> {s} <eos>" for s in src_raw]
tgt = [f"<sos> {s} <eos>" for s in tgt_raw]

print("Example source with markers:", src[0])
print("Example target with markers:", tgt[0])
```

This prints:

```
Example source with markers: <sos> i am a student <eos>
Example target with markers: <sos> ik ben een student <eos>
```

These markers play the same role as in the language model: they tell the model where sentences begin and end.

5.1.1.2 Tokenisers and Integer Sequences

We now build two tokenisers: one for the English source, one for the Dutch target. We then convert sentences into sequences of integer IDs.

```
# Separate tokenisers for source (English) and target (Dutch)
tokenizer_src = Tokenizer(filters="", lower=True)
tokenizer_tgt = Tokenizer(filters="", lower=True)

tokenizer_src.fit_on_texts(src)
tokenizer_tgt.fit_on_texts(tgt)

src_sequences = tokenizer_src.texts_to_sequences(src)
tgt_sequences = tokenizer_tgt.texts_to_sequences(tgt)

print("First source sequence:", src_sequences[0])
print("First target sequence:", tgt_sequences[0])
```

The printed sequences might look like:

```
First source sequence: [1, 6, 10, 3, 7, 2]
First target sequence: [1, 6, 11, 3, 7, 2]
```

Here, each integer refers to a word in a word-index dictionary such as `tokenizer_tgt.word_index` (e.g. `"ik" -> 6`, `"student" -> 7`).

We can look at the entire source and target language vocabularies:

```
tokenizer_src.index_word
```

gives:

```
{1: '<sos>',
 2: '<eos>',
 3: 'a',
 4: 'are',
 5: 'apples',
```

```
 6: 'i',
 7: 'student',
 8: 'teacher',
 9: 'you',
 10: 'am',
 11: 'she',
 12: 'is',
 13: 'he',
 14: 'like',
 15: 'likes',
 16: 'we',
 17: 'students',
 18: 'they',
 19: 'teachers'}
```

and

```
tokenizer_tgt.index_word
```

gives a similar list for the target language.

5.1.1.3 Vocabulary Sizes and Maximum Lengths

As before, we add 1 to the vocabulary sizes to reserve index 0 for padding. We also compute the maximum sentence length on each side.

```
# Vocabulary sizes (+1 for padding index 0)
num_src_tokens = len(tokenizer_src.word_index) + 1
num_tgt_tokens = len(tokenizer_tgt.word_index) + 1

# Maximum lengths
max_src_len = max(len(seq) for seq in src_sequences)
max_tgt_len = max(len(seq) for seq in tgt_sequences)

print("num_src_tokens:", num_src_tokens)
print("num_tgt_tokens:", num_tgt_tokens)
print("max_src_len:", max_src_len)
print("max_tgt_len:", max_tgt_len)
```

This gives:

```
num_src_tokens: 20
num_tgt_tokens: 19
max_src_len: 6
max_tgt_len: 6
```

This tells us how large the embedding layers must be and how many time steps the encoder and decoder will process. Each embedding layer needs to know how many possible token IDs exist so it can create a lookup table of the right size.

Concretely:

- The source embedding layer must have shape `Embedding(input_dim=num_src_tokens, output_dim=embed_dim)`. If `num_src_tokens = 20`, then the embedding layer is a table with 20 rows, one vector per word.
- The target embedding layer must have shape `Embedding(input_dim=num_tgt_tokens,output_dim=embed_dim)`. If `num_tgt_tokens = 19`, the decoder embedding layer is a table with 19 rows.

`embed_dim` is the number of embedding dimensions, a hyperparameter which we can choose.

5.1.1.4 Padding to Fixed Length

Neural networks operate on rectangular batches, so we pad all sequences on both sides to a fixed length using zeros (the padding ID). We use *post-padding* again, i.e. zeros are added at the end of shorter sentences.

```
# Pad with zeros on the right (post-padding)
src_sequences_padded = pad_sequences(
    src_sequences, maxlen=max_src_len, padding="post"
)
tgt_sequences_padded = pad_sequences(
    tgt_sequences, maxlen=max_tgt_len, padding="post"
)

print("encoder input shape:", src_sequences_padded.shape)
print("full target shape:", tgt_sequences_padded.shape)
print("Example padded source:", src_sequences_padded[0])
print("Example padded target:", tgt_sequences_padded[0])
```

For instance, if the first source sentence has length 6 and our `max_src_len` is also 6, no padding is needed. Shorter sentences will have zeros appended until they reach length 6.

5.1.1.5 Shifted Decoder Inputs and Targets

Teacher Forcing

When training the decoder of an encoder–decoder model, we must decide what input the decoder should receive at each time step. Recall that the decoder is a *conditional language model*: at every position it predicts the next target word.
There are two possibilities:

1. Option 1: Let the decoder feed itself (no teacher forcing).

   We could let the decoder use its *own* predictions as input at the next step. For example:

   - Start with `<sos>`.
   - Predict a word (e.g. `ik`).
   - Feed that prediction back in.
   - Predict the next word, and so on.

   This works during *inference*, but during training it causes problems: early mistakes propagate. If the model predicts `hij` instead of `ik` for the first word, then the correct

second word `ben` no longer fits the context. The model ends up learning from its own errors.

2. Option 2: Use teacher forcing (standard in NMT training).

   Instead of feeding the model's predictions back into the decoder, we feed the *correct* previous target word at each time step. This stabilises training and allows the model to learn grammatical patterns much more quickly.

To use teacher forcing we split each padded target sequence into two shifted versions:

$\text{decoder\_input\_data} = [w_0,\ w_1,\ w_2,\ \ldots,\ w_{T-1}]$,

$\text{decoder\_target\_data} = [w_1,\ w_2,\ w_3,\ \ldots,\ w_T]$.

In other words:

- the decoder input sequence drops the final token;
- the decoder target sequence drops the first token.

This ensures that at every time step the decoder learns to predict the next word in the sentence, exactly like the recurrent language model from the previous section.

```
# Decoder input / output for teacher forcing:
#   decoder_input_data:  <sos> ik ben een student
#   decoder_target_data: ik ben een student <eos>
decoder_input_data = tgt_sequences_padded[:, :-1]
decoder_target_data = tgt_sequences_padded[:, 1:]

print("decoder input shape:", decoder_input_data.shape)
print("decoder target shape:", decoder_target_data.shape)
print("decoder_input_data[0]:", decoder_input_data[0])
print("decoder_target_data[0]:", decoder_target_data[0])
```

This should print

```
decoder input shape: (12, 5)
decoder target shape: (12, 5)
decoder_input_data[0]: [ 1  6 11  3  7]
decoder_target_data[0]: [ 6 11  3  7  2]
```

So at each position the decoder sees the number of the previous word and learns to predict the next one, exactly as in the RNN language model.

5.1.1.6 Convenience Dictionaries for Decoding

As a final preprocessing step, we store some helper structures for use during generation: the target word-index mapping and the special IDs for `<sos>` and `<eos>`.

```
# Convenience dictionaries for later decoding
tgt_word_index = tokenizer_tgt.word_index
tgt_index_word = {i: w for w, i in tgt_word_index.items()}

eos_id = tgt_word_index["<eos>"]
```

```
sos_id = tgt_word_index["<sos>"]

print("sos_id:", sos_id, "->", tgt_index_word[sos_id])
print("eos_id:", eos_id, "->", tgt_index_word[eos_id])
```

These variables will be used later in the inference code to start decoding with `<sos>` and to stop when `<eos>` is generated.

Up to this point, the pipeline is almost identical to the language modelling setup, except that we now have *two* tokenisers and we split the target into decoder inputs and decoder targets.

5.1.2 Building and Training the Encoder–Decoder Model

After preprocessing the data (Steps 1–6), we now construct the full encoder–decoder architecture. This section mirrors the design of the first generation of neural machine translation systems (Cho et al., 2014; Sutskever et al., 2014): an encoder RNN compresses the source sentence into a fixed-size vector, and a decoder RNN generates the target sentence one word at a time, using teacher forcing during training.

5.1.2.1 Building the Encoder

The encoder is the part of the model that reads the source-language sentence and builds an internal representation of its meaning. Unlike the decoder, the encoder does not produce words; its task is to *understand* the input sentence as a whole.

```
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np

embed_dim = 64
hidden_size = 128

encoder_inputs = keras.Input(
    shape=(max_src_len,), dtype="int32", name="encoder_inputs"
)

encoder_embedding = layers.Embedding(
    input_dim=num_src_tokens,
    output_dim=embed_dim,
    mask_zero=True,
    name="encoder_embedding",
)

encoder_embedded = encoder_embedding(encoder_inputs)

encoder_rnn = layers.SimpleRNN(
    hidden_size,
    return_state=True,      # return only the final state
    name="encoder_rnn",
)

encoder_output, encoder_state = encoder_rnn(encoder_embedded)
```

Below, we explain the encoder code step by step.

```
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
```

These lines import the libraries used to build the model. `keras` and `layers` provide building blocks such as input layers, embedding layers, and recurrent layers. (`numpy` is a standard library for numerical data and is often used for preprocessing.)

```
embed_dim = 64
hidden_size = 128
```

Here we set two important sizes:

- `embed_dim`: how many numbers we use to represent each word (a word embedding).
- `hidden_size`: how large the encoder's internal memory is while reading a sentence.

Larger values can capture more information, but also make the model slower and more prone to overfitting if the dataset is small.

1. **The encoder input**

   ```
   encoder_inputs = keras.Input(
       shape=(max_src_len,), dtype="int32", name="encoder_inputs"
   )
   ```

   This defines what the encoder receives as input: a sequence of integers (word IDs), with one integer per word position.

   - `max_src_len` is the maximum source sentence length we allow.
   - Each sentence shorter than this is padded with a special padding symbol.

   So the encoder always receives a sentence in a fixed-length format, even though real sentences can be shorter.

2. **Turning word IDs into embeddings**

   ```
   encoder_embedding = layers.Embedding(
       input_dim=num_src_tokens,
       output_dim=embed_dim,
       mask_zero=True,
       name="encoder_embedding",
   )
   ```

   This creates an embedding layer for the source language.

   - `input_dim=num_src_tokens`: the size of the source vocabulary (how many different source tokens we know).
   - `output_dim=embed_dim`: each token is represented as a list of `embed_dim` numbers.
   - `mask_zero=True`: the token ID 0 is treated as padding and will be ignored.

   The embedding layer can be compared to a dictionary that maps each source token ID to a learned numeric representation.

```
encoder_embedded = encoder_embedding(encoder_inputs)
```

This applies the embedding layer to the input word IDs. After this step, we no longer have integers such as "57" or "1032", but a richer representation for each word position.

3. **The recurrent encoder**

```
encoder_rnn = layers.SimpleRNN(
    hidden_size,
    return_state=True,      # return only the final state
    name="encoder_rnn",
)
```

This defines a recurrent neural network (RNN) that reads the sentence one word at a time, from left to right.

As it reads each word, it updates an internal memory. At the end, this memory contains a summary of the whole sentence.

The option `return_state=True` means: *also return the final internal memory after reading the last word.*

4. **Running the encoder**

```
encoder_output, encoder_state = encoder_rnn(encoder_embedded)
```

This line actually runs the encoder over the embedded source sentence.

- `encoder_state` is the final internal representation after the last word has been read. It summarises the full source sentence.
- `encoder_output` is the same value in this configuration.

In other words: because we did *not* ask for outputs at every time step, the encoder returns only the final summary.

Intuition: the encoder reads the source sentence and produces one compact "meaning summary". The decoder will then use that summary to start generating the translation.

5.1.2.2 Building the Decoder

The decoder is responsible for *producing the translation*. It generates the target sentence one word at a time, starting from the meaning representation produced by the encoder.

During training, the decoder does not work completely on its own. Instead, it is guided by the correct target sentence. This training strategy is known as *teacher forcing*.

The decoder is structurally almost identical to the recurrent language model. The crucial difference is that its initial hidden state is set to the encoder state, so its predictions depend on the source sentence.

```
decoder_inputs = keras.Input(
    shape=(max_tgt_len - 1,), dtype="int32", name="decoder_inputs"
)
```

```
decoder_embedding = layers.Embedding(
    input_dim=num_tgt_tokens,
    output_dim=embed_dim,
    mask_zero=True,
    name="decoder_embedding",
)

decoder_embedded = decoder_embedding(decoder_inputs)

decoder_rnn = layers.SimpleRNN(
    hidden_size,
    return_sequences=True,    # produce one output per time step
    return_state=True,
    name="decoder_rnn",
)

decoder_outputs, _ = decoder_rnn(
    decoder_embedded,
    initial_state=encoder_state    # encoder → decoder connection
)

decoder_dense = layers.Dense(
    num_tgt_tokens,
    activation="softmax",
    name="decoder_output_dense",
)

decoder_outputs = decoder_dense(decoder_outputs)
```

Below, we explain the encoder code step by step.

1. **1) The decoder input**

   ```
   decoder_inputs = keras.Input(
       shape=(max_tgt_len - 1,), dtype="int32", name="decoder_inputs"
   )
   ```

   This defines what the decoder receives as input during training: a sequence of target-language word IDs.

   The target sentence is split into two parts:

   - **Decoder input:** <sos> y_1 y_2 ... y_{T-1}
   - **Decoder output:** y_1 y_2 ... y_T <eos>

   At each step, the decoder is shown the *correct previous word* and is trained to predict the next one.

2. **Target-language word embeddings**

   ```
   decoder_embedding = layers.Embedding(
       input_dim=num_tgt_tokens,
       output_dim=embed_dim,
       mask_zero=True,
       name="decoder_embedding",
   )
   ```

This embedding layer converts target-language word IDs into dense numerical representations.

- `input_dim=num_tgt_tokens`: the size of the target vocabulary
- `output_dim=embed_dim`: how many numbers are used to represent each word
- `mask_zero=True`: padding symbols are ignored

As in the encoder, padding words do not influence the model.

```
decoder_embedded = decoder_embedding(decoder_inputs)
```

This line applies the embedding layer to the decoder input. Each target word is now represented in a form that the neural network can work with.

3. **The decoder RNN**

```
decoder_rnn = layers.SimpleRNN(
    hidden_size,
    return_sequences=True,
    return_state=True,
    name="decoder_rnn",
)
```

The decoder uses a recurrent neural network to generate the translation step by step.

Unlike the encoder, the decoder must produce an output at *every word position*, because each position corresponds to a predicted target word.

- `return_sequences=True`: produce one output per word
- `return_state=True`: also return the final internal state

4. **Connecting the encoder and decoder**

```
decoder_outputs, _ = decoder_rnn(
    decoder_embedded,
    initial_state=encoder_state
)
```

This is the key moment where understanding turns into generation. The encoder has already read the entire source sentence and produced a summary of its meaning. This summary is passed to the decoder as its starting state.

Intuition: the decoder starts translating with the source sentence already "in mind".

The decoder then proceeds word by word, combining:

- the meaning from the encoder
- the correct previous target word (during training)

The second returned value (the final internal state) is ignored here, because it is not needed for training. It becomes important during translation at test time.

5. **From internal states to word probabilities**

```
decoder_dense = layers.Dense(
    num_tgt_tokens,
    activation="softmax",
    name="decoder_output_dense",
)
```

At this stage, the decoder has produced internal representations for each position in the target sentence. These are not yet actual words.

This layer converts each internal representation into a probability distribution over all target-language words.

- Each word in the vocabulary receives a probability
- All probabilities for one position sum to one

```
decoder_outputs = decoder_dense(decoder_outputs)
```

The dense layer is applied to every word position in the sentence. The result is a probability distribution over the target vocabulary for each position.

What the decoder produces For each sentence and each position in the target sentence, the decoder outputs a list of probabilities over all possible target words.

During training, the model is encouraged to assign a high probability to the correct next word.

Conceptually, the decoder learns to answer the question:

> "Given the source sentence and the words I have already produced, what is the most likely next word in the target language?"

In this basic encoder–decoder model, the decoder relies entirely on the single summary produced by the encoder. For longer sentences, this summary may be insufficient.

5.1.2.3 Building the Full Training Model

We now join encoder and decoder into a single model with two inputs and one output. During training, the decoder receives the shifted target sequence (`decoder_input_data`), and the model is trained to predict the next token (`decoder_target_data`) at each position.

```
nmt_model = keras.Model(
    inputs=[encoder_inputs, decoder_inputs],
    outputs=decoder_outputs,
    name="simple_rnn_seq2seq"
)

nmt_model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.01),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)

nmt_model.summary()
```

Here is a step-by-step explanation.

```
nmt_model = keras.Model(
    inputs=[encoder_inputs, decoder_inputs],
    outputs=decoder_outputs,
    name="simple_rnn_seq2seq"
)
```

This line tells Keras how all parts of the model are connected.

- The model has **two inputs**:
  - the source sentence (`encoder_inputs`),
  - the shifted target sentence used during training (`decoder_inputs`).
- The model produces **one output**:
  - the predicted target words at each position (`decoder_outputs`).

Conceptually, the model behaves as follows: The encoder reads the source sentence, the decoder reads the target sentence so far, and the model predicts the next target word at each position.

Telling the model how to learn

```
nmt_model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.01),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

The `compile` step specifies how the model should be trained.

Optimizer The optimizer controls how the model updates its internal parameters while learning from data.

- `Adam` is a widely used optimizer that adapts learning speed automatically.
- `learning_rate=0.01` controls how large each update step is.

In intuitive terms, the optimizer decides *how quickly* the model should change its behaviour when it makes a mistake.

Loss function The loss function tells the model what counts as a mistake.

- `sparse_categorical_crossentropy` compares the predicted word probabilities with the correct target word.
- It is called *sparse* because the correct word is given as a single word ID, not as a full list of probabilities.

At each position in the target sentence, the model is penalised when it assigns low probability to the correct word.

Metric `accuracy` reports how often the model's most likely predicted word matches the correct target word.

Accuracy is easy to interpret, but it is not a perfect evaluation measure for translation quality. For this reason, we have introduced in chapter 3 specialised MT metrics such as BLEU and chrF.

Inspecting the model

```
nmt_model.summary()
```

This command prints a summary of the model structure.

The summary shows:

- each layer in the model,
- how they are connected,
- how many trainable parameters they contain.

This is useful for:

- checking that the model was built as intended,
- understanding where most of the model's complexity lies,
- diagnosing mistakes in the model design.

> **Intuition:** the summary is a blueprint of the translation system, showing how information flows from the source sentence to the predicted target words.

This completes the classic sequence-to-sequence training architecture.

Visualising the Encoder–Decoder Network As before, we can ask Keras to draw a diagram of the model. This helps to see the structure of the encoder–decoder architecture: two inputs (source and shifted target), shared hidden dimension, and a softmax over the target vocabulary.

```
from tensorflow.keras.utils import plot_model

plot_model(
    nmt_model,
    to_file="rnn_seq2seq.png",
    show_shapes=True,
    show_layer_names=True,
)
```

The diagram in Figure 5.2 shows:

- an **encoder branch**, starting from `encoder_inputs`, passing through `encoder_embedding` and `encoder_rnn`, which produces the final hidden state;
- a **decoder branch**, starting from `decoder_inputs`, passing through `decoder_embedding` and `decoder_rnn`;
- the encoder's final state is used as the *initial state* of the decoder RNN, so the decoder is conditioned on the source sentence;
- a final `Dense` layer with softmax, which maps each decoder hidden state to a distribution over the target vocabulary.

Compared with the language model diagram, the main difference is the presence of two separate input branches (source and target) and the explicit connection from the encoder's final state into the decoder.

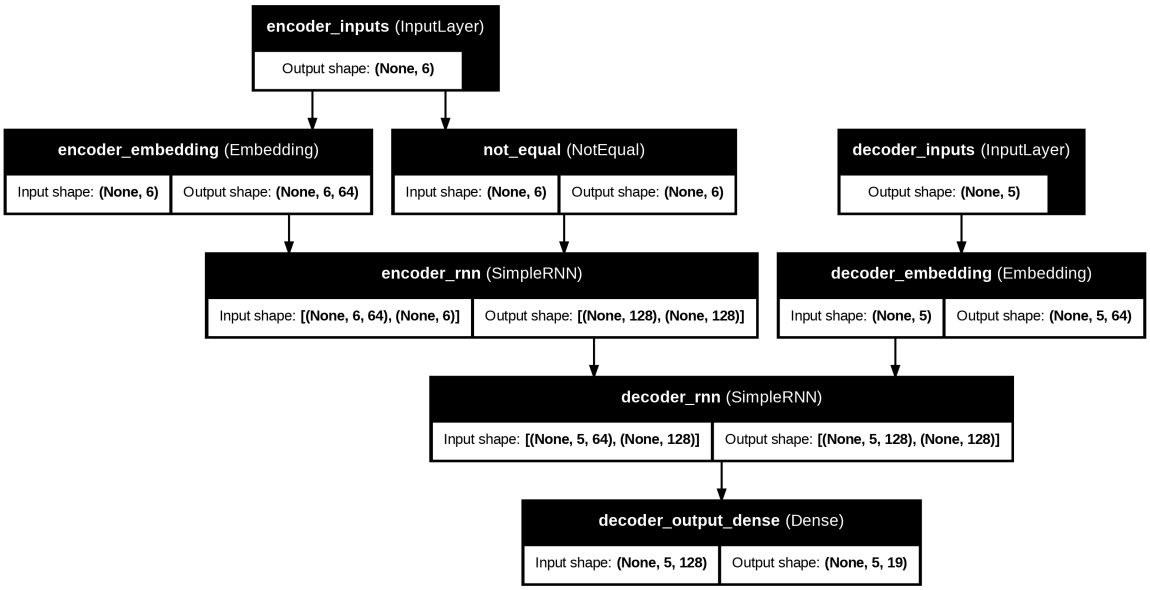


Figure 5.2: Visualisation of the simple RNN encoder–decoder model according to Keras.

5.1.2.4 Training the Seq2Seq Model

Training proceeds similarly to the recurrent language model, except that the decoder is now conditioned on the encoder state. On this small corpus the model will overfit quickly, but this is fine for illustration.

```
history = nmt_model.fit(
    [src_sequences_padded, decoder_input_data],
    decoder_target_data,
    batch_size=16,
    epochs=20,
    verbose=1
)
```

This command tells the model to learn from the training data by repeatedly comparing its predictions with the correct translations.

- `[src_sequences_padded, decoder_input_data]`: These are the **inputs** to the model:
  - `src_sequences_padded`: the source sentences, converted to word IDs and padded to a fixed length.
  - `decoder_input_data`: the target sentences shifted to the right (starting with `<sos>`), used to guide the decoder during training.
- `decoder_target_data`: This is the **expected output** of the model:
  - the correct next target word at each position
  - typically the same target sentence, shifted to the left (ending with `<eos>`)
- `batch_size=16`: The model processes 16 sentence pairs at a time before updating itself. Using batches makes training faster and more stable.
- `epochs=20`: The model goes through the entire training dataset 20 times. Each pass gives the model another opportunity to improve its translations.

- `verbose=1`: This option controls how much information is printed during training. With `verbose=1`, the model shows a progress bar and training statistics after each epoch.

At each training step, the model:

- reads a batch of source sentences,
- generates predictions for the next target words,
- compares its predictions with the correct translations,
- slightly adjusts its internal parameters to reduce future errors.

The returned object `history` stores information about the training process, such as how the loss and accuracy change over time. This information can later be used to create learning curves and analyse training behaviour.

5.1.3 Inference: Using the model to translate

During training, the model learns to translate by seeing complete sentence pairs. The decoder is helped by *teacher forcing*: instead of having to guess its own previous output, it is always given the correct previous target token. This makes learning faster and more stable.

During translation, however, the situation is different. The correct target sentence is unknown, so the decoder can no longer be helped in this way. It must generate the translation *one token at a time*, using its own previous predictions as input. For this reason, we need slightly different versions of the encoder and decoder for inference, even though they reuse the same learned weights.

5.1.3.1 Building Inference Models

Encoder inference model At translation time, the encoder still reads the entire source sentence, just as it does during training. The difference is that its output is now computed once, stored, and then reused while the decoder generates the translation word by word. For this reason, the encoder must be available as a separate model whose final hidden state can be passed explicitly to the decoder.

We therefore build an encoder inference model that takes a source sentence as input and returns its final hidden state:

```
encoder_model = keras.Model(
    encoder_inputs,
    encoder_state,
    name="encoder_inference"
)
```

This model uses the same encoder parameters as during training, but its output is now made explicit so that it can be reused during decoding.

Decoder inference model The decoder behaves very differently during inference than during training. Instead of receiving the full target sentence, it now works step by step.

At each step, the decoder:

- receives a single token (the previously generated word),

- receives the previous hidden state,
- predicts the next word,
- updates its hidden state.

To make this possible, we define a decoder inference model that explicitly takes both the current token and the current hidden state as input, and returns both the predicted next-token probabilities and the updated hidden state:

```
decoder_state_input = keras.Input(
    shape=(hidden_size,), name="decoder_state_input"
)
decoder_single_input = keras.Input(
    shape=(1,), dtype="int32", name="decoder_single_input"
)

decoder_single_embedded = decoder_embedding(decoder_single_input)

decoder_single_output, decoder_state_output = decoder_rnn(
    decoder_single_embedded,
    initial_state=decoder_state_input
)

decoder_single_probs = decoder_dense(decoder_single_output)

decoder_model = keras.Model(
    [decoder_single_input, decoder_state_input],
    [decoder_single_probs, decoder_state_output]
)
```

- `decoder_state_input` defines an input layer that receives the previous hidden state of the decoder RNN. Its shape corresponds to the size of the hidden state (`hidden_size`).
- `decoder_single_input` defines an input layer for a single decoder token (one time step), represented as an integer word index.
- `decoder_single_embedded` applies the embedding layer to the input token, converting the integer index into a dense vector representation.
- `decoder_rnn` processes the embedded token together with the previous decoder state (`initial_state`), producing:
  - the output for the current time step, and
  - the updated decoder hidden state.
- `decoder_dense` applies a dense (softmax) layer to the decoder output to obtain a probability distribution over the target vocabulary.
- `decoder_model` defines a Keras model that takes as input:
  - the current decoder token, and
  - the previous decoder hidden state,

  and outputs:
  - the predicted token probabilities, and

- the updated decoder hidden state.

This decoder behaves like a language model that we run manually, one step at a time.

5.1.4 Translation – Decoding

Finally, we define a simple decoding function. It encodes the source sentence, then repeatedly calls the decoder to produce the next target word, stopping when `<eos>` is generated.

```
def translate_rnn(sentence, max_len=10):
    # Preprocess source sentence
    src_text = f"<sos> {sentence.lower()} <eos>"
    seq = tokenizer_src.texts_to_sequences([src_text])[0]
    seq = pad_sequences([seq], maxlen=max_src_len, padding="post")

    # Encode
    state = encoder_model.predict(seq, verbose=0)

    # Start with <sos>
    target_id = sos_id
    decoded = []

    for _ in range(max_len):
        token_seq = np.array([[target_id]])
        probs, state = decoder_model.predict([token_seq, state], verbose=0)
        target_id = np.argmax(probs[0, 0, :])

        if target_id == eos_id:
            break

        decoded.append(tgt_index_word[target_id])

    return " ".join(decoded)
```

- **Goal of `translate_rnn()`** This function translates a single source sentence by running the encoder once and then generating the target sentence word-by-word with the decoder. This is necessary at inference time because we do not know the correct target words in advance (so we cannot use teacher forcing).

1. **Preprocess the source sentence.**
   - We add explicit start/end markers to the source sentence: `<sos>` (start of sentence) and `<eos>` (end of sentence).
   - We lowercase the sentence (`sentence.lower()`) to match the training preprocessing (if the model was trained on lowercase text).
   - We convert the text into integer token IDs using the `tokenizer_src`.

   ```
   src_text = f"<sos> {sentence.lower()} <eos>"
   seq = tokenizer_src.texts_to_sequences([src_text])[0]
   ```

2. **Pad the source sequence to a fixed length.** Neural models typically expect inputs of the same length within a batch. Even though we translate one sentence here, we still

pad it to `max_src_len` (the length used during training), adding padding tokens at the end (`padding="post"`).

```
seq = pad_sequences([seq], maxlen=max_src_len, padding="post")
```

3. **Encode the source sentence (run the encoder once).**
   - The encoder reads the entire padded source sequence.
   - It outputs a single hidden state vector, which summarises the source sentence.
   - This hidden state will be used to initialise the decoder.

```
state = encoder_model.predict(seq, verbose=0)
```

4. **Initialise decoding with `<sos>`.**
   - Decoding must start somewhere. We begin with the special start token `<sos>`.
   - `target_id` stores the current token ID that will be fed into the decoder.
   - `decoded` will store the generated target words.

```
target_id = sos_id
decoded = []
```

5. **Generate tokens one-by-one (autoregressive decoding).** We repeat the decoding step up to `max_len` times to avoid infinite loops (for example, if the model never produces `<eos>`).

   a) **Prepare the decoder input token.** The decoder expects a sequence of length 1 (one token), so we wrap `target_id` as a `(1,1)` NumPy array.

   ```
   token_seq = np.array([[target_id]])
   ```

   b) **Run one decoder step.** Conceptually, a decoder conditions on the entire previously generated target sequence. In this implementation, that history is stored in the hidden state, so the decoder only needs to receive the most recent token at each step.
      - The decoder receives:
        - the current token (`token_seq`), and
        - the current hidden state (`state`).
      - It returns:
        - `probs`: a probability distribution over the target vocabulary for the *next* token, and
        - an updated hidden state (also stored back into `state`).

   ```
   probs, state = decoder_model.predict([token_seq, state], verbose=0)
   ```

   c) **Choose the most likely next token.**

- `probs[0,0,:]` is the vocabulary-sized vector of probabilities for the next token. `probs` has three dimensions: batch size (here: 1 sentence), time steps (here: 1 token) and vocabulary (one probability per target word).
- `np.argmax(...)` selects the token ID with the highest probability. (This is *greedy decoding*.)
- We overwrite `target_id` with this newly predicted token, because it will be fed back into the decoder at the next step.

```
target_id = np.argmax(probs[0, 0, :])
```

d) **Stop if we predict <eos>.**

- If the predicted token is the end-of-sentence symbol, we stop decoding.
- Otherwise we convert the predicted ID back to a word and append it to the output list.

```
if target_id == eos_id:
    break

decoded.append(tgt_index_word[target_id])
```

6. **Return the final translation as text.**

- `decoded` is a list of words.
- We join them with spaces to form a sentence-like string.

```
return " ".join(decoded)
```

This completes a minimal recurrent encoder–decoder translation system, matching the architecture of the earliest neural machine translation models.

5.1.5 Example Translations (End-to-End)

To illustrate how the model performs translation at inference time, we now walk through complete examples using the `translate_rnn` function defined in section 5.1.4. The process mirrors the behaviour of the recurrent language model but conditions the decoder on the encoder's final state.

5.1.5.1 Example 1: Translating "she likes apples"

We begin by passing the English sentence to our translation function:

```
print(translate_rnn("she likes apples"))
```

Output after training on our toy corpus is:

```
zij lust appels
```

This result reflects patterns learned from the Dutch side of the corpus. Let us break down the steps of the inference procedure.

- **Preprocessing and Encoding**

  The sentence is first wrapped in `<sos>` and `<eos>`:

  `<sos> she likes apples <eos>`

  and converted into its integer sequence. After padding we obtain:

```
src_text = "<sos> she likes apples <eos>"
print(tokenizer_src.texts_to_sequences([src_text]))
```

```
[[1, 11, 15, 5, 2]]
```

  This padded sequence is then passed to the encoder:

```
state = encoder_model.predict(
    pad_sequences(
        tokenizer_src.texts_to_sequences([src_text]),
        maxlen=max_src_len,
        padding="post"
    ),
    verbose=0
)

print("Encoder state shape:", state.shape)
```

  The vector `state` is a dense summary of the entire source sentence and initialises the decoder.

- **First Decoder Step**

  We begin decoding with the token `<sos>`:

```
token = sos_id
token_seq = np.array([[token]])
probs, next_state = decoder_model.predict([token_seq, state], verbose=0)

predicted_id = np.argmax(probs[0, 0, :])
print("First predicted token:", tgt_index_word[predicted_id])
```

  This produces `zij`, reflecting that Dutch sentences for this corpus often begin with the subject pronoun.

- **Subsequent Decoder Steps**

  At each following step, we feed the previous output token and the updated hidden state:

```
decoded = []
state_now = next_state
current_token = predicted_id

for _ in range(5):
```

```
    token_seq = np.array([[current_token]])
    probs, state_now = decoder_model.predict([token_seq, state_now],
    ↪  verbose=0)
    current_token = np.argmax(probs[0,0,:])

    if current_token == eos_id:
        break

    decoded.append(tgt_index_word[current_token])

print("Decoded continuation:", decoded)
```

A continuation is:

```
['lust', 'appels']
```

Yielding the full translation:

```
zij lust appels
```

5.1.5.2 Example 2: “i am a student”

```
print(translate_rnn("i am a student"))
```

Expected output:

```
ik ben een student
```

Walking through the steps:

- encoder reads: `<sos> i am a student <eos>`
- initial decoder prediction: `ik`
- next predictions: `ben`, `een`, `student`
- stopping at `<eos>`

This matches exactly the Dutch side of the corpus.

5.1.5.3 Example 3: “you like apples”

```
print(translate_rnn("you like apples"))
```

Possible output:

```
jij lust appels
```

5.2 Scaling up: A Vanilla RNN NMT Baseline

The first neural machine translation systems were based on recurrent encoder–decoder architectures without attention (Cho et al., 2014; Sutskever et al., 2014). In this section, we will experimentally test how well such a system works for our data.

Kaggle session

We will no longer use Google Colab, as the special hardware (GPUs) that are available in Google are unreliable in the sense that it is unclear how much processing time we have. We will therefore use Kaggle instead. This is somewhat similar to Colab in the sense that we can run python notebooks and have free GPUs. While it is somewhat less user-friendly in interactive mode, it is actually intended to run (train models etc.) in an offline mode, so we do not need to stay logged in.
You should therefore make a Kaggle account and check the session HERE

In the previous section we implemented a minimal encoder–decoder model in Keras to understand every step of training and inference. We now switch to the `rnn_seq2seq.py` training script, which allows us to run controlled MT experiments on a larger dataset and to compare architectural improvements in a reproducible way.

The command-line options for the script are presented in Table 5.1.

The first thing we need to do is to download the required scripts and modules to the Kaggle session.

```
!pip install sacrebleu
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/rnn_seq2seq.py
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/plot_train_val.py
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/plot_history.py
from IPython.display import Image, display
```

To ensure that all improvements in this chapter are comparable, we re-use the data preparation and evaluation workflow introduced earlier:

- **Data preparation.** We use the same preprocessing pipeline and, crucially, a **fixed** development and test set (Chapter 2). This avoids "moving targets": if the dev set changes, BLEU changes even if the model does not.
- **Evaluation.** We will only do a final evaluation on the test set, once we've developed a system that works reasonably well on the test set, using overlap metrics and semantic metrics (Chapter 3): BLEU, chrF, BERTScore, and COMET.

5.2.1 Goal of the Baseline

This first script-based experiment establishes a baseline system that mirrors early NMT: a plain encoder–decoder architecture without attention, without gating (no GRU/LSTM), and without subwording.

The point is not to obtain state-of-the-art quality, but to create a **reference point** against which later improvements (GRU/LSTM, bidirectionality, attention, subwording) can be measured.

Option	Default	Comment
`-mode`	`train`	Run the script in training mode
`-src-file`	–	Source language training file (required)
`-tgt-file`	–	Target language training file (required)
`-src-val`	–	Source validation file
`-tgt-val`	–	Target validation file
`-epochs`	30	Number of training epochs
`-batch-size`	32	Number of sentence pairs per batch
`-emb-size`	64	Size of word embeddings
`-hidden-size`	128	Size of the RNN hidden state
`-enc-layers`	1	Number of encoder layers
`-dec-layers`	1	Number of decoder layers
`-rnn-type`	`rnn`	RNN cell type (`rnn`, `gru`, `lstm`)
`-attention`	`none`	Attention mechanism (`none`, `luong`)
`-bidirectional`	false	Use a bidirectional encoder
`-dropout`	0.0	Dropout probability
`-teacher-forcing`	0.7	Teacher forcing ratio
`-lr`	0.001	Learning rate
`-max-len`	20	Maximum sentence length (tokens)
`-limit`	–	Limit number of training sentence pairs
`-max-src-vocab`	–	Maximum source vocabulary size
`-max-tgt-vocab`	–	Maximum target vocabulary size
`-lower`	false	Lowercase all text
`-subword-type`	`none`	Subwording method (`none`, `bpe`, `unigram`)
`-src-sp-model`	–	Source SentencePiece model
`-tgt-sp-model`	–	Target SentencePiece model
`-save`	`model_att.pt`	Base filename for checkpoints
`-save-best`	–	Filename for best model checkpoint
`-keep-last`	1	Number of last checkpoints to keep
`-early-stopping`	0	Early stopping patience (0 disables)
`-early-metric`	`loss`	Metric for early stopping / best model
`-eval-metrics`	false	Compute BLEU/chrF/TER on validation data
`-show-val-examples`	0	Number of validation examples to print
`-save-val-json`	–	Save validation outputs as JSON
`-save-val-trans`	–	Save validation outputs as TSV
`-history-json`	–	Save training history as JSON
`-seed`	42	Random seed

Table 5.1: Command-line options for training mode in `rnn_seq2seq.py`.

5.2.2 Training the RNN Baseline

Train the baseline model with a simple RNN encoder and decoder:

```
!python rnn_seq2seq.py --src-file /kaggle/input/tatoeba-en-nl/train.en --tgt-file
↪  /kaggle/input/tatoeba-en-nl/train.nl \
--src-val /kaggle/input/tatoeba-en-nl/dev.en --tgt-val
↪  /kaggle/input/tatoeba-en-nl/dev.nl \
--epochs 10 \
--save rnn_baseline.pt \
--show-val-examples 5 \
--eval-metrics \
--lower \
--history-json rnn.hist
```

Note that the `!` indicates that the code is not python code, but a linux command. The `\` indicates that the command continues on the next line.

The options have the following purpose:

- `--src-file` / `--tgt-file` specify the parallel **training** data (source language and target language). Each line in the source file corresponds to the same line in the target file.
- `--src-val` / `--tgt-val` specify a **fixed development set**. We keep this set unchanged across experiments, so changes in BLEU/chrF reflect model differences rather than different validation samples.
- `--epochs 10` trains for 10 passes over the training set. This is sufficient to observe learning curves and obtain a usable baseline, while keeping runtime manageable for classroom experiments.
- `--save rnn_baseline.pt` saves the trained model checkpoint so it can be reused for translation and evaluation without retraining.
- `--show-val-examples 3` prints a small number of development-set examples during training. This provides a qualitative signal that complements the numeric metrics.
- `--eval-metrics` computes automatic MT metrics (e.g. BLEU/chrF, and any other metrics enabled in the environment) on the development set during training. These values are used for model comparison and early stopping in later experiments.
- `--history-json rnn.hist` writes per-epoch training information to a history file (e.g. training/validation loss and development-set metrics). This enables learning-curve plots and makes training behaviour comparable across runs.

While training, after each epoch, the script will translate the development set and calculate different metrics on it.

We can already see from the **Example validation translations** during training that results are rather bad and unmeaningful. But a question we can ask is whether the models are actually learning something.

5.2.3 Plotting the Learning Curve

Neural MT should gradually improve over epochs. A learning curve makes this visible and helps diagnose convergence and overfitting. The training script can write a JSON history file (e.g. `rnn.hist`) containing per-epoch values.

We have provided two scripts that plots the values in these history files, which have been downloaded in the initial step: `plot_train_val.py` and `plot_history.py`.

Interpreting Training and Validation Loss The `plot_train_val.py` script generates a plot that allows to compare the curves for training versus validation loss.

```
!python plot_train_val.py rnn.hist --save rnn.png
display(Image(filename="rnn.png"))
```

Comparing training and validation loss over epochs provides insight into how well a model generalises beyond the data it was trained on.

- When both curves decrease together, the model is learning patterns that transfer to unseen sentences.
- If training loss continues to decrease while validation loss stops improving or starts to increase, the model begins to overfit: its performance on the training data improves, but its performance on new data deteriorates.

The difference between the two curves, often called the *generalisation gap*, indicates how strongly the model adapts to the training data. In neural machine translation, a growing gap may reflect that the model increasingly captures properties specific to the training corpus, such as frequent constructions or recurring sentence patterns, rather than learning more general translation correspondences.

Training–validation comparisons are also useful for practical decisions. They help determine whether training should be stopped earlier (early stopping) and whether increasing model complexity is justified given the available data. For this reason, inspecting training and validation loss is an essential step in analysing and comparing NMT experiments.

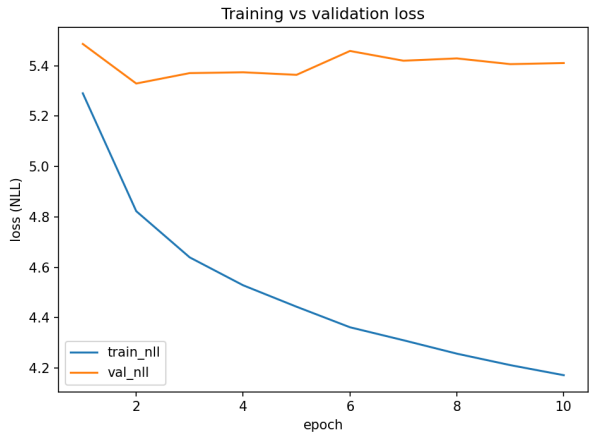


Figure 5.3: Training and validation loss over epochs for our plain RNN

As we can see in Figure 5.3 we see a clear case of overfitting. We can see that the loss on the training data steadily decreases, but that this is absolutely not the case for the validation loss.

This is confirmed by the BLEU score history, which can be visualised with

```
!python plot_history.py rnn.hist --metric bleu --save rnn_bleu.png
display(Image(filename="rnn_bleu.png"))
```

and is shown in Figure 5.4a. We see the BLEU score vary between 0.14 and 0.28, which are very low values, telling us that there is hardly any agreement between the references and the MT output. We also do not see a rise over the epochs, so the learning, as indicated on the training loss, is also not showing in the BLEU scores on the validation set.

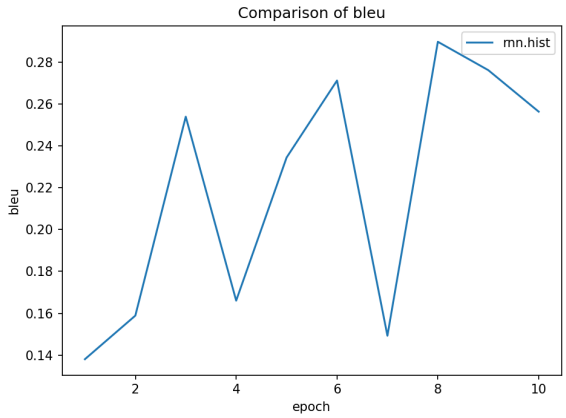
We can do a similar thing for the chrF score, which we show in Figure 5.4b, and which confirms that there is no improvement over epochs.

```
!python plot_history.py rnn.hist --metric chrf --save rnn_chrf.png
display(Image(filename="rnn_chrf.png"))
```

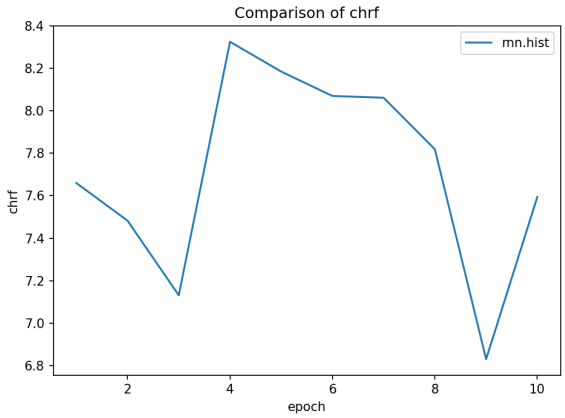
5.2.4 Translating with the `rnn_seq2seq.py` script

We can also use the script in translate mode to translate a file. You find the command-line options in Table 5.2.

To translate the test.en file we can use the following command, using the best model from the baseline training.



(a) Bleu scores on validation set over epochs for our plain RNN



(b) chrF score over epochs

Figure 5.4: Bleu and ChrF scores

Option	Default	Comment
`-mode`	`translate`	Run the script in translation mode
`-load`	–	Model checkpoint to load (required)
`-src-test`	–	Source file to translate
`-tgt-test`	–	Reference translations (optional, for scoring)
`-output`	–	Output file for generated translations
`-batch-size`	32	Batch size for decoding
`-max-len`	20	Maximum generated sentence length (tokens)
`-replace-unk`	false	Replace `<unk>` tokens using attention
`-show-src-unk`	false	Print source sentence with `<unk>` markers
`-lower`	false	Lowercase input text
`-subword-type`	`none`	Subwording method (`none`, `bpe`, `unigram`)
`-src-sp-model`	–	Source SentencePiece model
`-tgt-sp-model`	–	Target SentencePiece model
`-eval-metrics`	false	Compute BLEU/chrF/TER if references are provided
`-seed`	42	Random seed

Table 5.2: Command-line options for translation mode in `rnn_seq2seq.py`.

```
!python rnn_seq2seq.py \
  --mode translate \
  --load rnn_baseline.best.pt \
  --src-test /kaggle/input/tatoeba-en-nl/test.en \
  --tgt-test /kaggle/input/tatoeba-en-nl/test.nl \
  --output test.rnn_baseline.nl \
  --eval-metrics
```

This writes an output file with the translations and also provides us with scores for the metrics included in SacreBleu (Bleu, chrF, TER).

5.3 From Plain RNNs to Gated Recurrent Units and Long Short-Term Memory

The vanilla RNN encoder–decoder introduced in the previous section already demonstrates the core idea of neural machine translation. However, it also exhibits a well-known weakness: it struggles to remember information over longer sequences. This problem is commonly referred to

as the *vanishing gradient problem*.

In practice, this means that early words in a sentence may have little influence on later predictions, especially when sentences become longer. To address this limitation, researchers introduced *gated recurrent architectures*. These architectures modify the basic RNN cell so that the network can decide what information to keep, update, or forget.

In this section, we introduce two such architectures: the **Gated Recurrent Unit (GRU)** and the **Long Short-Term Memory (LSTM)**. Both can be used as drop-in replacements for plain RNNs in our NMT system.

5.3.1 The Idea of Gating

The key innovation behind GRUs and LSTMs is the concept of *gating*. A gate is a parameterised mechanism that regulates how information is propagated through a recurrent network over time.

In a plain RNN, the hidden state is updated at every time step using the same operation: the new input is always combined with the previous hidden state. As sequences become longer, this repeated mixing makes it difficult to preserve information originating from earlier positions in the sentence.

Gated recurrent architectures modify this update process by introducing additional control signals. These signals scale different components of the hidden state update, allowing the network to retain, suppress, or overwrite information in a controlled manner.

Each gate produces a continuous value between 0 and 1, which acts as a multiplicative factor. Values close to 0 strongly attenuate the corresponding information flow, while values close to 1 allow it to pass through largely unchanged. The parameters governing these gates are learned during training, just like all other model parameters.

In the context of machine translation, gating improves the ability of recurrent models to maintain relevant information—such as grammatical features or semantic content—across longer input sequences, thereby reducing the tendency to lose or distort earlier context.

Figure 5.5 shows the different architectures between the different types of RNN cells.

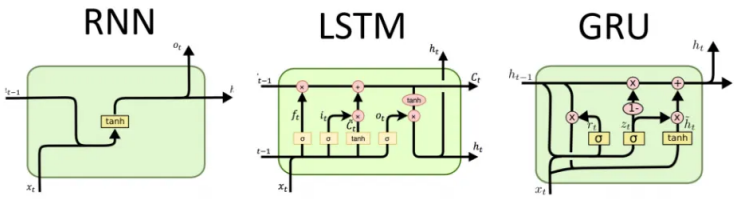


Figure 5.5: Architecture of the Hidden Layer cells: comparing RNN, LSTM and GRU (from `https://sirineamrane.medium.com/introduction-to-recurrent-neural-networks-classic-rnn-lstm-and-gru-ec76d84f286d`)

5.3.2 Gated Recurrent Units (GRU)

The Gated Recurrent Unit (GRU) was introduced by Cho et al. (2014) as an alternative to the plain RNN that can better model long-distance dependencies.

5.3.2.1 Intuition

Instead of blindly updating its internal state at every time step, a GRU uses *gates* to control information flow:

- an **update gate** decides how much of the previous state should be carried forward;
- a **reset gate** decides how much past information should be ignored when computing new content.

Intuitively, a GRU can learn to:

- keep important information over many time steps (e.g. subject or tense),
- forget irrelevant details,
- update its memory only when needed.

This makes GRUs much more effective than plain RNNs for translation, while keeping the architecture relatively simple.

5.3.2.2 GRUs in Our NMT System

In our training script, switching from a plain RNN to a GRU requires only a single option change, changing `-rnn-type` from its default value `rnn` to `gru`.

```
!python rnn_seq2seq.py --src-file /kaggle/input/tatoeba-en-nl/train.en --tgt-file
↪  /kaggle/input/tatoeba-en-nl/train.nl \
--src-val /kaggle/input/tatoeba-en-nl/dev.en --tgt-val
↪  /kaggle/input/tatoeba-en-nl/dev.nl \
--rnn-type gru \
--epochs 10 \
--save gru.pt \
--show-val-examples 5 \
--eval-metrics \
--lower \
--history-json gru.hist
```

All other aspects of the model remain unchanged. This allows us to directly measure the impact of gating on translation quality.

During the training process, we can now already see some meaningful words in the example validation translations.

Figure 5.6 shows that validation loss initially diminshes, but starts to converge after 6 epochs, while learning for the training data continues. So the model learns during the first few epochs, but then starts to overfit.

In Figure 5.7 we compare the loss and BLEU scores of the GRU models with the plain RNN models. We can see that GRU scores substantially better, both for loss (lower is better) and for BLEU (higher is better). While RNNs do show no learning on validation, GRU models at least show some loss reduction and BLEU score improvement.

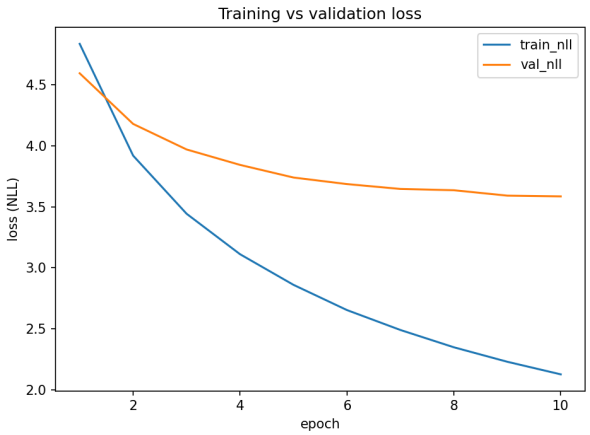
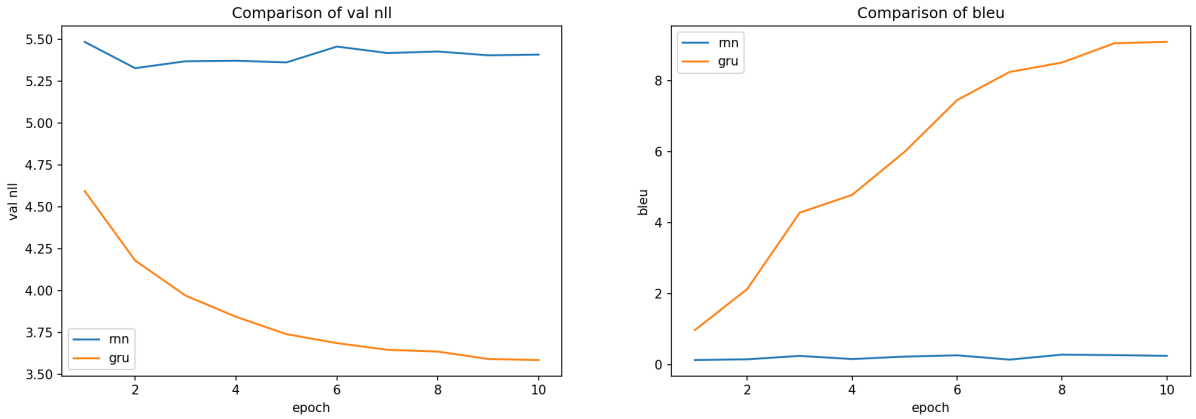


Figure 5.6: Training and validation loss over epochs for our GRU model



(a) Validation loss over epochs

(b) BLEU score over epochs

Figure 5.7: Comparing GRU with RNN

5.3.3 Long Short-Term Memory (LSTM)

The Long Short-Term Memory (LSTM) architecture was introduced earlier by Hochreiter and Schmidhuber (1997) and is one of the most influential recurrent models in neural network history.

5.3.3.1 Intuition

An LSTM explicitly separates:

- a **cell state** (long-term memory),
- a **hidden state** (short-term working memory).

Three gates control how information flows through the cell:

- a **forget gate** decides what information to discard;
- an **input gate** decides what new information to store;
- an **output gate** decides what information to expose to the next layer.

This explicit memory design allows LSTMs to preserve information over very long sequences,

making them particularly robust for sequence-to-sequence tasks such as translation.

5.3.3.2 LSTMs in Our NMT System

Switching to an LSTM in our experiments is again straightforward:

```
--rnn-type lstm
```

We can check the training versus validation loss history by generating Figure 5.8.

```
!python plot_train_val.py lstm.hist --save lstm.png
display(Image(filename="lstm.png"))
```

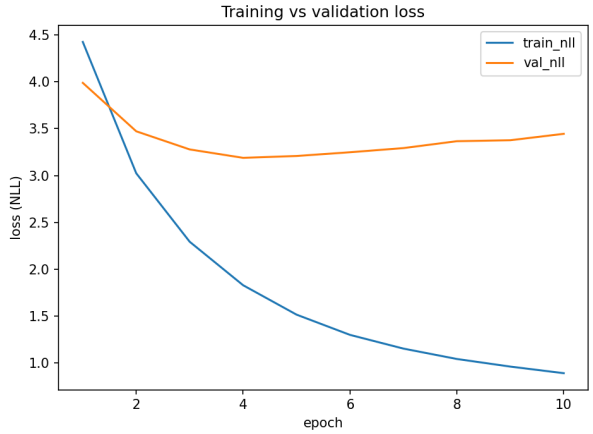
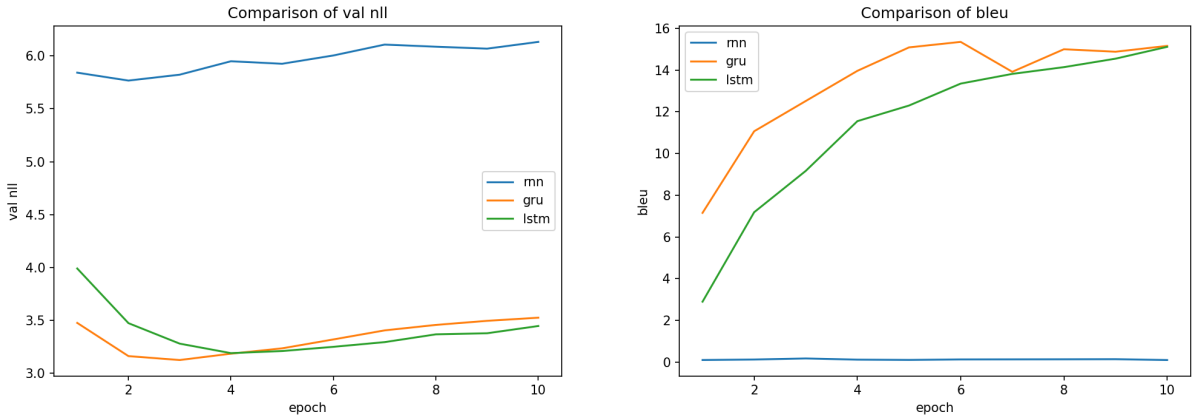


Figure 5.8: LSTM Training versus validaiton loss

Training LSTMs in this case seems to be effective up to 4 epochs, after which the system converges on the validation set.

We can now also compare the BLEU and loss scores for RNNs, GRUs and LSTMs under identical experimental conditions, resulting in the Figures in 5.9.



(a) Validation loss over epochs

(b) BLEU score over epochs

Figure 5.9: Comparing GRU with RNN

From these graphs we can conclude that vanilla RNNs function significantly worse than GRUs and LSTMs. The difference between these latter two does not seem to be very large. This may be due to the fact that Tatoeba sentences tend to be rather short, so the gain of using LSTMs is limited.

GRU vs. LSTM in practice. In many MT settings:

- GRUs train slightly faster and use fewer parameters;
- LSTMs can be more stable for very long sentences;
- differences in final BLEU scores are often modest.

For this reason, both architectures have been widely used in neural machine translation.

5.4 Bidirectional Encoders

Bidirectional recurrent networks were originally introduced for general sequence modelling tasks by Schuster and Paliwal (1997). The first neural machine translation systems based on encoder–decoder architectures (Cho et al., 2014; Sutskever et al., 2014) employed unidirectional encoders that processed the source sentence from left to right. Shortly thereafter, bidirectional encoders were adopted in neural machine translation to improve the quality of source sentence representations.

Early influential NMT models incorporating bidirectional encoders include the systems of Bahdanau et al. (2015) and Luong et al. (2015), which demonstrated that processing the source sentence in both directions leads to more informative encodings and improved translation quality. As a result, bidirectionality became a standard component of recurrent neural machine translation architectures.

So far, all encoder–decoder models we have considered process the source sentence from left to right. At each time step, the encoder's hidden state summarises the input seen *so far*, but has no access to future words. This asymmetry turns out to be a limitation for many language pairs, especially when important information appears late in the source sentence.

Bidirectional recurrent networks address this limitation by processing the source sentence in *both directions*.

5.4.1 Motivation

Consider the following English example: `The book that I bought yesterday is expensive.`

When reading from left to right, the encoder does not encounter the main verb *is* until late in the sentence. However, this verb is crucial for generating a correct Dutch translation, particularly for word order and verb placement.

A unidirectional encoder must compress all earlier information without knowing what comes next. A bidirectional encoder, by contrast, has access to both past and future context at every position.

5.4.2 What Does Bidirectional Mean?

A bidirectional recurrent encoder consists of two separate RNNs:

- a **forward RNN** that reads the source sentence from left to right;

- a **backward RNN** that reads the source sentence from right to left.

For each position in the sentence, the hidden states from these two RNNs are combined (typically by concatenation). As a result, the encoder representation at each position contains information about:

- the words that precede the current position, and
- the words that follow it.

This makes the source representation more informative, without changing the decoder architecture.

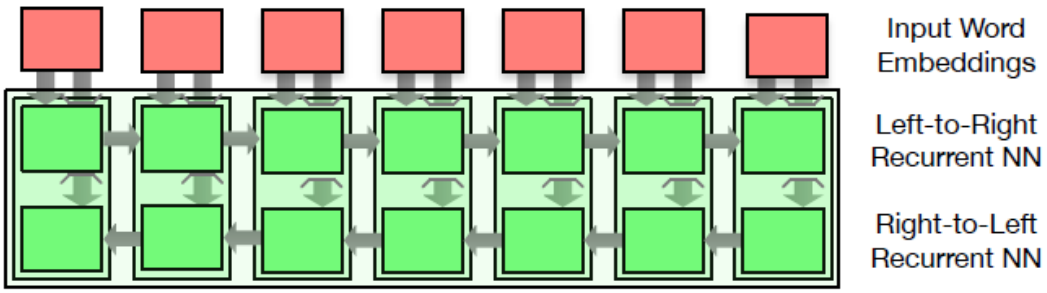


Figure 5.10: Bidirectional Encoder: It consists of two recurrent neural networks, running right to left and left to right. The encoder states are the combination of the two hidden states of the recurrent neural networks. From **koehn2020neural**.

5.4.3 Bidirectionality in Encoder–Decoder NMT

In neural machine translation, bidirectionality is almost always applied *only to the encoder*:

- The encoder benefits from seeing the full source sentence in both directions.
- The decoder must remain left-to-right, because it generates the target sentence sequentially.

From an architectural point of view, nothing else changes: the decoder still starts from an initial state derived from the encoder, and training still uses teacher forcing.

With a bidirectional encoder, the decoder is initialised with a state that combines the final forward and backward encoder states, for instance, by concatenation. This state encodes information from both directions of the source sentence.

5.4.4 Using a Bidirectional Encoder in Our Experiments

In our training script, enabling a bidirectional encoder is straightforward: we just need to add the option `-bidirectional`. Note that we continue using `-rnn-type lstm`.

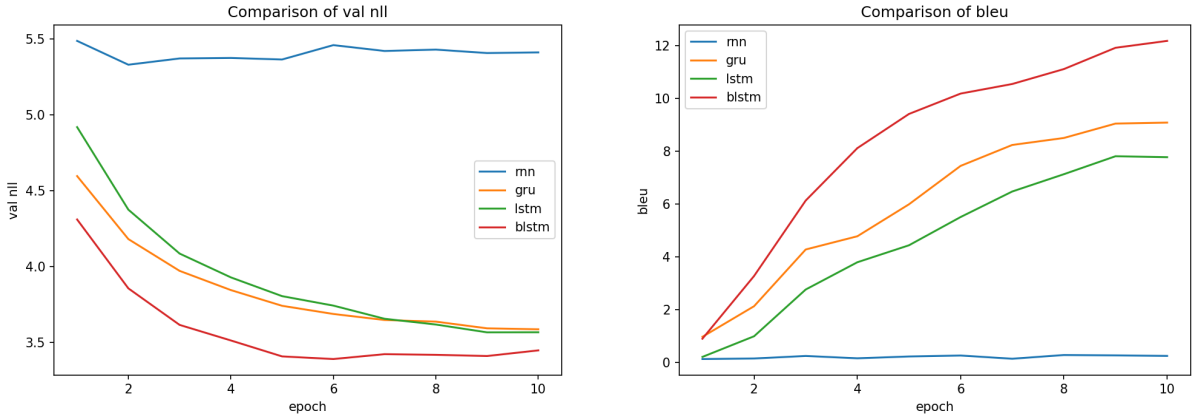
```
!python rnn_seq2seq.py --src-file /kaggle/input/tatoeba-en-nl/train.en --tgt-file
↪  /kaggle/input/tatoeba-en-nl/train.nl \
--src-val /kaggle/input/tatoeba-en-nl/dev.en --tgt-val
↪  /kaggle/input/tatoeba-en-nl/dev.nl \
--rnn-type lstm \
--epochs 10 \
--save blstm.pt \
--show-val-examples 5 \
```

```
--eval-metrics \
--lower \
--bidirectional \
--history-json blstm.hist
```

This option replaces the unidirectional encoder RNN with a bidirectional one. All other hyperparameters can be kept identical, allowing for a controlled comparison.

Figure 5.11: Validation loss over epochs for a Bidirectional LSTM model

As we can see from Figure 5.11, the validation loss improves over the few first epochs and then converges, while the training loss further improves, leading to overfitting.



(a) Validation loss over epochs

(b) BLEU score over epochs

Figure 5.12: Comparing Bidirectional LSTM with previous techniques

Figure 5.12 shows us the improvements of adding bidirectionality over the previous approaches. We see in Figure 5.12a that the validation loss is substantially lower for the bidirectional model, and a similar observation can be made about the BLEU score in Figure 5.12b, where we see a substantially better BLEU score for the bidirectional model compared to the prevous approaches.

5.4.5 Limitations

While bidirectional encoders improve source representations, they do not remove the fundamental bottleneck of encoder–decoder models: the decoder still relies on a fixed-size summary of the

source sentence. This limitation motivates the introduction of attention mechanisms, which we discuss in the next section.

5.5 Cross-Lingual Attention

5.5.1 Introduction

Cross-lingual attention was first introduced into neural machine translation by Bahdanau et al. (2015). Shortly thereafter, alternative formulations were proposed, notably by Luong et al. (2015), which differ in how the relevance between decoder states and encoder states is computed. Despite these variations, the core idea remains the same: target words are generated by dynamically attending to the source sentence.

Early encoder–decoder models rely on a single fixed-size vector to summarise the entire source sentence. Even when using gated recurrent units or bidirectional encoders, this design introduces a fundamental limitation: all source information must be compressed into one representation before decoding begins. As sentences become longer or structurally more complex, this information bottleneck increasingly degrades translation quality.

Cross-lingual attention was introduced to address this limitation by allowing the decoder to directly access different parts of the source sentence during translation. This idea is motivated by human translation behaviour: when producing a target word, translators typically focus on a specific source word or phrase rather than on a global sentence summary. In early encoder–decoder NMT systems, such selective focus was impossible—once encoding was complete, the decoder could not revisit the source sentence.

With attention, the encoder no longer produces a single vector but a sequence of hidden states, one for each source position, forming a distributed representation of the input. At each decoding step, the decoder computes a set of weights over these encoder states, indicating how strongly each source position contributes to the prediction of the current target word. The resulting weighted combination of encoder states, known as the *context vector*, enables the decoder to condition each target word on the most relevant parts of the source sentence.

5.5.2 Attention as Cross-Lingual Alignment

Cross-lingual attention can be interpreted as a form of *soft alignment* between source and target tokens, as shown in Figure 5.13.

For each target position, the attention mechanism produces a probability distribution over source positions:

- high weights indicate strong relevance,
- low weights indicate weak or no relevance.

Unlike traditional word alignment models in statistical MT, attention does not commit to a single alignment point. Instead, it distributes probability mass across multiple source positions, allowing for many-to-one and one-to-many correspondences. This is particularly important for languages with different word orders or rich morphology.

Worked Example: Cross-lingual Attention

Consider the translation of the English sentence:

Source (EN): `She eats apples`

into Dutch:

Target (NL): `Zij eet appels`

During decoding, the model generates the target sentence one word at a time. At each decoding step, an attention distribution is computed over the encoder hidden states corresponding to the source tokens `she`, `eats`, and `apples`.

Attention weights when predicting `eet`:

Source word	`she`	`eats`	`apples`
Attention weight	0.15	0.70	0.15

The decoder focuses primarily on the source word `eats`, which provides the most relevant information for predicting the target verb `eet`.

Attention weights when predicting `appels`:

Source word	`she`	`eats`	`apples`
Attention weight	0.10	0.15	0.75

Here, the attention distribution shifts toward the source word `apples`, allowing the decoder to generate the correct noun in the target language.
At each step, the context vector is computed as a weighted sum of the encoder states, enabling dynamic, word-level alignment between source and target sentences. This soft alignment is visualised in Figure 5.13.

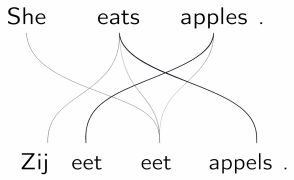


Figure 5.13: Soft alignment for `She eats apples` → `Zij eet appels`

The encoder–decoder models introduced so far rely on a single fixed-size vector to summarise the entire source sentence. Even with gated recurrent units and bidirectional encoders, this design imposes a fundamental limitation: all source information must be compressed into one representation before decoding begins. As sentences become longer or structurally more complex, this bottleneck degrades translation quality.

Cross-lingual attention was introduced to address this limitation by allowing the decoder to directly access different parts of the source sentence during translation.

5.5.3 How Attention Fits into the Encoder–Decoder Architecture

With the introduction of attention, the role of the encoder changes subtly. Rather than producing a single summary vector for the entire source sentence, the encoder outputs a sequence of

representations, one for each source position. The decoder therefore no longer relies exclusively on its initial hidden state to encode all source information. Instead, relevant source information is retrieved dynamically at each decoding step through the attention mechanism, allowing the decoder to adapt its focus as the translation unfolds.

The decoder's hidden state and the context vector are combined to predict the next target word. Importantly, the decoder remains a left-to-right conditional language model; attention augments it with source-side context rather than replacing it.

5.5.4 Practical Consequences for Translation

Introducing cross-lingual attention leads to several practical improvements in neural machine translation. By allowing the decoder to access source information directly at each time step, attention enables more robust handling of long sentences and reduces the risk of omitting relevant source content. It also contributes to improved word order and overall adequacy by grounding each target word in the most relevant parts of the source sentence. In addition, the attention weights provide interpretable alignment patterns that can be visualised, offering valuable insight into the model's translation behaviour.

These gains are often larger than those obtained from increasing model depth or hidden size, making attention one of the most influential innovations in recurrent neural machine translation. The gains of adding attention in our experiments are shown in Figure 5.15.

5.5.5 Cross-lingual Attention in practice

In the script that we use we can activate the attention mechanism of Luong et al. (2015) by using the option `-attention luong`. We keep using a bidirectional LSTM model.

```
!python rnn_seq2seq.py --src-file /kaggle/input/tatoeba-en-nl/train.en --tgt-file
↪  /kaggle/input/tatoeba-en-nl/train.nl \
--src-val /kaggle/input/tatoeba-en-nl/dev.en --tgt-val
↪  /kaggle/input/tatoeba-en-nl/dev.nl \
--rnn-type lstm \
--epochs 10 \
--save blstm_att.pt \
--show-val-examples 5 \
--eval-metrics \
--lower \
--bidirectional \
--history-json blstm_att.hist \
--attention luong
```

Similar as before, Figure 5.14 shows overfitting. Initially we see improvement for both training and validation loss, but already after two epochs validation loss seems to more or less converge.

Compared to the previous approaches, Figure 5.15 shows substantial further improvement that can be attributed to the attention mechanism, given that all other factors stay the same. This holds both for validation loss score, in Figure 5.15a as for BLEU scores in Figure 5.15b.

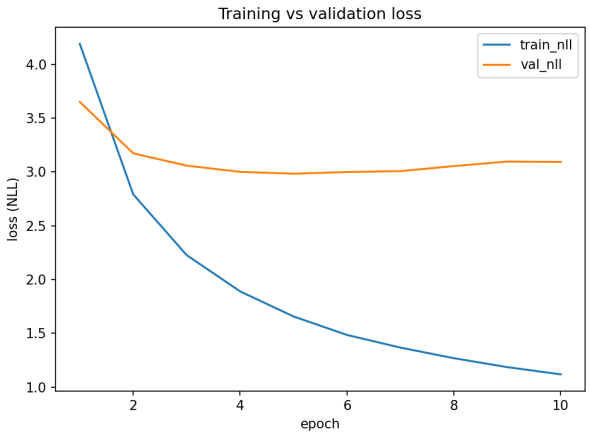
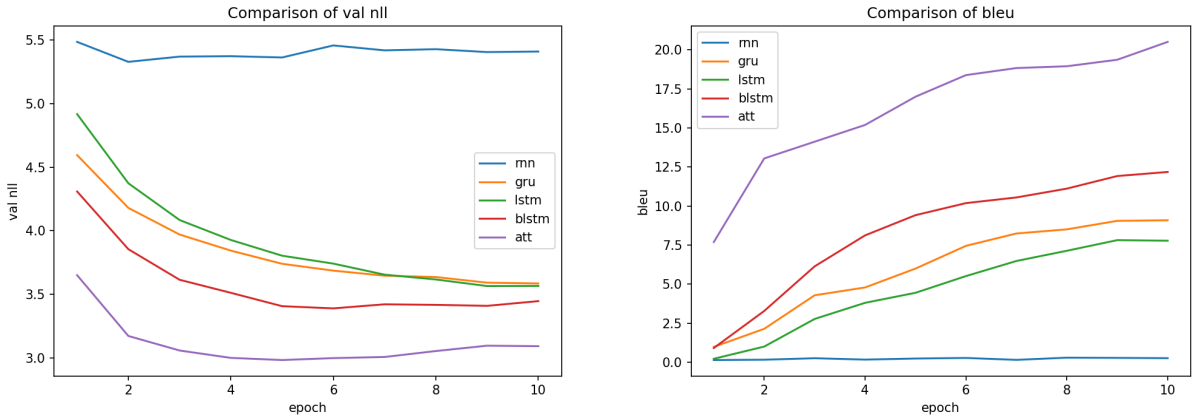


Figure 5.14: Validation versus Training loss over epochs for LSTM Bidirectional Attention



(a) Validation loss over epochs

(b) BLEU score over epochs

Figure 5.15: Comparing Bidirectional LSTM with attention with previous techniques

5.5.6 Relation to Earlier Approaches

Attention-based NMT systems reintroduce an explicit notion of alignment into neural translation models. In this sense, attention can be seen as a neural counterpart to alignment models used in statistical machine translation, but learned jointly with the translation model and optimised end-to-end.

This connection highlights how neural MT integrates ideas from earlier MT paradigms while overcoming many of their limitations.

Chapter 6

Subwording and Transformers

6.1 Subwording in Neural Machine Translation

6.1.1 Motivation

Early neural machine translation systems typically operated on *word-level vocabularies*. While conceptually simple, this design choice quickly reveals several fundamental weaknesses when applied to realistic translation tasks.

First, word-level models suffer from the **out-of-vocabulary (OOV) problem**. Because the vocabulary is fixed at training time, any word that is not among the most frequent N training words is replaced by a generic `<unk>` token. This disproportionately affects:

- proper names,
- domain-specific terminology,
- inflected word forms,
- compounds (especially in languages such as German or Dutch).

As a result, word-level systems often produce translations that are fluent but semantically incomplete.

Second, word-level modelling exacerbates **data sparsity**. From the model's perspective, related word forms such as *run*, *runs*, *running*, and *ran* are completely independent symbols. Learning reliable representations for each of these forms requires large amounts of training data, which is often unavailable.

Finally, large word vocabularies significantly increase **model size and computational cost**. Embedding matrices and output layers scale linearly with vocabulary size, making training slower and more memory-intensive.

Subwording addresses all three issues simultaneously.

6.1.2 The Core Idea of Subwording

The central idea behind subwording is to abandon the assumption that the word is the smallest meaningful unit for translation. Instead, words are decomposed into **subword units** that occur frequently across many word forms.

These subword units occupy a middle ground:

- Compared to words, they are more flexible and allow generalisation to unseen forms;
- Compared to characters, they retain linguistic structure and keep sequences at a manageable length.

For example, the English word `unbelievable` might be segmented as:

```
un + believe + able
```

Crucially, a previously unseen word such as `unbelievability` can still be processed by recombining known subword units. From a modelling perspective, this means that:

- the vocabulary becomes much smaller,
- every input string can be represented without `<unk>` tokens,
- morphological regularities can be learned implicitly.

Subword units are not defined manually. Instead, they are learned automatically from the training corpus based on frequency statistics.

6.1.3 Byte Pair Encoding

Byte Pair Encoding (BPE) was one of the first widely adopted subword segmentation methods in neural machine translation. Starting from a character-level representation, BPE:

1. identifies the most frequent adjacent symbol pair,
2. merges it into a new symbol,
3. repeats this process for a fixed number of merge operations.

The result is a vocabulary consisting of characters, frequent character sequences, and full words where appropriate.

BPE Trace with Pair Frequencies and Updated Corpus

Corpus (whitespace-tokenised, characters + </w>):

```
R e s u m p t i o n </w> o f </w> t h e </w> s e s s i o n </w> .  </w> Y o u
</w> h a v e </w> r e q u e s t e d </w> a </w> d e b a t e </w> o n </w> t h i
s </w> s u b j e c t </w> i n </w> t h e </w> c o u r s e </w> o f </w> t h e
</w> n e x t </w> f e w </w> d a y s </w> , </w> d u r i n g </w> t h i s </w> p
a r t - s e s s i o n </w> .  </w>
```

Step 1: Merge `(e, </w>)` → `e</w>`

Pair	Freq.	Sel.
(e, </w>)	6	⋆
(n, </w>)	5	
(t, h)	5	
(e, s)	4	
(o, n)	4	
(i, o)	3	

Updated corpus:

R e s u m p t i o n </w> o f </w> t h e</w> s e s s i o n </w> . </w> Y o u </w> h a v e</w> r e q u e s t e d </w> a </w> d e b a t e</w> o n </w> t h i s </w> s u b j e c t </w> i n </w> t h e</w> c o u r s e</w> o f </w> t h e</w> n e x t </w> f e w </w> d a y s </w> , </w> d u r i n g </w> t h i s </w> p a r t - s e s s i o n </w> . </w>

Step 2: Merge (n, </w>) → n</w>

Pair	Freq.	Sel.
(n, </w>)	5	⋆
(t, h)	5	
(e, s)	4	
(o, n)	4	
(i, o)	3	
(h, e</w>)	3	

Updated corpus:

R e s u m p t i o n</w> o f </w> t h e</w> s e s s i o n</w> . </w> Y o u </w> h a v e</w> r e q u e s t e d </w> a </w> d e b a t e</w> o n</w> t h i s </w> s u b j e c t </w> i n</w> t h e</w> c o u r s e</w> o f </w> t h e</w> n e x t </w> f e w </w> d a y s </w> , </w> d u r i n g </w> t h i s </w> p a r t - s e s s i o n</w> . </w>

Step 3: Merge (t, h) → th

Pair	Freq.	Sel.
(t, h)	5	⋆
(e, s)	4	
(o, n</w>)	4	
(i, o)	3	
(h, e</w>)	3	
(s, </w>)	3	

Updated corpus:

R e s u m p t i o n</w> o f </w> th e</w> s e s s i o n</w> . </w> Y o u </w> h a v e</w> r e q u e s t e d </w> a </w> d e b a t e</w> o n</w> th i s </w> s u b j e c t </w> i n</w> th e</w> c o u r s e</w> o f </w> th e</w> n e x t </w> f e w </w> d a y s </w> , </w> d u r i n g </w> th i s </w> p a r t - s e s s i o n</w> . </w>

Step 4: Merge `(th, e</w>)` → `the</w>`

Pair	Freq.	Sel.
`(e, s)`	4	
`(o, n</w>)`	4	
`(i, o)`	3	
`(th, e</w>)`	3	⋆
`(s, </w>)`	3	
`(s, u)`	2	

Updated corpus:

```
R e s u m p t i o n</w> o f </w> the</w> s e s s i o n</w> .  </w> Y o u </w> h
a v e</w> r e q u e s t e d </w> a </w> d e b a t e</w> o n</w> th i s </w> s u
b j e c t </w> i n</w> the</w> c o u r s e</w> o f </w> the</w> n e x t </w> f e
w </w> d a y s </w> , </w> d u r i n g </w> th i s </w> p a r t - s e s s i o
n</w> .  </w>
```

Step 5: Merge `(e, s)` → `es`

Pair	Freq.	Sel.
`(e, s)`	4	⋆
`(o, n</w>)`	4	
`(i, o)`	3	
`(s, </w>)`	3	
`(s, u)`	2	
`(o, f)`	2	

Updated corpus:

```
R es u m p t i o n</w> o f </w> the</w> s es s i o n</w> .  </w> Y o u </w> h a
v e</w> r e q u es t e d </w> a </w> d e b a t e</w> o n</w> th i s </w> s u b j
e c t </w> i n</w> the</w> c o u r s e</w> o f </w> the</w> n e x t </w> f e w
</w> d a y s </w> , </w> d u r i n g </w> th i s </w> p a r t - s es s i o n</w>
.  </w>
```

Step 6: Merge `(o, n</w>)` → `on</w>`

Pair	Freq.	Sel.
`(o, n</w>)`	3	⋆
`(s, s)`	3	
`(i, o)`	3	
`(s, </w>)`	3	
`(o, f)`	2	
`(f, </w>)`	2	

Updated corpus:

```
R es u m p t i on</w> o f </w> the</w> s es s i on</w> .  </w> Y o u </w> h a v
e</w> r e q u es t e d </w> a </w> d e b a t e</w> on</w> th i s </w> s u b j e
c t </w> i n</w> the</w> c o u r s e</w> o f </w> the</w> n e x t </w> f e w
</w> d a y s </w> , </w> d u r i n g </w> th i s </w> p a r t - s es s i on</w> .
</w>
```

Step 7: Merge `(s, u)` → `su`

Pair	Freq.	Sel.
`(s, u)`	2	⋆
`(s, s)`	3	
`(i, o)`	3	
`(s, </w>)`	3	
`(o, f)`	2	
`(o, u)`	2	

Updated corpus:

```
R es u m p t i on</w> o f </w> the</w> s es s i on</w> .  </w> Y o u </w> h a v
e</w> r e q u es t e d </w> a </w> d e b a t e</w> on</w> th i s </w> su b j e c
t </w> i n</w> the</w> c o u r s e</w> o f </w> the</w> n e x t </w> f e w </w>
d a y s </w> , </w> d u r i n g </w> th i s </w> p a r t - s es s i on</w> .
</w>
```

Step 8: Merge `(i, o)` → `io`

Pair	Freq.	Sel.
`(i, o)`	3	⋆
`(s, s)`	3	
`(s, </w>)`	3	
`(o, f)`	2	
`(o, u)`	2	
`(th, i)`	2	

Updated corpus:

```
R es u m p t io n</w> o f </w> the</w> s es s io n</w> .  </w> Y o u </w> h a v
e</w> r e q u es t e d </w> a </w> d e b a t e</w> on</w> th i s </w> su b j e c
t </w> i n</w> the</w> c o u r s e</w> o f </w> the</w> n e x t </w> f e w </w>
d a y s </w> , </w> d u r i n g </w> th i s </w> p a r t - s es s io n</w> .
</w>
```

6.1.4 SentencePiece Unigram Subwording

SentencePiece Unigram (Kudo and Richardson, 2018) is an alternative subword segmentation method that differs fundamentally from Byte Pair Encoding (BPE). Whereas BPE constructs subword units through a sequence of deterministic merge operations, the Unigram model starts from a large set of candidate subword pieces and gradually removes pieces that are least useful for explaining the training corpus.

A central idea of the Unigram approach is that segmentation is *probabilistic*. Instead of enforcing a single fixed segmentation during training, the model allows multiple possible segmentations for each word and learns which subword pieces are most helpful overall.

6.1.4.1 Corpus Representation

SentencePiece does not rely on external tokenization. Instead, it treats whitespace as an ordinary symbol by replacing it with a special boundary marker, commonly rendered as `_`. For example, the string:

> Resumption of the session

is internally represented as:

```
_Resumption _of _the _session
```

This makes the model independent of language-specific tokenization rules and allows it to learn word-initial subword units such as `_the` or `_session`.

6.1.4.2 Initial Subword Inventory

Training begins with a large inventory of candidate subword pieces. This inventory always contains all individual characters (including the boundary marker `_`), and typically also contains many longer character sequences that occur frequently in the corpus.

For the present toy example, we assume an initial inventory that includes:

- full words such as `_the`, `_of`, and `_session`,
- frequent suffixes such as `tion` and `ion`,
- shorter fragments such as `_se`, `ss`, and `sess`.

In real SentencePiece training, this inventory is much larger and is derived automatically from corpus statistics.

6.1.4.3 Competing Segmentations

Because the inventory contains both long and short pieces, a single word may be segmented in several different ways. For example, the word `_session` can be represented either as a single subword unit:

```
_session
```

or as a sequence of smaller pieces, such as:

```
_sess ion
```

or even:

```
_se ss ion
```

During training, SentencePiece considers all such alternatives rather than committing to one fixed segmentation.

6.1.4.4 Learning and Pruning

The Unigram model assigns a likelihood to each subword piece based on how useful it is for explaining the corpus. Pieces that are frequently used in good segmentations receive higher likelihood, while pieces that rarely contribute are down-weighted.

Training proceeds iteratively. After estimating the usefulness of each subword piece, a portion of the least useful pieces is removed from the inventory. The remaining pieces are then re-evaluated on the corpus, and the process repeats until a target vocabulary size is reached.

This pruning strategy allows the model to balance two competing goals:

- keeping whole-word units for very frequent words, and
- retaining smaller, reusable pieces for less frequent or novel words.

6.1.4.5 Final Segmentation

After training has converged, segmentation is obtained by selecting the most likely sequence of subword pieces for each input string. On the corpus fragment used throughout this chapter, a plausible outcome is that frequent function words remain intact, while rarer or longer words are decomposed into meaningful subparts:

- `_the` → `_the`
- `_of` → `_of`
- `_session` → `_session`
- `_requested` → `_request ed`
- `_Resumption` → `_Resum ption`

Comparison with BPE. While BPE builds subword units through a fixed sequence of merges, SentencePiece Unigram starts with many candidate pieces and removes them gradually. This probabilistic and pruning-based perspective makes Unigram particularly well suited for data augmentation techniques such as subword regularization and is one of the reasons it is widely used in modern neural machine translation systems.

6.1.5 Applying SentencePiece

Kaggle session

For this part, we start with a new Kaggle session, available HERE.

6.1.5.1 Installing the modules

As a first step we need to install the software to enable training the sentence piece model

```
!pip install sentencepiece
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/sentencepiece_train.py
```

- `!pip install sentencepiece` installs the sentencepiece python module

- `sentencepiece_train.py` is a script to make oiur lives easier when we need to train a SentencePiece model. It takes three arguments:
  - The input corpus to learn from
  - The prefix of the output file name it writes the model to
  - Optionally, the maximum vocabulary size (set by default to 8000).

We also install the scripts from the previous session.

```
!pip install sacrebleu
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/rnn_seq2seq.py
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/plot_train_val.py
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/plot_history.py
from IPython.display import Image, display
```

6.1.5.2 Training the SentencePiece model

We need to train the sentencepiece models so the `rnn_seq2seq.py` script can use them to subword the data. We do not need to apply the subwording to the data.

We do this for the source and target language separately

```
!python sentencepiece_train.py /kaggle/input/tatoeba-en-nl/train.en spm.en
!python sentencepiece_train.py /kaggle/input/tatoeba-en-nl/train.nl spm.nl
```

This results in two output files per language. For English these are `spm.en.model` and `spm.en.vocab`.

6.1.5.3 Training the MT system

Similarly to the previous chapter we train the NMT system, with the same options as before. We just add `-subword-type unigram` to tell the system that we want it to work with SentencePiece unigrams, and then the `-src-sp-model` and `-tgt-sp-model` options that point to the model files generated by the training step, for the source and target language respectively.

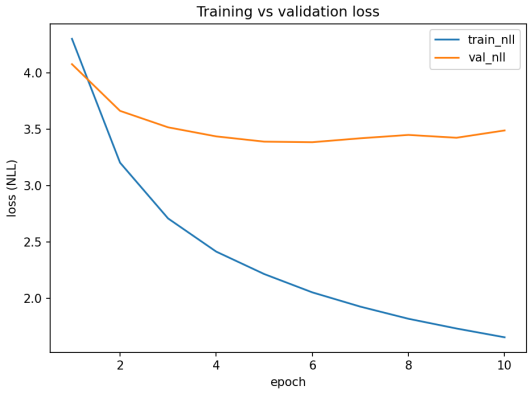
```
!python rnn_seq2seq.py --src-file /kaggle/input/tatoeba-en-nl/train.en --tgt-file
↪  /kaggle/input/tatoeba-en-nl/train.nl \
--src-val /kaggle/input/tatoeba-en-nl/dev.en --tgt-val
↪  /kaggle/input/tatoeba-en-nl/dev.nl \
--rnn-type lstm \
--epochs 10 \
--save blstm_att_sp.pt \
--show-val-examples 5 \
--eval-metrics \
--lower \
--bidirectional \
--history-json blstm_att_sp.hist \
--attention luong \
--subword-type unigram \
--src-sp-model spm.en.model \
--tgt-sp-model spm.nl.model
```

6.1.5.4 Results

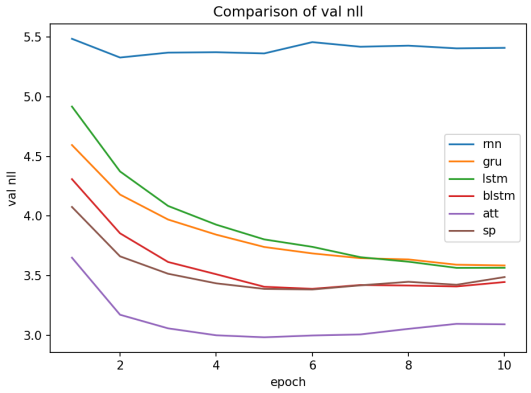
Also like before, we can now again compare training vs validation loss and create the charts that compare with the previous models. As we are in a new kaggle session, we have created a dataset that contains the history files built in the previous session, which can be used as input files for the current session.

```
!python plot_train_val.py blstm_att_sp.hist --save blstm_att_sp.png
display(Image(filename="blstm_att_sp.png"))
```

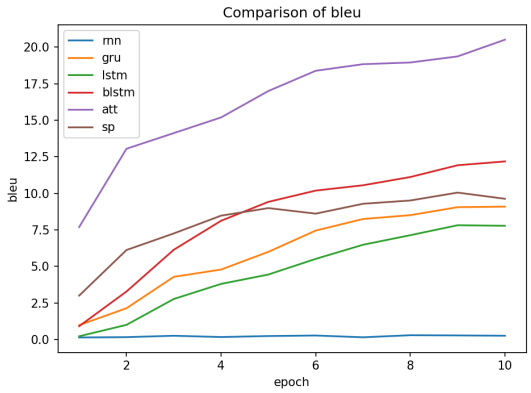
The results in Figure 6.1a show that also here the model overfits, after initial learning for about thee epochs.



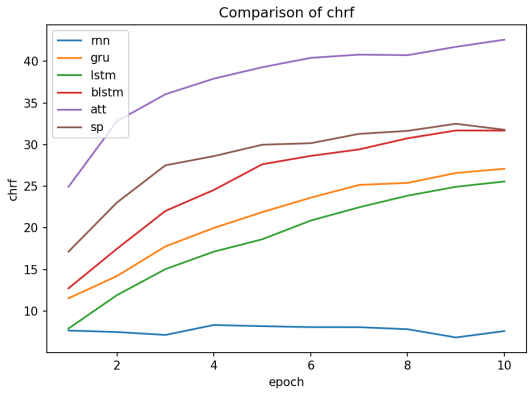
(a) Validation versus Training loss for the SentencePiece model



(b) Comparing NLL for the different systems



(c) BLEU scores comparison



(d) chrF score comparison

Figure 6.1: Results for the system with separately trained sentence piece models

Figures 6.1b, 6.1c and 6.1d show that subwording in the current case does not improve the results. We can attribute this to the fact that our training corpus is rather small and does not contain many different words.

6.1.5.5 Training a Joint Vocabulary

In NMT systems (and in multilingual AI systems in general) the subwording is often applied over languages, i.e. building a single vocabulary over the source and target language.

We can do this through the following steps:

1. Concatenate the two corpora, e.g. with the linux tool `!cat`
2. Train the SentencePiece model on the joint corpus
3. Train the NMT model, using the same joint model for both the `-src-sp-model` and the `-tgt-sp-model`.

```
!cat /kaggle/input/tatoeba-en-nl/train.en /kaggle/input/tatoeba-en-nl/train.nl >
↪  train.en_train.nl
!python sentencepiece_train.py train.en_train.nl spm.joint
!python rnn_seq2seq.py --src-file /kaggle/input/tatoeba-en-nl/train.en --tgt-file
↪  /kaggle/input/tatoeba-en-nl/train.nl \
--src-val /kaggle/input/tatoeba-en-nl/dev.en --tgt-val
↪  /kaggle/input/tatoeba-en-nl/dev.nl \
--rnn-type lstm \
--epochs 10 \
--save blstm_att_sp.pt \
--show-val-examples 5 \
--eval-metrics \
--lower \
--bidirectional \
--history-json blstm_att_spj.hist \
--attention luong \
--subword-type unigram \
--src-sp-model spm.joint.model \
--tgt-sp-model spm.joint.model
```

Similar as in Figure 6.1, we see in Figure 6.2 that the training overfits. We also see that the model using the joint subwording approach scores consistently better for NLL, BLEU and chrF.

6.2 Transformers

6.2.1 Motivation: beyond recurrent models

So far, we have seen that neural machine translation models evolved from simple encoder–decoder architectures to recurrent models with attention. While attention already alleviates many limitations of RNN-based models, *recurrence itself remains a bottleneck.*

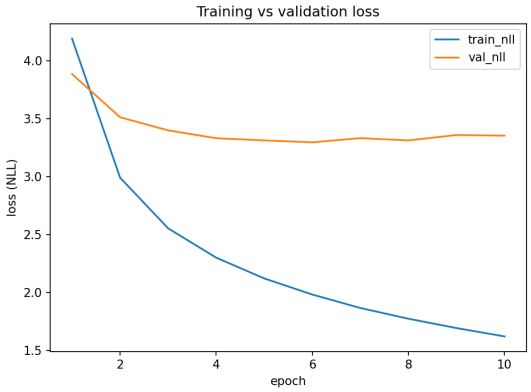
Recurrent models process tokens sequentially, which limits parallelisation, slows down training, and makes it harder to capture long-range dependencies. Transformers, introduced by Vaswani et al. (2017), address these issues by *removing recurrence entirely.* Instead, all tokens in a sentence are processed in parallel using attention mechanisms.

This architectural change leads to faster training, better scalability, and substantial improvements in translation quality.

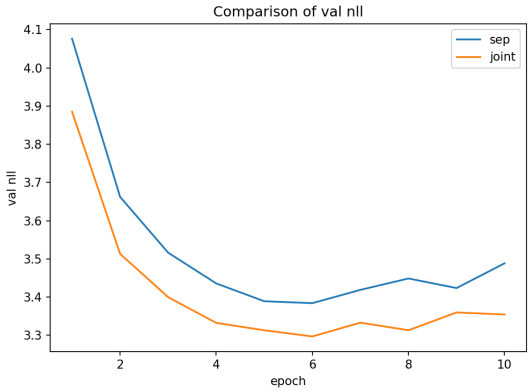
6.2.2 Core idea: attention is all you need

The central idea behind Transformers can be summarised as follows:

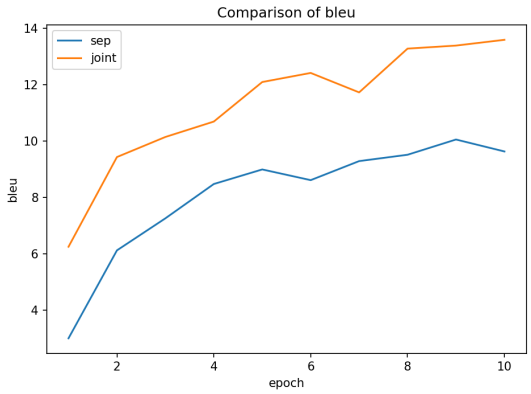
> *Each token directly attends to all other tokens in the sentence, independently of their distance.*



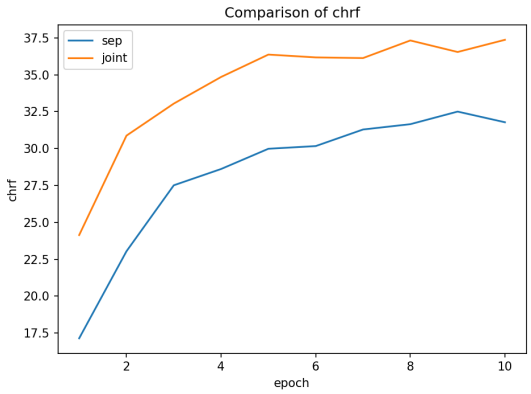
(a) Validation versus Training loss for the SentencePiece model



(b) Comparing NLL for the different systems



(c) BLEU scores comparison



(d) chrF score comparison

Figure 6.2: Results for the system with jointly trained sentence piece models, compared with separately trained

Rather than passing information through a sequence of hidden states, each token builds its representation by selectively focusing on other tokens in the same sentence. Long-distance dependencies are therefore handled as easily as local ones.

This mechanism is known as *self-attention*.

6.2.3 Self-attention

Self-attention allows each token in a sentence to compute a representation that takes into account *all other tokens in the same sentence*. Rather than processing tokens sequentially, all tokens are considered in parallel.

To illustrate this, consider the simple sentence:

```
She eats apples .
```

After subword tokenisation, embedding lookup, and positional encoding, each token is represented as a vector. Self-attention then updates these vectors by letting each token selectively focus on other tokens in the sentence.

6.2.3.1 Queries, keys, and values

For each token, three vectors are derived: a **query** vector, a **key** vector, and a **value** vector.

These vectors serve different roles in the attention mechanism.

Consider the token `eats`. When computing its new representation, the query vector of `eats` is compared to the key vectors of all tokens in the sentence (`she`, `eats`, `apples`, and the punctuation). This comparison determines how much attention `eats` pays to each token.

Intuitively:

- the query of `eats` expresses what information the verb is looking for;
- the keys of the other tokens describe what information they provide;
- the values contain the information that will be combined to form the new representation of `eats`.

Because `she` is the subject of the verb and `apples` is its object, their key vectors are likely to match the query of `eats` more strongly than the keys of less relevant tokens, such as punctuation.

6.2.3.2 Attention weights and contextual representations

Based on these query–key comparisons, the model computes attention weights over all tokens. The new representation of `eats` is then obtained as a weighted sum of the value vectors of all tokens.

As a result, the representation of `eats` encodes not only the meaning of the verb itself, but also information about its subject and object. In the same way, the token `apples` can attend to `eats` to capture its role as the object of the verb, and `she` can attend to `eats` to capture agreement and predicate information.

Self-attention therefore produces *contextualised representations*: the vector associated with a word depends on the entire sentence in which it occurs.

6.2.4 Example: self-attention and pronoun interpretation in translation

Self-attention is particularly important for machine translation because ambiguities in the source language often force explicit choices in the target language. Pronoun interpretation is a clear example of this phenomenon.

Consider the English sentences and their French translations in Figure 6.3.

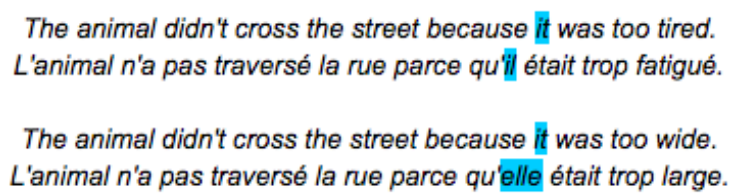


Figure 6.3: Two similar sentences which require pronoun resolution to be properly translated into French. Image from `https://research.google/blog/transformer-a-novel-neural-network-architecture-for-language-understanding/`

In English, the pronoun `it` is ambiguous and can refer to either `animal` or `street`. In French,

however, the translator must make this reference explicit by choosing between the masculine pronoun *il* and the feminine pronoun *elle*.

Figure 6.4 illustrates how self-attention helps resolve this ambiguity. The figure visualises the attention distribution for the token `it` in both sentences.

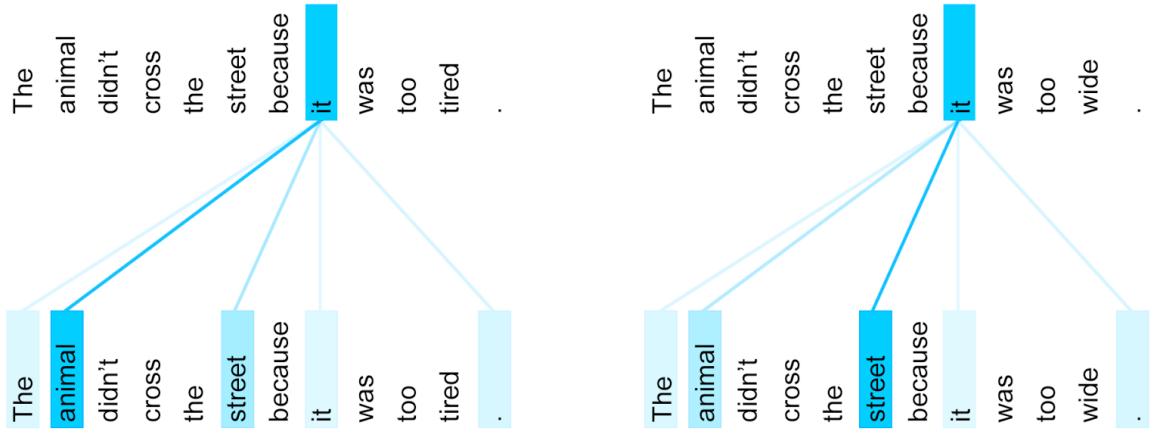


Figure 6.4: Self-attention patterns for the pronoun `it` in two similar sentences. In sentence (a), `it` attends most strongly to `animal`, whereas in sentence (b) it attends most strongly to `street`. Darker connections indicate higher attention weights. Image from `https://research.google/blog/transformer-a-novel-neural-network-architecture-for-language-understanding/`

Although the two English sentences differ only in the final adjective, the semantic interpretation of `it` changes. Being `too tired` is a property of the `animal`, while being `too wide` is a property of the `street`. Through self-attention, the Transformer assigns higher weights to the token that provides the most relevant contextual information.

As a result, the contextualised representation of `it` differs across the two sentences, allowing the translation system to select the appropriate gender in French. This illustrates how self-attention supports the resolution of anaphora and semantic ambiguity, both of which are central challenges in machine translation.

6.2.5 Multi-head attention

Instead of computing a single attention distribution, Transformers use *multi-head attention*. Several attention mechanisms operate in parallel, each with its own set of query, key, and value projections.

Different attention heads can specialise in different types of relations, such as subject–verb agreement, modifier–noun relations, or long-distance dependencies. The outputs of all heads are concatenated and linearly transformed to form the input to the next component in the model.

Example: parallel attention patterns Consider again the sentences shown in Figure 6.4:

```
The animal didn't cross the street because it was too tired.
The animal didn't cross the street because it was too wide.
```

For the token `it`, different attention heads within the same layer may focus on different parts of

the sentence. One head may attend strongly to `animal`, capturing a plausible semantic relation, while another head may attend to `street`, capturing an alternative interpretation.

Multi-head attention therefore allows the model to represent multiple possible relations *simultaneously*. At this stage, however, these parallel views are not yet combined into a single, unambiguous interpretation.

6.2.6 Multiple layers

While multi-head attention provides multiple perspectives in parallel, stacked layers allow the model to refine and integrate these perspectives *progressively*.

A Transformer encoder or decoder consists of a stack of identical layers. The output of one layer serves as the input to the next, allowing attention to be applied repeatedly over increasingly contextualised representations.

Example: progressive disambiguation across layers Returning to the pronoun `it` in Figure 6.4, lower layers may primarily capture surface-level associations and local co-occurrence patterns. At this level, both `animal` and `street` remain plausible antecedents.

In higher layers, the model integrates information from a broader context, including the semantic contribution of the adjectival phrases `too tired` and `too wide`. This enables later layers to favour the interpretation of `it` as referring to `animal` in the first sentence and to `street` in the second.

Stacking multiple layers therefore allows the model to move from shallow, ambiguous representations to more abstract and semantically informed ones.

6.2.7 Complementary roles of heads and layers

Multi-head attention and stacked layers serve complementary purposes in a Transformer architecture.

Multi-head attention increases the *breadth* of representation by capturing diverse relations in parallel within a single layer. Multiple layers increase the *depth* of processing by repeatedly transforming and refining these representations.

Both mechanisms are essential: without multiple heads, the model would miss important relational patterns; without multiple layers, it would struggle to combine these patterns into a coherent, contextually appropriate interpretation.

6.2.8 Transformer decoder

The Transformer decoder is also organised as a stack of layers. Each decoder layer contains three main components:

1. **Masked self-attention**, ensuring that each target token attends only to previously generated tokens.
2. **Encoder–decoder attention (cross-attention)**, allowing each target token to attend to the encoder output.
3. **A feed-forward network**. Attention decides what information to use; the feed-forward network decides how to transform that information.

As in the encoder, stacking multiple decoder layers allows the model to progressively integrate target-side context with source-side information. Lower layers focus more on local fluency and short-range dependencies, while higher layers increasingly incorporate global source context.

6.2.9 Autoregressive decoding

Like recurrent neural machine translation models, Transformers generate translations autoregressively: target tokens are produced one by one, each conditioned on the previously generated tokens. Decoding therefore remains sequential at inference time.

The key difference with RNN-based decoders lies in how past information is represented and accessed. In an RNN decoder, all information about the target prefix and the source sentence must be compressed into a single evolving hidden state that is passed from one time step to the next. As decoding progresses, this state is repeatedly updated, which can make it difficult to preserve long-range dependencies.

In a Transformer decoder, there is no recurrent hidden state. Instead, at each decoding step, the model uses masked self-attention to attend directly to *all previously generated target tokens*, and encoder–decoder attention to attend to the entire source sentence. Rather than propagating information through time, the model recomputes attention over the full available context at each step.

During training, this design allows all target positions to be processed in parallel by masking future tokens, something that is not possible with RNNs. During decoding, although token generation is still sequential, each prediction benefits from direct access to global source and target context.

Beam search is commonly used to explore multiple translation hypotheses during autoregressive decoding.

6.2.10 Why Transformers work well for machine translation

Transformers combine several architectural properties that make them particularly effective for machine translation. By replacing recurrence with self-attention, they avoid the information bottlenecks inherent in recurrent models and enable efficient parallelisation during training.

Multi-head attention allows the model to capture different types of linguistic relations in parallel, while stacked layers progressively refine these representations into more abstract and semantically informed ones. In the decoder, repeated integration of target-side context and source-side information supports stable and informed translation decisions.

Together with their natural compatibility with subword-based vocabularies, these properties allow Transformers to handle long sentences, long-distance dependencies, and complex reordering patterns more effectively than earlier neural architectures. As a result, Transformers have become the dominant architecture in modern machine translation.

Although originally proposed for machine translation, the Transformer architecture has since been adapted to encoder-only, decoder-only, and encoder–decoder models. Across these variants, self-attention remains the core mechanism underlying scalable and high-quality sequence modelling.

6.2.11 Hands-on Transformers

Kaggle session

For transformers we have created a new Kaggle session, which you can find HERE.

First we need to download and install the necessary modules.

```
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/plot_train_val_new.py
!wget http://ccl.kuleuven.be/~vincent/MTAT/transformer.py
from IPython.display import Image, display
!pip -q install -U evaluate sacrebleu
```

Then we run the `transformer.py` script. The command-line options are shown in Table 6.1.

Option	Default	Description
`-src-file`	–	Source language training file
`-tgt-file`	–	Target language training file
`-src-val`	–	Source language validation file
`-tgt-val`	–	Target language validation file
`-spm-src-model`	–	SentencePiece model for the source language
`-spm-tgt-model`	–	SentencePiece model for the target language
`-save`	model	Output directory where the model and tokenizers are saved
`-enc-layers`	6	Number of Transformer encoder layers
`-dec-layers`	6	Number of Transformer decoder layers
`-emb-size`	512	Embedding size
`-hidden-size`	2048	Feed-forward layer size inside Transformer blocks
`-num-heads`	8	Number of attention heads
`-batch-size`	32	Number of sentence pairs per training batch
`-epochs`	30	Number of training epochs
`-lr`	5e-4	Learning rate for Adam optimizer
`-max-src-len`	128	Maximum source sentence length (after subwording)
`-max-tgt-len`	128	Maximum target sentence length (after subwording)
`-lower`	False	Lowercase input text before tokenization
`-seed`	42	Random seed for reproducibility
`-eval-metrics`	False	Compute automatic metrics on validation data
`-show-val-examples`	0	Number of validation translations to print after each epoch
`-num-beams`	4	Beam size used for validation-time generation
`-max-gen-len`	128	Maximum length of generated translations
`-history-json`	None	Path to JSON file for saving training/validation history

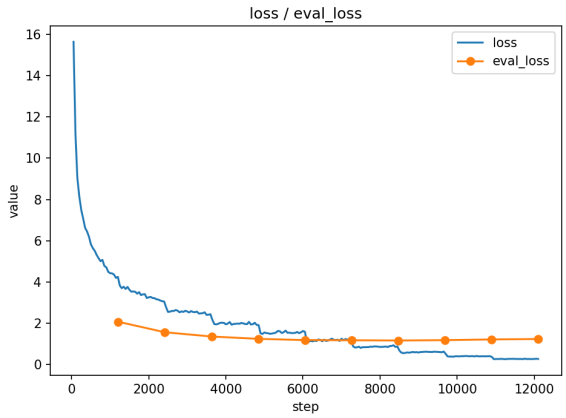
Table 6.1: Command-line options of `transformer.py`

```
!python transformer.py \
  --src-file /kaggle/input/tatoeba-en-nl/train.en \
  --tgt-file /kaggle/input/tatoeba-en-nl/train.nl \
  --src-val  /kaggle/input/tatoeba-en-nl/dev.en \
  --tgt-val  /kaggle/input/tatoeba-en-nl/dev.nl \
  --spm-src-model /kaggle/input/transformers-input-spms/spm.en.model \
  --spm-tgt-model /kaggle/input/transformers-input-spms/spm.nl.model \
  --save transformer_run \
  --epochs 3 \
  --eval-metrics \
  --show-val-examples 5 \
```

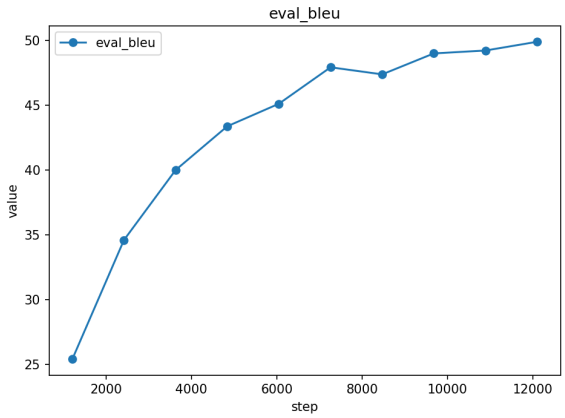
```
--history-json transformer.hist
```

Transformers are evaluated every fixed number of training steps rather than only at the end of an epoch, so we already see validation loss part-way through the epoch.

We show the results of training the transformer for 10 epochs on the Tatoeba data set for English to Dutch in Figure 6.5. We can see in Figure 6.5a that it takes several epochs before the training loss gets better than the loss on validation set. From Figure 6.5b we see that we reach Bleu scores of nearly 50, which is substantially higher than what we saw in the previous section, before using Transformers.



(a) Validation loss over epochs



(b) BLEU score over epochs

Figure 6.5: Results for Transformer MT system

6.3 Multilingual Neural Machine Translation

6.3.1 Introduction

Most of the world's languages are low-resource: for many language pairs, little or no parallel training data is available. Training a separate bilingual MT system for every language pair is therefore impractical, both economically and technically. Multilingual neural machine translation (MNMT) addresses this limitation by training a *single* model that can translate between multiple languages.

Beyond practical considerations, multilinguality also raises fundamental questions about representation learning: to what extent can a neural model share linguistic knowledge across languages, and under what conditions does such sharing improve or degrade translation quality?

In multilingual MT, a single encoder–decoder model is trained on parallel data from multiple language pairs. The core architectural idea remains unchanged: the encoder maps a source sentence to a sequence of representations, and the decoder generates a target sentence conditioned on those representations. What distinguishes multilingual MT from bilingual MT is that model parameters are shared across languages, vocabularies are often shared using subword units, and the model must be explicitly informed of the intended target language.

A common solution is to prepend a special *language tag* to the source sentence, indicating which language the model should generate. For example: `<2nl> She eats apples .`

The language tag conditions the decoder on the target language, allowing the same model to

produce translations in different languages depending on the tag provided. Despite its simplicity, this mechanism has proven highly effective and was central to early multilingual MT systems.

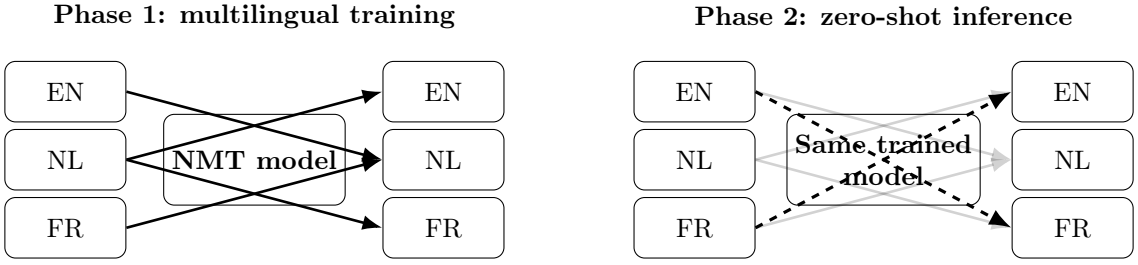
Multilingual MT systems almost always rely on shared subword vocabularies (See section 6.1). Sharing subword units across languages encourages the model to align similar morphemes, cognates, and function words. For closely related languages, this sharing can substantially improve translation quality. For unrelated languages with different scripts or typological properties, the benefits are more limited, but shared vocabularies still reduce model complexity and facilitate parameter sharing.

One of the most important effects of multilingual training is *cross-lingual transfer*: knowledge learned from high-resource language pairs can improve translation quality for low-resource language pairs. Positive transfer is more likely when languages share lexical, syntactic, or semantic properties. For example, training on multiple Germanic languages can improve performance for a low-resource Germanic language. At the same time, multilinguality can also lead to *negative transfer*, where competition between languages degrades performance, particularly for typologically distant languages or when high-resource languages dominate training.

6.3.2 Zero-shot translation

In multilingual neural machine translation (MNMT), a single model is trained jointly on multiple language pairs, sharing parameters across languages. An interesting consequence of this shared representation space is **zero-shot translation**: the model can sometimes translate between two languages for which it has never seen direct parallel training data.

For example, in Figure 6.6 if the model is trained on English–Dutch, Dutch-English, French–Dutch and Dutch-French data (Figure 6.6a, it may still learn to translate between English and French and French-English by routing both languages through the shared encoder–decoder space (Figure 6.6b. In the figures this corresponds to language pairs that are not explicitly connected by training arrows but are nevertheless reachable through the multilingual model block. Zero-shot translation thus emerges from parameter sharing and cross-lingual generalisation, rather than from explicit bilingual supervision.



(a) Training directions.

(b) Add EN↔FR zero-shot using the same model.

Figure 6.6: Multilingual MT: straight source-to-target flows that traverse a shared multilingual model; zero-shot directions appear after training on directions involving NL.

6.3.3 Model Capacity and Data Imbalance

Multilingual MT introduces additional challenges related to model capacity and data imbalance. High-resource languages tend to dominate training, which can harm performance for low-resource languages. Several mitigation strategies are commonly employed. Data sampling and temperature-based upsampling increase the relative frequency of low-resource languages during training,

ensuring that the model does not simply optimise for the largest language pairs. Language-specific adapters or routing mechanisms introduce small sets of parameters specialised for individual languages or language groups, reducing harmful interference while keeping most parameters shared. Increasing model capacity is a more general solution: when a model is too small, different languages must compete for the same parameters, whereas larger models can allocate sufficient representational capacity to multiple languages.

Together, these observations underline an important principle: multilinguality is not free. While sharing parameters across languages enables cross-lingual transfer and improves data efficiency, it also introduces competition within the model. Effective multilingual MT therefore requires careful balancing between shared and language-specific representations.

Evaluating multilingual MT systems is correspondingly more complex than evaluating bilingual systems. Performance can vary widely across languages, domains, and translation directions, and improvements for one language pair may coincide with degradation for another. As a result, evaluation is often reported per language or per direction, complemented by macro-averaged scores or targeted analysis of low-resource languages.

Multilingual MT laid important groundwork for later developments in neural machine translation. Shared vocabularies, language-agnostic representations, and cross-lingual transfer all predate explicit pretraining–fine-tuning paradigms.

In later chapters, these ideas will reappear in the context of large pretrained encoder–decoder models and decoder-only language models.

6.3.4 Hands-on Multilingual Models

Kaggle session

There is an example Kaggle session that you find HERE

Let's try out building a multilingual model.

6.3.4.1 Downloading and installing

Like previously, we first install all necessary code

```
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/plot_train_val_new.py
!wget http://ccl.kuleuven.be/~vincent/MTAT/transformer.py
from IPython.display import Image, display
!pip -q install -U evaluate sacrebleu
!pip install sentencepiece
!wget https://www.ccl.kuleuven.be/~vincent/MTAT/sentencepiece_train.py
```

6.3.4.2 Add target language tags to the data

We will now train the model on the English-Dutch data, in two directions, and on the French-Dutch data, also in two directions.

So, we need to tell the model which is the target language, so it can learn to translate in a specific language.

We therefore define the python function `add_targetlang_prefix` that takes three input arguments:

1. the name of the source file
2. the language tag to add
3. the name of the output file

```
def add_targetlang_prefix (file,target,outputfile):
    f=open(file)
    lines=f.readlines()
    lines_tgt=[f"{target} {s}" for s in lines]
    with open(outputfile, "w") as f:
        for line in lines_tgt:
            f.write(line)
```

With this function, we create variants of the training and development files with the target language tag. As shown in Table 6.2, we do this for the `train` and `dev` files in both language directions, for English-Dutch as well as for French-Dutch.

Original File	Tag	Ouput File
tatoeba-en-nl/train.en	<2nl>	en-nl-train.en_2nl
tatoeba-en-nl/dev.en	<2nl>	en-nl-dev.en_2nl
tatoeba-fr-nl/train.fr	<2nl>	fr-nl-train.fr_2nl
tatoeba-fr-nl/dev.fr	<2nl>	fr-nl-dev.fr_2nl
tatoeba-en-nl/train.nl	<2en>	en-nl-train.nl_2en
tatoeba-en-nl/dev.nl	<2en>	en-nl-dev.nl_2en
tatoeba-fr-nl/train.nl	<2fr>	fr-nl-train.nl_2fr
tatoeba-fr-nl/dev.nl	<2fr>	fr-nl.dev.nl_2fr

Table 6.2: Adding target language labels to the `train` and `dev` files

```
add_targetlang_prefix("/kaggle/input/tatoeba-en-nl/train.en",
                      "<2nl>",
                      "en-nl-train.en_2nl")
add_targetlang_prefix("/kaggle/input/tatoeba-en-nl/dev.en",
                      "<2nl>",
                      "en-nl-dev.en_2nl")
add_targetlang_prefix("/kaggle/input/tatoeba-fr-nl-prepared/tatoeba-fr-nl/train.fr",
                      "<2nl>",
                      "fr-nl-train.fr_2nl")
add_targetlang_prefix("/kaggle/input/tatoeba-fr-nl-prepared/tatoeba-fr-nl/dev.fr",
                      "<2nl>",
                      "fr-nl-dev.fr_2nl")

## REVERSE DIRECTIONS
add_targetlang_prefix("/kaggle/input/tatoeba-en-nl/train.nl",
                      "<2en>",
                      "en-nl-train.nl_2en")
add_targetlang_prefix("/kaggle/input/tatoeba-en-nl/dev.nl",
                      "<2en>",
                      "en-nl-dev.nl_2en")
```

```
add_targetlang_prefix("/kaggle/input/tatoeba-fr-nl-prepared/tatoeba-fr-nl/train.nl",
                      "<2fr>",
                      "fr-nl-train.nl_2fr")
add_targetlang_prefix("/kaggle/input/tatoeba-fr-nl-prepared/tatoeba-fr-nl/dev.nl",
                      "<2fr>",
                      "fr-nl-dev.nl_2fr")
```

6.3.4.3 Concatenate all training files and all dev file

Now we need to create two multilingual parallel corpora, which serve as the source and target side of the training and development data respectively. Note that in the source side each sentence is prefixed with a target language tag.

We do this by carefully concatenating the files so that the target sentences accord with the target language tag of the aligned source sentence.

We can use the linux `cat` command which takes any number of files as arguments and writes them to the output file specified after the > symbol (redirecting the standard output).

```
!cat en-nl-train.en_2nl fr-nl-train.fr_2nl \
     en-nl-train.nl_2en fr-nl-train.nl_2fr > train.2multi.src

!cat /kaggle/input/tatoeba-en-nl/train.nl \
    /kaggle/input/tatoeba-fr-nl-prepared/tatoeba-fr-nl/train.nl \
    /kaggle/input/tatoeba-en-nl/train.en \
    /kaggle/input/tatoeba-fr-nl-prepared/tatoeba-fr-nl/train.fr > train.2multi.tgt

!cat en-nl-dev.en_2nl fr-nl-dev.fr_2nl \
     en-nl-dev.nl_2en fr-nl-dev.nl_2fr > dev.2multi.src

!cat /kaggle/input/tatoeba-en-nl/dev.nl\
    /kaggle/input/tatoeba-fr-nl-prepared/tatoeba-fr-nl/dev.nl\
    /kaggle/input/tatoeba-en-nl/dev.en \
    /kaggle/input/tatoeba-fr-nl-prepared/tatoeba-fr-nl/dev.fr > dev.2multi.tgt
```

6.3.4.4 Train a joined SentencePiece Model

In the next step, we train a joined SentencePiece model for all the languages. We first concatenate the source and the target train files into one big file and train SentencePiece on that file.

(Concatenating source and target is maybe unnecessary, we could experimentally test whether this actually helps or not).

```
!cat train.2multi.src train.2multi.tgt > train.2multi.srctgt
!python sentencepiece_train.py train.2multi.srctgt spm.multi
```

6.3.4.5 Actual training

Now we can run the actual training. Note that we can use the same script as before, so only the data is changed compared to the single translation-pair approach, the training algorithm stays exactly as it was.

```
!python transformer.py \
  --src-file train.2multi.src \
  --tgt-file train.2multi.tgt \
  --src-val  dev.2multi.src \
  --tgt-val  dev.2multi.tgt \
  --spm-src-model spm.multi.model \
  --spm-tgt-model spm.multi.model \
  --save multi_nmt \
  --epochs 10 \
  --eval-metrics \
  --show-val-examples 10 \
  --history-json transformer.multi.hist
```

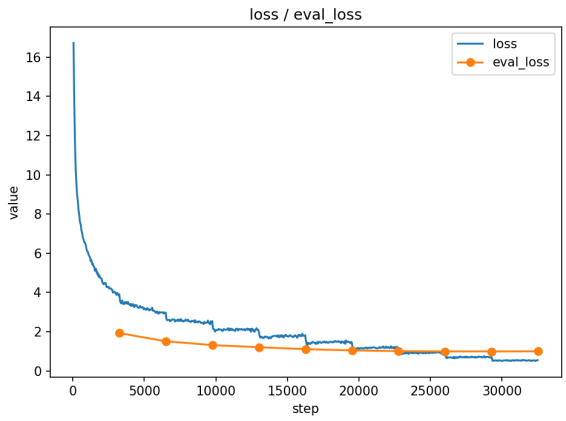
6.3.4.6 Compare results with previous approaches

```
!python plot_train_val_new.py transformer.multi.hist \
    --train-key loss --val-key eval_loss --save tfm_train_val.png

display(Image(filename="tfm_train_val.png"))
```

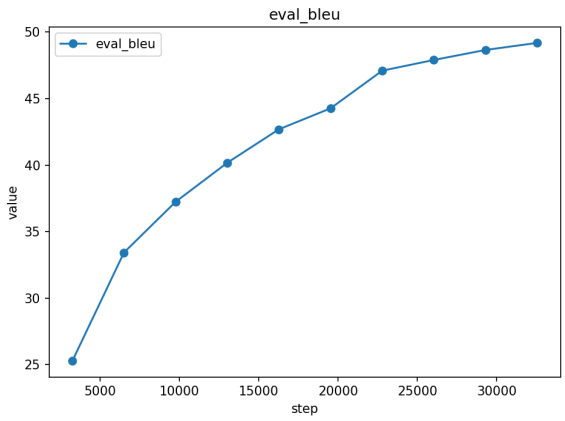
The resulting figure is shown in Figure 6.7a.

```
!python plot_train_val_new.py transformer.multi.hist \
  --val-key eval_bleu \
  --save tfm_bleu.png

display(Image(filename="tfm_bleu.png"))
```

The resulting figure is shown in Figure 6.7b.



(a) Validation loss over epochs



(b) BLEU score over epochs

Figure 6.7: Results for Multilingual Transformer MT system

We can see that the BLEU scores are now converging just below 50, whereas in Figure 6.5b, we converged just a bit higher. While this seems to indicate that we lost some of the translation quality compared to single translation-pair approach, we should be aware that results are not fully comparable, as in 6.7b results are presented for the combined development set that contains

four different translation directions, whereas results in 6.5b only evaluated a single translation direction.

6.3.4.7 Testing the zero-shot language pair

Now we have build a (small) multilingual model. So that was Phase 1 in Figure 6.6. Now we enter Phase 2, in which we actually test the English-to-French translation, for which no examples were seen in the training data.

Up till now, we have not yet translated with trained models outside of the training script. We can use the `transfomer_translate.py` script to actually translate files.

When we want to translate `dev.en` into French, we need to prefix the sentences with the `<2fr>` tag.

We can then run the `transformer_translate.py` script that takes the arguments shown in Table 6.3.

Argument	Default	Description
`-model-dir`	-	Directory passed as `-save` in `transformer.py`
`-src-file`	-	Source file to translate
`-out-file`	-	Output translations file
`-batch-size`	32	Batch size during translation
`-max-src-len`	128	Maximum nr of tokens in source
`-max-gen-len`	128	Maximum length of translation
`-num-beams`	4	Number of alternatives under consideration

Table 6.3: The options for the `transformer_translate.py` script.

```
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/
↪  transformer_translate.py

add_targetlang_prefix("/kaggle/input/tatoeba-en-nl/dev.en",
                      "<2fr>",
                      "en-nl-dev.en_2fr")

!python transformer_translate.py \
     --model multi_nmt \
     --src-file en-nl-dev.en_2fr \
     --out-file en-nl-dev.en_2fr.zeroshot.fr
```

After running the script, results can be found in `en-nl-dev.en_2fr.zeroshot.fr`.

In order to manually check the translation quality, we can peak inside the source file and the resulting translation file, using the `head` command.

```
!head -n 20 en-nl-dev.en_2fr
!head -n 20 en-nl-dev.en_2fr.zeroshot.fr
```

We can see that the translation coming out of this model are not very good when translation from English to French, but we can also see that the model has learned a little bit to do this.

The main reasons why the resulting model is not very good is due to the limited scale of the Tatoeba dataset, so there is not enough opportunity to learn proper representations of the

languages in the zero-shot case.

Chapter 7

Pretrained models

7.1 A brief chronology of pretrained models in machine translation

As we have seen in previous chapters, neural machine translation systems were initially trained *from scratch* using parallel corpora only. Early encoder–decoder architectures with attention (Bahdanau et al., 2015) relied exclusively on supervised translation data. A major architectural shift occurred with the introduction of the Transformer (Vaswani et al., 2017), which replaced recurrent networks with self-attention mechanisms and enabled more efficient training on large-scale datasets. This development laid the foundation for subsequent large pretrained models.

The next major breakthrough came from *self-supervised pretraining*. Contextual representation models such as ELMo (Peters et al., 2018) and, in particular, BERT (Devlin et al., 2019) demonstrated that language models trained on large amounts of monolingual text could learn general linguistic representations that transfer effectively to downstream tasks. Although BERT itself is not a generative translation model, it established the paradigm of large-scale pretraining followed by task-specific fine-tuning.

In parallel, autoregressive generative models such as GPT (Radford et al., 2018) showed that large language models trained to predict the next token can acquire substantial linguistic and even cross-lingual knowledge. While early GPT models were primarily monolingual, they demonstrated that generation capabilities could emerge from large-scale pretraining alone.

Researchers soon extended the pretraining paradigm directly to sequence-to-sequence architectures for translation. Models such as T5 (Raffel et al., 2020) and mBART (Liu et al., 2020) applied large-scale denoising or text-to-text objectives to encoder–decoder models before fine-tuning them for machine translation. These approaches significantly improved performance, especially in low-resource scenarios.

From 2021 onward, massively multilingual pretrained models further expanded the scope of translation. Systems such as mT5 (Xue et al., 2021), M2M-100 (Fan et al., 2021), and NLLB (Costa-jussà et al., 2022) demonstrated that a single model can support translation across hundreds of languages. More recently, very large language models such as GPT-3 (Brown et al., 2020) and PaLM (Chowdhery et al., 2022) have shown that translation capabilities can emerge even without explicit parallel training, particularly when guided through prompting or instruction tuning.

Overall, the development of pretrained MT models reflects a broader shift in natural language processing: from task-specific systems trained solely on parallel data to large, general-purpose pretrained models that are adapted to translation via fine-tuning or prompting.

7.2 The pretraining–fine-tuning paradigm

Modern pretrained encoder–decoder models are trained in two distinct stages: a large-scale *pretraining* phase followed by a task-specific *fine-tuning* phase. This training paradigm represents a fundamental shift from earlier neural machine translation systems, which were trained exclusively on parallel data from scratch.

During pretraining, the model is exposed to very large amounts of unlabelled text and trained using a self-supervised objective. Self-supervision means that the training signal is derived from the text itself rather than from manually annotated labels. For encoder–decoder models, this may take the form of a denoising objective: parts of the input text are corrupted, and the model is trained to reconstruct the original content. In this phase, the model does not learn translation. Instead, it learns general properties of language, such as syntactic structure, semantic relations, long-distance dependencies, and how to generate fluent text.

Fine-tuning constitutes the second stage. Here, the pretrained model is adapted to a specific downstream task using supervised data. In the case of machine translation, this involves training on parallel sentence pairs:

Fine-tuning for translation

Input: `The cat sits on the mat.`
Target: `De kat zit op de mat.`

Because the encoder already encodes meaningful sentence representations and the decoder already produces coherent text, fine-tuning does not start from random parameters. Instead, it builds on linguistic knowledge acquired during pretraining and focuses on learning the cross-lingual mapping between source and target languages.

This two-stage procedure has several important consequences. Training converges faster than when starting from scratch, performance improves in low-resource settings, and the resulting systems often generalize better beyond the exact domain of the training data. The models discussed in the following sections mainly differ in how they implement the pretraining phase: the specific objective used, the scale of the data, and whether the pretraining is monolingual or multilingual.

7.2.1 T5 (2019/2020): Text-to-Text Transfer Transformer

T5 (Raffel et al., 2020) was one of the first large-scale models to systematically explore how transfer learning can be unified across many NLP tasks within a single encoder–decoder architecture. Its central idea is simple but powerful: *every task is framed as text-to-text generation.*

Rather than designing task-specific output layers (e.g. classification heads), T5 treats all problems as conditional generation. A task is specified directly in the input as text, and the model produces the corresponding output sequence.

7.2.1.1 Pretraining objective

T5 adopts a span-corruption (denoising) objective. Random contiguous spans of tokens are removed from the input and replaced with special sentinel tokens. The decoder is trained to reconstruct the missing spans in order.

T5 span corruption example

Input: `Thank you <X> me to your party <Y> week.`
Target: `<X> for inviting <Y> last <Z>`

T5 reconstructs the missing spans autoregressively using the decoder. This ensures that both encoder and decoder are fully trained during pretraining, which makes the model naturally suitable for sequence-to-sequence tasks such as translation.

7.2.1.2 Task specification through prefixes

Because T5 uses a single unified text-to-text format, the task itself must be specified explicitly in the input. For translation, this is typically done using a natural-language prefix:

Translation fine-tuning example

Input: `translate English to Dutch: The cat sits on the mat.`
Target: `De kat zit op de mat.`

The prefix acts as an instruction that conditions the model's behaviour. Importantly, this prefix is not part of pretraining; it is introduced during fine-tuning to distinguish translation from other possible tasks.

7.2.1.3 Relevance for machine translation

T5 was not pretrained on parallel corpora, yet its encoder–decoder structure and generative training objective make it particularly well suited for adaptation to translation. The encoder already learns rich contextual representations, while the decoder learns fluent sequence generation. Fine-tuning then focuses primarily on learning the cross-lingual mapping between source and target languages.

In practice, T5 demonstrates that strong translation performance can be obtained even when pretraining is purely monolingual, provided that the model architecture supports conditional generation.

7.2.1.4 Hands-on: Fine-tuning T5 on Tatoeba EN–NL in Kaggle

Kaggle Session

You can find the Kaggle session for hands-on T5 training HERE

For the actual training we use a slightly adapted script, called `finetune_pretrained_t5.py`. For the actual translation of a file, we also used a somewhat adapted script, called `pretrained_translate.py`.

Download and install necessary modules

```
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/
↪  finetune_pretrained_t5.py
!pip install sacrebleu
```

```
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/
↪  pretrained_translate.py
```

Actual training

```
!python finetune_pretrained_t5.py \
  --src-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/train.en \
  --tgt-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/train.nl \
  --src-val /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/dev.en \
  --tgt-val /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/dev.nl \
  --epochs 5 --batch-size 16 --lr 5e-5 \
  --save t5_en_nl \
  --eval-metrics \
  --history-json t5.hist \
  --show-val-examples 5 \
  --max-src-len 128 --max-tgt-len 128 \
  --num-beams 4 --max-gen-len 128
```

Translation of a file (and evaluation)

```
!python pretrained_translate.py \
  --model-dir t5_en_nl \
  --src-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/test.en \
  --out-file test.t5.nl \
  --ref-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/test.nl \
  --metrics bleu,chrf \
  --batch-size 32 \
  --max-src-len 128 \
  --max-gen-len 128 \
  --num-beams 4
```

And again, we can take a peak at the first 20 translations, using the `head` command.

```
!head -n 20 test.t5.nl
```

7.2.2 mT5 (2021): Multilingual Text-to-Text Transfer Transformer

mT5 (Xue et al., 2021) extends the T5 framework to the multilingual setting. Architecturally, mT5 is identical to T5: it uses the same Transformer encoder–decoder structure and the same span-corruption pretraining objective, so we can use the exact same scripts as for T5. The key difference lies in the training data and linguistic scope.

7.2.2.1 Multilingual pretraining

While T5 is pretrained primarily on English data (the C4 corpus), mT5 is pretrained on the massively multilingual *mC4* corpus, which covers over 100 languages. The span-corruption objective remains unchanged: random contiguous spans are replaced by sentinel tokens and the decoder reconstructs the missing content.

Because pretraining data spans many languages, the model learns:

- cross-lingual lexical representations,
- shared subword units across languages,

- language-agnostic sentence structure patterns.

7.2.2.2 Vocabulary and tokenization

Another important difference concerns the tokenizer. T5 uses a SentencePiece model trained mostly on English text, whereas mT5 uses a multilingual SentencePiece vocabulary trained jointly on mC4. As a result:

- mT5 handles non-English scripts and morphologies more effectively,
- cross-lingual transfer becomes possible even before fine-tuning.

7.2.2.3 Implications for machine translation

Because mT5 has already seen many languages during pretraining, fine-tuning for translation between non-English languages typically requires fewer task-specific examples than with T5.

In practice:

- T5 is well suited for English-centric translation tasks.
- mT5 is preferable when translating between multiple languages or when the source or target language is not English.

7.2.2.4 Limitations

Despite its multilingual nature, mT5 is still pretrained using a denoising objective rather than explicit parallel translation data. It therefore requires supervised fine-tuning on parallel corpora to achieve competitive translation quality.

Overall, mT5 demonstrates that multilingual pretraining combined with a unified text-to-text framework enables broad cross-lingual transfer without modifying the underlying architecture.

7.2.2.5 Hands-on mT5

mT5 session on Kaggle

You can find the Kaggle session for hands-on mT5 training HERE

Download and install packages

```
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/
↪  finetune_pretrained_t5.py
!pip install sacrebleu
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/
↪  pretrained_translate.py
```

Actual training Note that we reduced the `-batch-size` to 8 in order not to run into memory problems.

```
!python finetune_pretrained_t5.py \
  --pretrained-model google/mt5-small \
  --src-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/train.en \
  --tgt-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/train.nl \
  --src-val /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/dev.en \
  --tgt-val /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/dev.nl \
  --epochs 3 --batch-size 8 --lr 5e-5 \
  --save mt5_en_nl \
  --eval-metrics --history-json mt5.hist \
  --show-val-examples 5
```

Translation and evaluation

```
!python pretrained_translate.py \
  --model-dir t5_en_nl \
  --src-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/test.en \
  --out-file test.t5.nl \
  --ref-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/test.nl \
  --metrics bleu,chrf \
  --batch-size 32 \
  --max-src-len 128 \
  --max-gen-len 128 \
  --num-beams 4
```

7.2.3 mBART (2020): Multilingual Denoising Pretraining for Seq2Seq

mBART (Liu et al., 2020) extends BART to the multilingual setting and was one of the first models to demonstrate that *multilingual denoising pretraining alone can substantially improve machine translation*.

Architecturally, mBART is a standard Transformer encoder–decoder model. Unlike T5/mT5, however, mBART was explicitly designed with multilingual sequence-to-sequence tasks such as translation in mind.

7.2.3.1 Pretraining objective

mBART is pretrained using a *denoising autoencoding objective*. Full monolingual documents from multiple languages are corrupted and the model is trained to reconstruct the original text.

The noise function includes:

- text infilling (masking contiguous spans),
- sentence permutation,
- token masking.

mBART denoising example

Original: `The cat sits on the mat.`
Corrupted input: `The <mask> on the mat sits cat.`
Target: `The cat sits on the mat.`

The encoder processes the corrupted multilingual input, and the decoder reconstructs the clean sentence. Because this is done across many languages, the model learns language-specific as well

as shared cross-lingual representations.

7.2.3.2 Multilingual pretraining

mBART is pretrained on large monolingual corpora covering many languages (e.g. CC25 in the original paper).

During pretraining:

- both encoder and decoder see data from multiple languages,
- language identity is encoded via special language tokens,
- parameters are shared across languages.

Unlike T5, mBART does *not* rely on natural-language task prefixes. Instead, it uses **explicit language tags** to control generation.

7.2.3.3 Language tags for translation

For translation, mBART requires:

- a source-language tag at the beginning of the input,
- a target-language tag to guide decoding.

Translation fine-tuning example (mBART)

Input: `<en_XX> The cat sits on the mat.`
Target: `<nl_NL> De kat zit op de mat.`

These tags are part of the tokenizer vocabulary and were already present during pretraining. This is an important difference from T5, where task prefixes are introduced only at fine-tuning time.

7.2.3.4 Relevance for machine translation

mBART was the first model to show that multilingual denoising pretraining followed by supervised fine-tuning on parallel corpora significantly improves translation quality, especially for low-resource languages.

Compared to T5/mT5:

- mBART is explicitly designed for multilingual sequence-to-sequence tasks,
- language control is handled via special tokens rather than text prefixes,
- pretraining more closely resembles the encoder–decoder dynamics of translation.

In practice, mBART tends to perform particularly well in multilingual and low-resource translation scenarios.

7.2.3.5 Limitations

Despite its strong multilingual pretraining, mBART:

- is still trained only on monolingual denoising data,
- requires supervised parallel data for high-quality translation,
- can be memory-intensive due to large vocabulary and multilingual scope.

Nevertheless, mBART marked an important step toward modern multilingual encoder–decoder LLMs.

7.2.3.6 Hands-on: Fine-tuning mBART on Tatoeba EN–NL in Kaggle

Kaggle Session

You can find the Kaggle session for hands-on mBART training HERE

For training we use the same script as for T5/mT5, but specify the pretrained mBART model.

Download and install necessary modules

```
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/
↪  finetune_pretrained_mbart.py
!pip install sacrebleu
!wget https://raw.githubusercontent.com/VincentCCL/MTAT/refs/heads/main/
↪  pretrained_translate.py
```

Actual training We use a smaller batch size to avoid GPU memory issues.

```
!python finetune_pretrained_mbart.py \
  --pretrained-model facebook/mbart-large-50-many-to-many-mmt \
  --src-lang en_XX --tgt-lang nl_XX \
  --src-file /kaggle/input/.../train.en \
  --tgt-file /kaggle/input/.../train.nl \
  --src-val  /kaggle/input/.../dev.en \
  --tgt-val  /kaggle/input/.../dev.nl \
  --epochs 3 --batch-size 4 --lr 3e-5 \
  --save mbart_en_nl \
  --eval-metrics --history-json mbart.hist \
  --show-val-examples 5 \
  --max-src-len 128 --max-tgt-len 128 \
  --num-beams 4 --max-gen-len 128
```

Translation of a file (and evaluation)

```
!python pretrained_translate.py \
  --model-dir mbart_en_nl \
  --src-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/test.en \
  --out-file test.mbart.nl \
  --ref-file /kaggle/input/datasets/vincentvandeghinste/tatoeba-en-nl/test.nl \
  --metrics bleu,chrf \
  --batch-size 16 \
  --num-beams 4
```

And again, we can inspect the first 20 translations:

```
!head -n 20 test.mbart.nl
```

7.2.4 M2M-100 (2021): Direct Many-to-Many Translation

M2M-100 (Fan et al., 2021) differs from T5/mBART in that it is pretrained directly on parallel translation data rather than purely monolingual denoising.

Training objective.

- Standard supervised translation objective (maximum likelihood on parallel data).
- Many-to-many training across 100 languages.

Unlike earlier multilingual systems, it does not rely on pivoting through English.

Significance. This model shows that large-scale multilingual training on parallel data alone can produce strong many-to-many performance.

Hands-on suggestion. Students can:

- Compare M2M-100 with mBART for a specific language pair.
- Evaluate zero-shot performance.
- Observe differences in vocabulary handling.

7.2.5 NLLB (2022): Scaling to Low-Resource Languages

The NLLB project (Costa-jussà et al., 2022) extended massively multilingual MT to 200+ languages, focusing especially on low-resource and underrepresented languages.

Training setup.

- Large-scale parallel data mining.
- Multilingual supervised training.
- Improved data filtering and balancing.

Unlike purely denoising-based pretraining, NLLB combines data curation and scale with multilingual supervision.

Impact. NLLB demonstrates:

- translation for hundreds of languages,
- strong zero-shot and low-resource performance,
- practical deployment readiness.

Hands-on suggestion. Students can:

- Fine-tune NLLB on a small domain-specific dataset.
- Compare performance before and after fine-tuning.
- Analyse domain adaptation effects.

7.2.6 Summary of pretraining strategies

Chronologically, encoder–decoder pretraining evolved as follows:

1. **Monolingual denoising (T5)**
2. **Multilingual denoising (mBART, mT5)**
3. **Massively multilingual supervised training (M2M-100)**
4. **Large-scale multilingual mining and supervision (NLLB)**

Across these models, three main pretraining paradigms emerge:

- Denoising autoencoding
- Text-to-text span corruption
- Multilingual supervised translation

Together, they illustrate the transition from "learning translation from scratch" to "adapting a large pretrained multilingual model".

7.3 Motivation: why pretraining?

Training neural machine translation (NMT) systems from scratch is data- and compute-intensive. High-quality parallel corpora are available only for a limited number of language pairs and domains, while many real-world translation scenarios involve low-resource languages, specialised domains, or rapidly changing content.

Using Pretrained models addresses these challenges by separating learning into two phases:

- **Pretraining**, in which a model is trained on very large amounts of data to acquire general linguistic and representational knowledge;
- **Fine-tuning**, in which the pretrained model is adapted to a specific translation task, language pair, or domain.

This paradigm is a form of *transfer learning*: knowledge acquired during pretraining is reused to improve performance, stability, and data efficiency in downstream MT tasks. In the previous chapter, Section 6.3 was also an example of transfer learning.

Figure 7.1 sketches the idea of transfer learning. In the case of pretrained models, Data 1 is a very large dataset on which Model 1 is trained, on a number of tasks, which may be very different from translation, such as explained in Section 7.6. The learned representational knowledge is then *transferred* to the new task, in our case translation. So Data 2 consists of translation examples, and the knowledge learned during pretraining is used as a starting point and *fine-tuned* on Data 2, leading to Model 2.

7.4 From multilingual NMT to explicit pretraining

Historically, multilingual NMT systems preceded explicit pretraining–fine-tuning approaches. Early multilingual models were trained directly on parallel data for multiple language pairs using

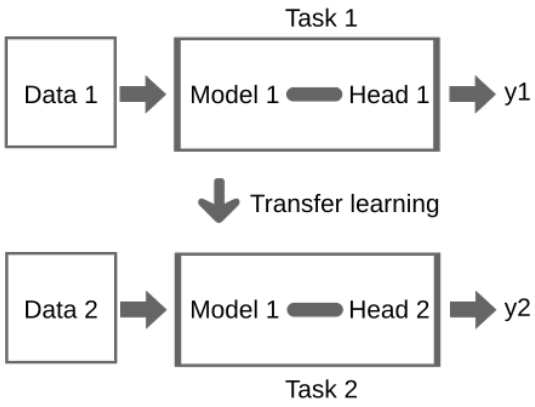


Figure 7.1: Transfer learning. Image from `https://en.wikipedia.org/wiki/Transfer_learning#/media/File:Transfer_learning.svg`

shared vocabularies and language tags. These systems already exhibited transfer effects, especially for low-resource languages, as we saw in Section 6.3.

Later work made pretraining an explicit design choice. Instead of learning only from parallel data, models were first trained on large-scale monolingual or multilingual corpora using self-supervised objectives, and only then fine-tuned for translation. This shift improved robustness, generalisation, and performance in low-resource and domain-adaptation settings.

7.5 What is pretrained in MT models?

Pretrained MT models are almost always based on the Transformer encoder–decoder architecture. Pretraining does not change the architecture itself, but rather how its parameters are initialised.

During pretraining, the model learns:

- lexical and subword representations;
- syntactic and semantic regularities;
- long-range dependencies and discourse patterns;
- cross-lingual correspondences (in multilingual settings).

As a result, fine-tuning can focus on task-specific alignment between source and target languages rather than rediscovering basic linguistic structure.

7.6 Pretraining objectives for MT

Unlike autoregressive language models, pretrained MT models typically use *sequence-to-sequence* pretraining objectives that resemble translation more closely.

7.6.1 Denoising autoencoding

A common objective is denoising autoencoding. The model is given a corrupted version of a sentence and trained to reconstruct the original sentence.

Typical noise functions include:

- token masking;
- token deletion;
- sentence permutation or span shuffling.

This objective forces the encoder to build meaningful representations of incomplete input and the decoder to generate fluent, coherent output, closely mirroring the translation process.

Denoising autoencoder example

Consider the following sentence:

```
She eats apples every day .
```

During pretraining, the model does not see the sentence in its original form. Instead, a noisy version is constructed by applying random corruptions.

Original sentence:

```
She eats apples every day .
```

Corrupted input (noise applied):

```
She <mask> apples day .
```

In this example, the token `eats` has been masked and the word `every` has been deleted.

The model is trained to reconstruct the original sentence from the corrupted input:

```
She eats apples every day .
```

The encoder must build a meaningful representation of the incomplete input, while the decoder learns to generate a fluent and coherent sentence that restores the missing information.

Although no translation is involved, this objective closely resembles machine translation: the model maps an imperfect input sequence to a well-formed output sequence.

This training setup is an instance of self-supervised learning. No external labels are required: the supervision signal is generated automatically by corrupting the input sentence and using the original sentence as the target. The model therefore learns from raw text alone, without human annotation, while still being trained in a supervised sequence-to-sequence setting. This combination of automatic target construction and explicit learning objectives makes self-supervision particularly well suited to large-scale pretraining.

7.6.2 Multilingual denoising

In multilingual pretraining, denoising is applied across multiple languages using a shared subword vocabulary. This encourages the model to learn language-agnostic representations and facilitates cross-lingual transfer.

Multilingual denoising autoencoder example

In multilingual denoising pretraining, the same objective is applied across multiple languages using a shared model and a shared subword vocabulary.

Consider the following parallel sentences:

English:

```
She eats apples every day .
```

Dutch:

```
Zij eet elke dag appels .
```

During pretraining, sentences from different languages are corrupted independently and used as inputs to the same model.

Corrupted English input:

```
She <mask> apples day .
```

Corrupted Dutch input:

```
Zij <mask> dag appels .
```

In both cases, tokens have been masked or deleted, but no translation signal is provided. The model is trained to reconstruct each sentence in its original language:

Target outputs:

```
She eats apples every day .
Zij eet elke dag appels .
```

By training on many languages in this way, the model learns shared representations across languages, enabling cross-lingual transfer when it is later fine-tuned for machine translation.

7.7 Representative pretrained MT models

7.7.1 MarianMT

MarianMT models are bilingual pretrained transformer models trained on large-scale parallel corpora. Each model is specialised for a single language pair and distributed via modern MT toolkits. MarianMT illustrates how pretraining on large parallel data can substantially improve performance over training from scratch.

7.7.2 mBART

mBART extends the BART denoising autoencoder to a multilingual setting. The model is pretrained on large monolingual corpora in many languages using a denoising objective, and then fine-tuned on parallel data for translation. mBART demonstrated strong gains in low-resource and domain-adaptation scenarios.

7.7.3 mT5

mT5 adapts the text-to-text framework of T5 to a massively multilingual corpus. All tasks, including translation, are framed as text generation problems. While not specific to MT, mT5 serves as a strong multilingual pretrained backbone for translation tasks.

7.7.4 NLLB

The NLLB (No Language Left Behind) models represent large-scale multilingual pretraining explicitly optimised for translation. They cover hundreds of languages, with a strong focus on low-resource settings, and combine large-scale data collection with careful training and evaluation strategies.

7.8 Fine-tuning pretrained models for MT

Fine-tuning adapts a pretrained model to a specific translation task. In most cases, the entire model is updated using parallel data, although partial freezing of layers is sometimes used for efficiency or stability.

Key advantages of fine-tuning include:

- faster convergence;
- improved performance with limited parallel data;
- better domain adaptation;
- increased robustness and fluency.

Fine-tuning can be applied to bilingual, multilingual, or domain-specific datasets, depending on the target application.

7.9 Comparison with training from scratch

Compared to training from scratch, pretrained MT models:

- require significantly less parallel data;
- achieve higher quality, especially for low-resource languages;
- are less sensitive to hyperparameter choices;
- better preserve fluency and adequacy under domain shift.

The main cost of pretraining lies in its computational expense, which is borne once and amortised across many downstream tasks.

7.10 Relation to large language models

Pretrained MT models anticipate many ideas later popularised by large language models (LLMs). Both rely on large-scale pretraining, shared representations, and task adaptation via fine-tuning or prompting.

A key difference is that pretrained MT models remain explicitly sequence-to-sequence and translation-focused, whereas LLMs are trained primarily as general-purpose language models. Understanding pretrained MT models therefore provides a natural conceptual bridge between classical NMT and modern LLM-based translation approaches.

7.11 Summary

Pretrained models have become a central paradigm in machine translation. By decoupling general language learning from task-specific adaptation, they enable efficient, robust, and high-quality translation across languages and domains. In the next chapter, we build on this foundation by examining multilingual MT models in more detail and analysing how cross-lingual transfer emerges in large pretrained systems.

Chapter 8

Decoder Only Models

8.1 Machine Translation with Decoder-Only Language Models

8.1.1 A paradigm shift

Recent advances in machine translation have been driven not only by new data and scale, but also by a fundamental architectural shift. Whereas classical neural MT relies on encoder–decoder models explicitly trained to map a source sentence to a target sentence, large language models (LLMs) are typically *decoder-only* architectures trained with a generic next-token prediction objective.

In this paradigm, translation is no longer the primary training task. Instead, it emerges as a *capability* of a general-purpose language model that has been exposed to large amounts of multilingual text.

8.1.2 Decoder-only architecture

Decoder-only models consist of a stack of Transformer decoder blocks trained autoregressively. At each step, the model predicts the next token given all previous tokens in the input sequence.

Unlike encoder–decoder MT models:

- there is no separate encoder for the source sentence;
- source and target text are concatenated into a single input sequence;
- translation is framed as conditional language generation via prompting.

This architectural choice has important consequences for how translation is trained, controlled, and evaluated.

8.1.3 Translation as prompting

In decoder-only models, translation is typically performed through *prompts* that specify the task and provide the source sentence. For example:

```
Translate the following sentence from English to Dutch: She eats apples.
```

The model then generates the translation autoregressively. No architectural changes are required; translation is treated as one task among many.

Prompt-based translation offers great flexibility, but also reduces explicit control over adequacy, terminology, and consistency compared to classical MT systems.

8.1.4 Pretraining objectives

Decoder-only LLMs are pretrained using a causal language modeling objective: predicting the next token in large-scale text corpora. These corpora are predominantly monolingual, although multilingual data is often included.

As a result, translation ability arises indirectly from:

- exposure to parallel or comparable texts;
- multilingual co-occurrence patterns;
- instruction-like text present in the data.

This contrasts sharply with encoder–decoder MT models, where parallel data and translation-specific objectives play a central role from the outset.

8.1.5 Instruction tuning

Instruction tuning adapts a pretrained language model to follow natural-language instructions. The model is fine-tuned on datasets consisting of *instruction–response* pairs, covering a wide range of tasks, including translation.

For MT, instruction tuning has two main effects:

- it stabilises prompt-based translation behaviour;
- it improves adherence to task descriptions and output format.

However, instruction tuning does not fundamentally change the model's objective: translation remains one behaviour among many rather than a specialised task.

8.1.6 Reinforcement learning from human feedback

Reinforcement learning from human feedback (RLHF) further adapts decoder-only models by optimising them toward preferred outputs. Human or automated preferences are used to train a reward model, which then guides policy optimisation.

In the context of translation, RLHF tends to prioritise:

- fluency and naturalness;
- politeness and coherence;
- general acceptability of the output.

While these properties are valuable, they do not necessarily align with traditional MT criteria such as adequacy, completeness, or terminological consistency. This mismatch has important implications for professional translation use.

8.1.7 Strengths and limitations of LLM-based MT

Decoder-only models offer several advantages for translation:

- strong fluency and grammaticality;
- flexibility across languages and domains;
- zero-shot and few-shot translation capabilities.

At the same time, they exhibit clear limitations:

- weaker guarantees of adequacy;
- sensitivity to prompting and wording;
- inconsistent terminology and style;
- limited controllability compared to classical MT systems.

These trade-offs highlight that LLM-based translation optimises a different notion of quality than traditional MT systems.

8.1.8 MADLAD: translation-specialised decoder-only models

MADLAD (Massively Multilingual Adaptation of Large Language Models for Translation) represents an important counterpoint to general-purpose LLMs. MADLAD models are decoder-only and large-scale, but are trained explicitly for translation using massive amounts of parallel data.

Key characteristics of MADLAD include:

- hundreds of languages with balanced coverage;
- translation-centric training objectives;
- strong performance in low-resource settings;
- competitive or superior results compared to prompted general-purpose LLMs on MT benchmarks.

MADLAD demonstrates that architectural scale alone is insufficient: translation quality depends critically on training data and objectives.

8.1.9 Comparison with encoder–decoder MT

The contrast between encoder–decoder MT models and decoder-only LLMs can be summarised as follows:

- encoder–decoder models offer stronger task-specific control and adequacy;
- decoder-only models offer greater flexibility and generalisation;
- translation-specific training remains crucial for professional-quality MT.

Rather than replacing classical MT, LLM-based approaches introduce a complementary paradigm with different strengths and weaknesses.

8.1.10 Implications for translation practice

For translators and translation technologists, decoder-only models raise practical and ethical questions:

- how to evaluate translation quality beyond surface fluency;
- how to ensure faithfulness and completeness;
- how to integrate LLM-based MT into professional workflows.

Understanding the training objectives and limitations of these models is essential for responsible use.

8.1.11 Summary

Decoder-only language models represent a distinct paradigm for machine translation. Through prompting, instruction tuning, and reinforcement learning, they achieve impressive fluency and flexibility, but often diverge from traditional MT optimisation goals. Models such as MADLAD show that large-scale decoder-only architectures can be successfully adapted for translation when trained with MT-specific objectives. Together with encoder–decoder and multilingual models, they complete the modern landscape of machine translation approaches.

Bibliography

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2015). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *ICLR*.

Bengio, Yoshua et al. (2003). "A Neural Probabilistic Language Model". In: *Journal of Machine Learning Research* 3.

Brown, Peter F. et al. (1993). "The Mathematics of Statistical Machine Translation: Parameter Estimation". In: *Computational Linguistics* 19.2.

Brown, Tom, Benjamin Mann, Nick Ryder, and et al. (2020). "Language Models are Few-Shot Learners". In: *NeurIPS*.

Callison-Burch, Chris, Cameron Fordyce, Philipp Koehn, Christof Monz, and Josh Schroeder (2007). "Meta-evaluation of machine translation". In: *Proceedings of the Second Workshop on Statistical Machine Translation*. ACL, pp. 136–158.

Cho, Kyunghyun, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio (2014). "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, pp. 1724–1734. DOI: 10.3115/v1/D14-1179.

Chowdhery, Aakanksha, Sharan Narang, Jacob Devlin, and et al. (2022). "PaLM: Scaling Language Modeling with Pathways". In: *arXiv preprint arXiv:2204.02311*.

Costa-jussà, Marta R., James Cross, Onur Celebi Wang, Vishrav Chaudhary, Angela Fan, Cynthia Gao, Xian Li Garcia, Francisco Guzmán, Philipp Koehn, Andrei Mourachko, and et al. (2022). "No Language Left Behind: Scaling Human-Centered Machine Translation". In: *arXiv preprint arXiv:2207.04672*.

CrossLang (2025). *Human Evaluation of Machine Translation Quality – A Quick Guide/*.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *NAACL*.

European Association for Machine Translation (2024). *What is Machine Translation?* URL: https://eamt.org/what-is-machine-translation/.

Fan, Angela, Shruti Bhosale, Holger Schwenk, and et al. (2021). "Beyond English-Centric Multilingual Machine Translation". In: *JMLR*.

Graham, Yvette, Timothy Baldwin, Alistair Moffat, and Justin Zobel (Aug. 2013). "Continuous Measurement Scales in Human Evaluation of Machine Translation". In: *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*. Ed. by Antonio Pareja-Lora, Maria Liakata, and Stefanie Dipper. Sofia, Bulgaria: Association for Computational Linguistics, pp. 33–41. URL: https://aclanthology.org/W13-2305/.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9(8), pp. 1735–1780.

Hutchins, W. John (Sept. 2004). "The Georgetown-IBM experiment demonstrated in January 1954". In: *Proceedings of the 6th Conference of the Association for Machine Translation in the Americas: Technical Papers*. Ed. by Robert E. Frederking and Kathryn B. Taylor. Washington, USA: Springer, pp. 102–114. URL: https://aclanthology.org/2004.amta-papers.12/.

Intento (2025). *State of Machine Translation 2025*.

Jurafsky, Daniel and James H. Martin (2025). *Speech and Language Processing*. 3rd ed. Draft of August 24, 2025. Prentice Hall.

Koehn, Philip (2025). *Machine Translation: Evaluation*. URL: `http://mt-class.org/jhu/slides/lecture-evaluation.pdf`.

Koehn, Philipp (2004). "Statistical Significance Tests for Machine Translation Evaluation". In: *Proceedings of EMNLP*, pp. 388–395.

– (2020). *Neural Machine Translation*. Cambridge University Press.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst (June 2007). "Moses: Open Source Toolkit for Statistical Machine Translation". In: *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*. Ed. by Sophia Ananiadou. Prague, Czech Republic: Association for Computational Linguistics, pp. 177–180. URL: `https://aclanthology.org/P07-2045/`.

Kudo, Taku and John Richardson (2018). "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Brussels, Belgium: Association for Computational Linguistics, pp. 66–71.

Liu, Yinhan, Jiatao Gu, Naman Goyal, and et al. (2020). "Multilingual Denoising Pre-training for Neural Machine Translation". In: *TACL*.

Lommel, Arle, Aljoscha Burchardt, and Hans Uszkoreit (2014). "The MQM Framework: A Comprehensive Framework for Translation Quality Assessment". In: *Proceedings of LREC*.

Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning (2015). "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of EMNLP*.

Nagao, Makoto (1984). "A framework of a mechanical translation between Japanese and English by analogy principle". In: *Proc. of the International NATO Symposium on Artificial and Human Intelligence*. Lyon, France: Elsevier North-Holland, Inc., pp. 173–180. ISBN: 0444865454.

Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu (2002). "BLEU: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of ACL*, pp. 311–318.

Peters, Matthew, Mark Neumann, Mohit Iyyer, and et al. (2018). "Deep Contextualized Word Representations". In: *NAACL*.

Pierce, John R., John B. Carroll, Eric P. Hamp, David G. Hays, Charles F. Hockett, Anthony G. Oettinger, and Alan Perlis (1966). *Languages and Machines: Computers in Translation and Linguistics*. Tech. rep. Automatic Language Processing Advisory Committee.

Plitt, Mirko and François Masselot (2010). "A productivity test of statistical machine translation post-editing in a typical localization context". In: *Prague Bulletin of Mathematical Linguistics* 93, pp. 7–16.

Popovic, Maja (2015). "chrF: character n-gram F-score for automatic MT evaluation". In: *Proceedings of the Tenth Workshop on Statistical Machine Translation*. ACL, pp. 392–395.

Post, Matt (2018). "A call for clarity in reporting BLEU scores". In: *Proceedings of WMT*, pp. 186–191.

Prates, Marcelo O.R., Pedro H. Avelar, and Luis C. Lamb (2019). "Assessing gender bias in machine translation: A case study with Google Translate". In: *Neural Computing and Applications* 32, pp. 6363–6381.

Radford, Alec, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever (2018). "Improving Language Understanding by Generative Pre-Training". In: *OpenAI Technical Report*.

Raffel, Colin, Noam Shazeer, Adam Roberts, and et al. (2020). "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *JMLR*.

Rei, Ricardo, Craig Stewart, Ana Farinha, and Alon Lavie (2020). "COMET: A Neural Framework for MT Evaluation". In: *Proceedings of EMNLP*, pp. 2685–2702.

Schuster, Mike and Kuldip K. Paliwal (1997). "Bidirectional Recurrent Neural Networks". In: *IEEE Transactions on Signal Processing* 45.11, pp. 2673–2681.

Sellam, Thibault, Dipanjan Das, and Ankur P Parikh (2020). "BLEURT: Learning Robust Metrics for Text Generation". In: *Proceedings of ACL*, pp. 7881–7892.

Sennrich, Rico (2018). "Machine Translation: Lecture Notes". In: Available online.

Snover, Matthew, Bonnie Dorr, Richard Schwartz, Linnea Micciulla, and John Makhoul (2006). "A Study of Translation Edit Rate with Targeted Human Annotation". In: *Proceedings of AMTA*, pp. 223–231.

Sutskever, Ilya et al. (2014). "Sequence to Sequence Learning with Neural Networks". In: *NIPS*.

Tiedemann, Jörg (2012). "Parallel Data, Tools and Interfaces in OPUS". In: *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, pp. 2214–2218. URL: https://aclanthology.org/L12-1246.

Vanroy, Bram, Arda Tezcan, and Lieve Macken (2023). "MATEO: A Machine Translation Evaluation Online Platform". In: *Proceedings of the 24th Annual Conference of the European Association for Machine Translation*. Tampere, Finland: European Association for Machine Translation, pp. 389–393. URL: https://aclanthology.org/2023.eamt-1.52.

Vaswani, Ashish et al. (2017). "Attention Is All You Need". In: *NIPS*.

Vauquois, Bernard (1968). "Structures profondes et traduction automatique. Le système du C.E.T.A." In: *Revue roumaine de linguistique* XIII (2), p. 105. ISSN: 0035-3957.

Vermeer, Hans J. (1989). "Skopos and Commission in Translational Action". In: *Readings in Translation Theory*. Ed. by Andrew Chesterman. Helsinki: Oy Finn Lectura Ab, pp. 173–187.

Weaver, Warren (1949). *Translation*. Memorandum, reprinted in Locke and Booth (1955).

Xue, Linting, Noah Constant, Adam Roberts, and et al. (2021). "mT5: A Massively Multilingual Pre-trained Text-to-Text Transformer". In: *NAACL*.

Zhang, Tianyi, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi (2020). "BERTScore: Evaluating Text Generation with BERT". In: *Proceedings of ICLR*.