Neural Machine Translation with RNNs

Machine Translation: Advanced Topics

Vincent Vandeghinste

KU Leuven – Brussels

2026

Outline

Neural MT with RNNs

From Language Modeling to Translation

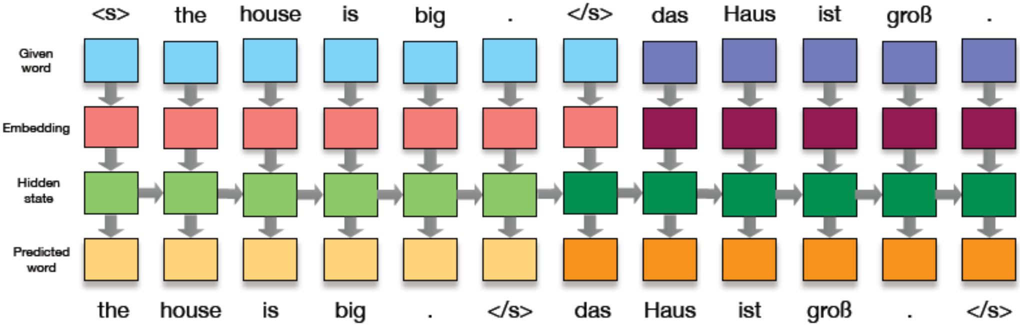
- In an RNN language model (RNN-LM), the model predicts the **next word** given previous words.
- Neural MT with RNNs introduces one essential extension: **conditional generation**.
- We split the model into two parts:
  - **Encoder RNN:** reads the *source* sentence and builds a representation.
  - **Decoder RNN:** generates the *target* sentence, conditioned on the encoder.
- Intuition: translation = language modeling *with source context*.

Seq2Seq Encoder–Decoder (Classic RNN NMT)

- **Encoder:** processes $x_1, \ldots, x_T$ and outputs a final state h_T.
- **Decoder:** predicts $y_1, \ldots, y_{T'}$ step-by-step.
- Key connection: decoder starts from the encoder summary:

$$s_0 = h_T$$

- This creates a **fixed-size bottleneck**: all source information must fit in one vector.



Toy Example

Toy Parallel Corpus (EN $\rightarrow$ NL)

We illustrate the preprocessing pipeline on a tiny parallel corpus:

Source (EN)

- i am a student
- i am a teacher
- you are a student
- . . .
- she likes apples

Target (NL)

- ik ben een student
- ik ben een leraar
- jij bent een student
- . . .
- zij lust appels

Goal

Convert text into fixed-size integer tensors so an encoder–decoder can be trained.

Step 1: Add Sentence Boundary Tokens

We add explicit boundary markers on **both** sides:

Example with markers

EN: `<sos> i am a student <eos>`
NL: `<sos> ik ben een student <eos>`

- `<sos>` tells the decoder how to start generation.
- `<eos>` tells the decoder when to stop.
- These play the same role as in language modeling, but now for **two languages**.

Step 2: Build Two Vocabularies

We build **separate** word→ID mappings:

EN word→ID (excerpt)

<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

NL word→ID (illustrative excerpt)

<sos>→1
<eos>→2
een→3
ik→6
ben→11
student→7
jij→8
zij→9
lust→10
appels→12

Step 3: Convert Sentences to Integer Sequences

Apply the mapping to each token:

Example: EN $\rightarrow$ IDs

EN: `<sos> i am a student <eos>`
$\Rightarrow$ `[1, 6, 10, 3, 7, 2]`

Example: NL $\rightarrow$ IDs

NL: `<sos> ik ben een student <eos>`
$\Rightarrow$ `[1, 6, 11, 3, 7, 2]`

- After this step, each sentence is a variable-length list of integers.
- Neural models train in batches, so we need a fixed length next.

Step 4: Pad to a Fixed Length

We pad with zeros on the right (post-padding):

Padding idea

If `max_len = 6` and a sequence has length 4:
`[1, 9, 4, 2]` $\Rightarrow$ `[1, 9, 4, 2, 0, 0]`

- Padding enables rectangular tensors: **batch** $\times$ **time**.
- With masking (`mask_zero=True`), the model ignores padding positions.

Vocabulary Size and Padding Index

Two practical numbers matter for model shapes:

- **Vocabulary size** (per language): number of known tokens.
- We add 1 so that **ID 0** is reserved for padding.

From the toy example

`num_src_tokens = 20` (including padding ID 0)
`num_tgt_tokens = 19` (including padding ID 0)
`max_src_len = 6`, `max_tgt_len = 6`

- Embedding tables have one row per token ID.
- Max lengths define how many time steps encoder/decoder process.

Why We Shift the Target for Training

The decoder is trained like a language model: predict the next token.

Teacher forcing (concept)

At time t, the decoder receives the **correct previous token** y_{t-1} and must predict y_t.

This is implemented by creating two shifted sequences:

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

Teacher Forcing (Training the Decoder)

- During training, we know the full reference translation.
- The decoder is a conditional LM: at position t it predicts y_t given prior target words.
- **Teacher forcing:** feed the *gold* previous token as input (stabilises training).
- Conceptually:

$$p(y_t \mid y_{<t}, x) \quad \text{is trained with } y_{t-1} \text{ from the reference.}$$

- During inference, teacher forcing is impossible: the model must use its **own** previous predictions.

Step 5: Decoder Input vs Decoder Target (Shift)

Example target sentence (already with markers):

Target sentence

`<sos> ik ben een student <eos>`
IDs: `[1, 6, 11, 3, 7, 2]`

Decoder input

Drop the last token:
`[1, 6, 11, 3, 7]`

Decoder target

Drop the first token:
`[6, 11, 3, 7, 2]`

- The model learns: given `<sos> ik ben een student` predict `ik ben een student <eos>`.
- Exactly like next-word prediction, but conditioned on the source sentence.

Step 6: What the Model Sees During Training

For each training example:

- **Encoder input:** padded EN ID sequence (full source sentence)
- **Decoder input:** shifted NL IDs (teacher forcing input)
- **Supervision:** decoder targets (next-token labels)

Summary (one sentence pair)

EN encoder input: `[1, 6, 10, 3, 7, 2]`
NL decoder input: `[1, 6, 11, 3, 7]`
NL decoder target: `[6, 11, 3, 7, 2]`

- This creates a clean supervised signal at every target position.
- During inference, we feed back the model's own predictions instead.

Inference: Autoregressive Decoding

- Encode the source sentence once $\Rightarrow$ initial decoder state.
- Generate tokens iteratively:
  1. start with <sos>
  2. predict next token distribution
  3. choose a token (often greedy or beam search)
  4. stop when <eos> is generated
- This mismatch between training (teacher forcing) and inference (self-feeding) is one reason why early seq2seq RNNs can be fragile.

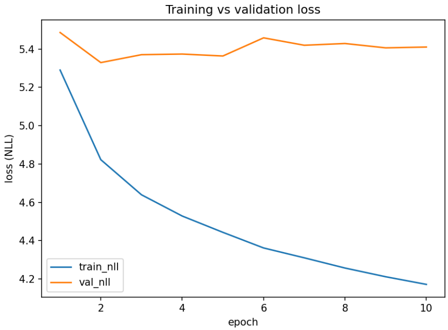
Baseline Experiment (Vanilla RNN)

Baseline Setup: What We Keep Fixed

- Controlled comparison: same data preparation and fixed dev/test split.
- Goal: establish a **reference point** before architectural improvements.
- Evaluation on development set during training (diagnostics):
  - **Loss** (cross-entropy / NLL)
  - **BLEU** and **chrF** (overlap metrics)
- Final test evaluation is performed once a reasonable dev configuration is chosen.

Baseline Result: Overfitting in Loss

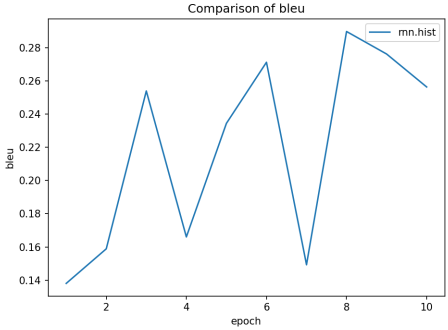
- Training loss decreases steadily $\Rightarrow$ model can fit training data.
- Validation loss does *not* improve similarly $\Rightarrow$ poor generalisation.
- This is a classic **overfitting** pattern on limited data / capacity mismatch.



Training vs validation loss for plain RNN.

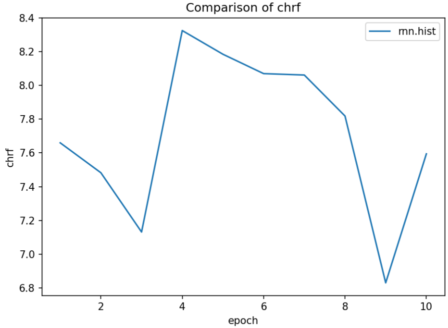
Baseline Result: BLEU Stays Very Low

- BLEU remains extremely low (roughly **0.14–0.28** on the development set).
- No consistent upward trend across epochs.
- Interpretation:
  - the model reduces training loss,
  - but this does not translate into improved n-gram overlap with references.
- Conclusion: the vanilla RNN baseline fails to learn useful translation correspondences.



Baseline Result: chrF Also Shows No Improvement

- chrF follows a similar pattern: no clear improvement over epochs.
- Since chrF operates on character n-grams:
  - it is more tolerant to morphology,
  - but still detects poor translation quality.
- Both BLEU and chrF confirm:
  - decreasing training loss does *not* imply better translation quality.



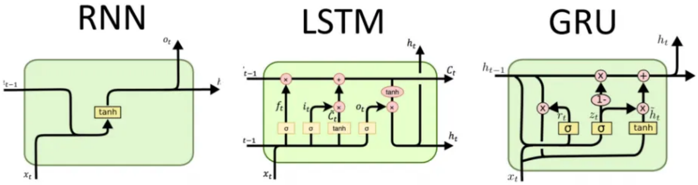
Gated RNNs: GRU and LSTM

Why Gating? Vanishing Gradients and Long Context

- Plain RNNs struggle to preserve information over many time steps:
  - gradients shrink during backpropagation through time,
  - early words lose influence on later predictions.
- In MT this harms:
  - adequacy (missing content),
  - agreement phenomena,
  - long-distance reordering.
- **Gates** introduce learnable control over what to keep vs forget.

GRU: Intuition

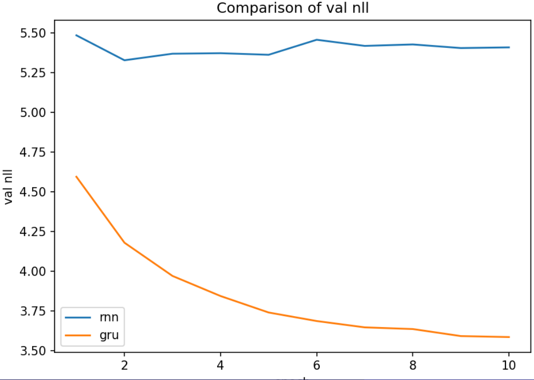
- GRU (Cho et al. (2014)) uses two gates:
  - **Update gate:** how much previous state to carry forward.
  - **Reset gate:** how much past information to ignore when computing new content.
- Effect: better modeling of dependencies without the full complexity of LSTM.
- In practice: often a strong, efficient default for recurrent MT.

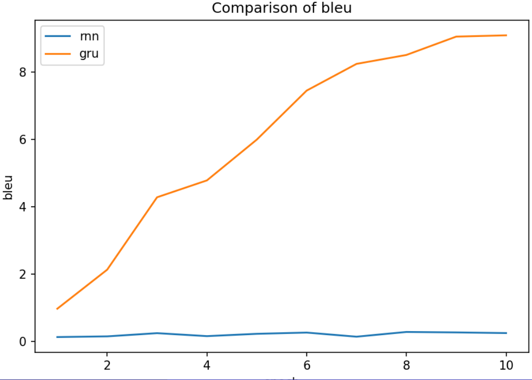


Comparing RNN, GRU, LSTM cell structure (illustration).

GRU: Results (Learning Curves)

- Validation loss improves in early epochs, then converges ⇒ still overfitting, but **learns something**.
- Compared with plain RNN:
  - **lower** validation loss,
  - **higher** BLEU on dev.



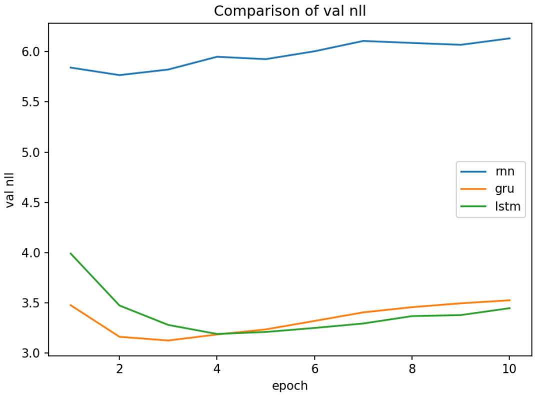


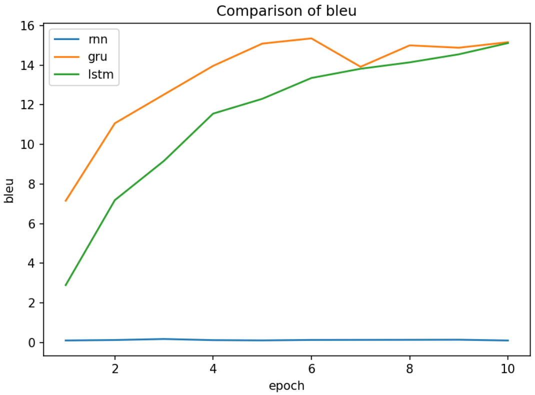
LSTM: Intuition

- LSTM (Hochreiter and Schmidhuber (1997)) separates:
  - **Cell state** (long-term memory),
  - **Hidden state** (short-term working state).
- Three gates:
  - **Forget gate** (discard),
  - **Input gate** (write),
  - **Output gate** (expose).
- Often more stable on longer sequences; sometimes slightly heavier than GRU.

RNN vs GRU vs LSTM: Results Summary

- Plain RNN performs worst on dev: little to no BLEU growth.
- GRU and LSTM both improve dev loss and BLEU substantially.
- Difference GRU vs LSTM is modest here, plausibly because sentences are short.





Validation loss (left) and BLEU (right) across recurrent cell types.

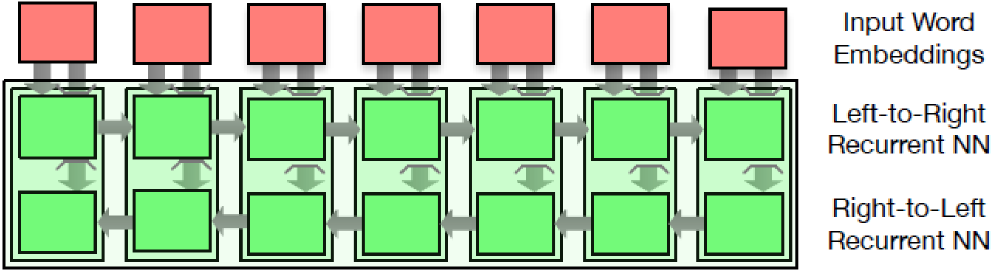
Bidirectional Encoders

Motivation: Why Bidirectional?

- Unidirectional encoder reads left-to-right: each hidden state only sees the *past*.
- For translation, important cues may appear late (verbs, negation, disambiguators).
- Bidirectional encoder processes the source sentence:
  - forward ($1 \rightarrow T$) and
  - backward ($T \rightarrow 1$),

  and combines both representations.
- Decoder remains left-to-right (generation constraint).

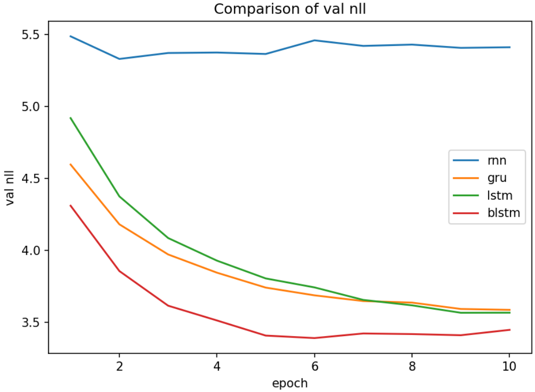
Bidirectional Encoder: Concept

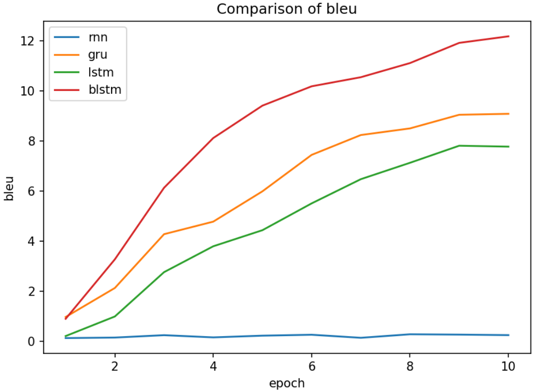


- Each source position gets a representation with **left and right context**.
- Typically used only in the encoder; decoder stays causal.

Bidirectional LSTM: Results

- Compared to (uni) LSTM:
  - lower validation loss,
  - higher BLEU on dev.
- Still exhibits overfitting: validation improves early and then saturates.





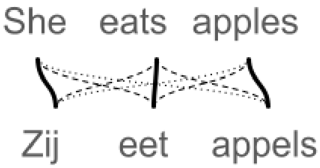
Cross-lingual attention

Why Attention? The Fixed-Vector Bottleneck

- Without attention, the decoder depends on a single encoder summary vector.
- This creates an **information bottleneck**, especially for:
  - longer sentences,
  - complex reordering,
  - multiword correspondences.
- Attention (Bahdanau et al., 2015; Luong et al., 2015) lets the decoder **consult the source** at every decoding step.

Attention as Soft Alignment

- Without attention, the decoder only receives a single summary of the source sentence.
- With attention, the decoder can **look back at the source sentence** at every step.
- When predicting a target word, the model:
  - assigns a relevance score to each source word,
  - focuses more strongly on the most relevant parts,
  - combines this information to predict the next word.
- This creates a **soft alignment** between source and target words.



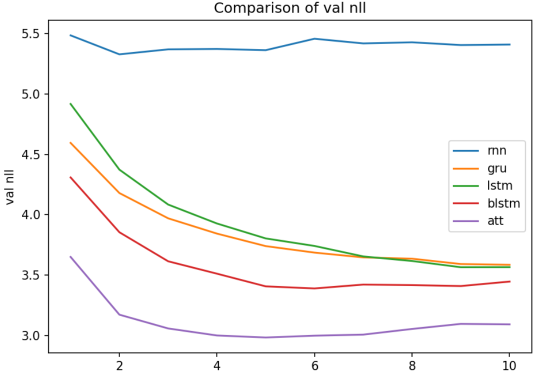
Darker lines = stronger focus.

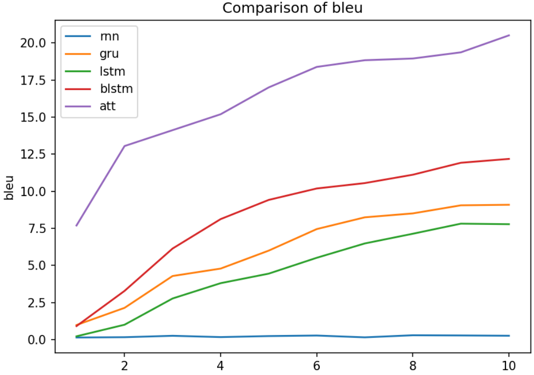
Attention: Practical Consequences

- Better adequacy: reduced omissions (decoder can revisit relevant source parts).
- Better word order: supports reordering and long-distance dependencies.
- Often the biggest jump in quality among recurrent NMT upgrades.
- Attention weights are interpretable and can be visualised as alignments.

Bidirectional LSTM + Attention: Results

- Adding attention to the bidirectional LSTM yields a **substantial improvement**.
- Improvements visible in:
  - lower dev loss,
  - higher BLEU and chrF.
- Overfitting pattern remains: dev improvements saturate after few epochs.





Model Comparison: What Helped Most?

- **Plain RNN** → weak baseline, little dev BLEU progress.
- **Gating (GRU/LSTM)** → clear improvement (better memory).
- **Bidirectional encoder** → further gains (richer source encoding).
- **Attention** → strongest improvement (removes bottleneck).

Take-home message

Most quality gains in recurrent NMT come from **better handling of context**: gates help preserve it, bidirectionality enriches it, and attention makes it *directly accessible* during decoding.

Limitations of Recurrent NMT (Even with Attention)

- Sequential computation limits training/inference speed (hard to parallelise).
- Long-range dependencies remain challenging compared to fully attention-based models.
- Motivates the next step in MT history: **Transformers**.

Hands-on RNNs

For the hands-on sessions, we first start in Google Colab, and for larger size models we switch to Kaggle.

- Colab Link
- Kaggle Link