

Lecture 3: Sequence Modeling, Translation, and Attention

A language model predicts what comes next. A translator must also carry what a source sentence says into a target sentence, which may have a different length and a different word order. This lecture follows how neural networks learned to translate between 2013 and 2017, and how two problems they met on the way led to the Transformer.

Access to the source. In the first neural translators, one network read the source sentence and compressed it into a single state of fixed size (**encoding**), and a second network wrote the translation from that state, one word at a time (**decoding**). Every word of the translation is then predicted from the same summary, and a fixed number of numbers has to hold the whole sentence, however long it is. The longer the sentence, the less room for each word, and some early translators built this way lost accuracy as sentences grew longer. (Trained to write strings of letters backwards, a task that needs only memory, a network of this kind fails on long strings, §5.) **Attention** removes this limit. The encoder keeps one state for every source word, and before writing each word the decoder reads a weighted average of these states, with weights computed by comparing its own state, which summarizes what it has written so far, with each source state. So for each word it writes, the decoder can draw mainly on the source words that matter for that word.

Computing one position at a time. A recurrent network computes its state at each position from its state at the position before. So even in training, when the whole target sentence is known in advance, the prediction of the t -th target word needs the state computed for the prediction of word $t-1$, which needs the state for word $t-2$, and so on. The predictions are computed one after another, in a chain as long as the sentence, and parallelism cannot help within a sentence. Convolutional networks removed the chain first, by computing each position from a small window of neighboring positions in the layer below. **Self-attention** removes the chain as well: each position reads directly all the positions it may see (in a decoder, only those up to its own), and no position waits for another, so in training all positions of a layer are computed at the same time. Generating a translation still goes one word at a time, for every model, since each word must be chosen before the next can be predicted.

The lecture has four parts:

1. sequence tasks and translation (§§1–2): what sequence models do; translation as conditional language modeling, with its training, decoding, and evaluation;
2. recurrent encoder–decoders (§§3–5): the encoder and decoder roles, the sequence-to-sequence translator, and its drawbacks;
3. attention (§§6–7): its inception and its variants;
4. beyond recurrence (§§8–10): convolutional sequence models, translation from attention alone, and self-attention and the Transformer.

Parts 2 and 3 are the core. Throughout the notes, the translation examples are English image captions translated into French by small models trained for these notes, and one caption returns in all four parts.

1. From language modeling to sequence modeling

From n-gram counts to learned word vectors, recurrent networks, and LSTMs, the language models of [Lecture 2](#) changed how they computed, not what they computed: a probability for every sequence of tokens, built from predictions of the next token. Language is one source of sequences among many, and predicting the next element is one of several things a model can do with a sequence.

A **sequence** is an ordered list $x_{1:T} = (x_1, \dots, x_T)$ whose length varies from one example to the next. Order carries information: *dog bites man* and *man bites dog* contain the same words. And since the length is not fixed in advance, a sequence model cannot give each position its own parameters, the way a network with a fixed number of inputs gives each input its own weights. Instead it applies one computation, with the same parameters, at every position, as the recurrent networks of [Lecture 2 \(Section 7\)](#) apply one cell at every step; this is what lets it handle sequences of any length. The elements need not be words. A thermometer read every hour gives a sequence of numbers; an accelerometer worn on the wrist gives three numbers at every time step; a weather model's state at each hour is a whole field of temperature, pressure, and wind on a grid, so that each prediction is a map, not a single number (**spatiotemporal** forecasting).

Sequence tasks

Sequence tasks differ in what is given and what is produced. [Figure 1](#) draws one example of each task in the table below.

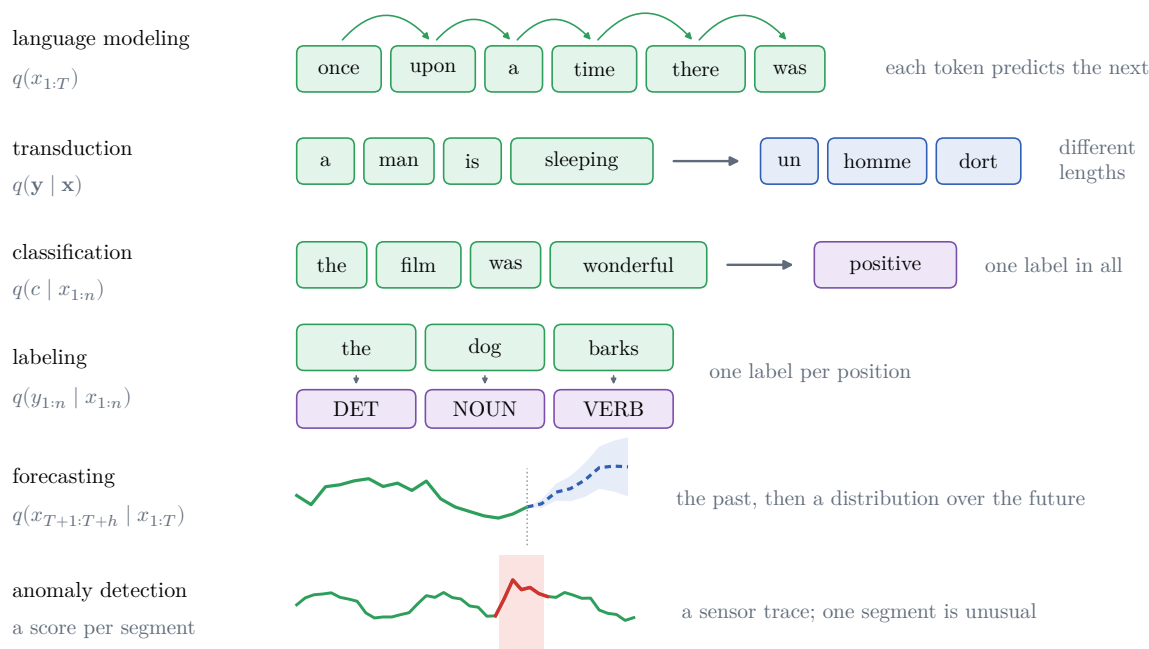


Figure 1. One example of each sequence task. In transduction, the output sequence may differ in length and order from the input. The shaded band is a forecast's uncertainty; the red segment is an anomaly.

Task	Given and produced	Examples
Language modeling	A distribution over sequences, $q(x_{1:T})$, usually factored into next-token predictions	Predicting or generating text
Sequence transduction	An input sequence, then an output sequence, possibly of different length and order, $q(\mathbf{y} \mid \mathbf{x})$	Translation, speech recognition, summarization
Sequence classification	A sequence, then one label, $q(c \mid x_{1:n})$	The sentiment of a review; the activity in an accelerometer recording
Sequence labeling	A label for every position, $q(y_{1:n} \mid x_{1:n})$	Parts of speech; names of people and places; events in a sensor stream
Forecasting	The past of a series, then its future, $q(x_{T+1:T+h} \mid x_{1:T})$	Temperature, electricity demand, traffic, weather fields
Anomaly detection	Observations, then a score of how unusual each one or each segment is	Monitoring machines; abnormal sensor readings

The categories overlap: they name what is wanted from a model, and they are not disjoint classes. Labeling, which keeps the length and the order, can be read as a transduction, and so can forecasting. A language model continuing a prompt also does a transduction, since it produces an output conditioned on an input. And a forecasting model can detect anomalies, by flagging the observations it gave low probability.

Current practice has blurred the lines further for language. Many language tasks can be written as a prompt followed by an output and handed to a single language model that continues the prompt: a review followed by *Sentiment:*, or an English sentence followed by *French:*. Lectures 5 and 7 show how models are trained to do this. Not every sequence task has become text generation: a forecast of a weather field or of a machine’s vibrations is not naturally written in words, and such forecasts are made by models built for numerical data.

Translation

From here on the notes concentrate on a special case of transduction: translation. It contains, in one task, the main difficulties of producing one sequence from another (lengths differ, order changes, words correspond to phrases or to nothing), and the mechanism developed for it, attention, became the central operation of the Transformer. The data of translation are a **parallel corpus**: a collection of sentences paired with their translations. These notes use Multi30k (Elliott et al., 2016; 2017), a small English–French corpus of image captions. Figure 2 shows one of its pairs.

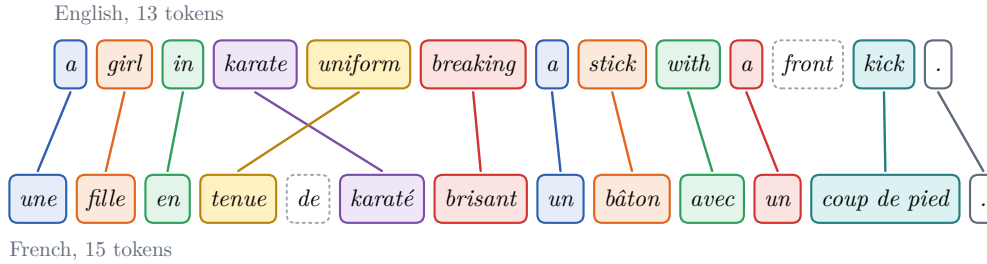


Figure 2. Words correspond to words, to phrases, or to nothing, and their order changes. A Multi30k caption and its reference translation; words on boxes of the same color, joined by a line, correspond (colors repeat; the lines give the pairs). *Karate uniform* becomes *tenue de karaté*, with the noun first (the crossing lines); *kick* becomes three words, *coup de pied*; and *front* and *de* (dashed boxes) have no counterpart.

Thirteen English tokens, counting the full stop, become fifteen French ones. The correspondence in the figure, called the **alignment** of the pair, was drawn by hand: nothing in a parallel corpus says which French word comes from which English word, and a translator has to learn such correspondences from the pairs alone.

Learning to translate from parallel corpora is older than neural networks. From about 1990, statistical systems treated the alignment as a hidden variable, estimated from the corpus, and from 2003 the best of them, **phrase-based** systems, translated a sentence piece by piece, in units of a few words. The open-source Moses toolkit (Koehn et al., 2007) made them the baseline that every neural translator of 2014 was compared with. [Appendix B.1](#) tells this history briefly.

2. Translation as conditional sequence prediction

Every translation model of this lecture shares four things, whatever its architecture: the probability model, its training, the search that turns it into a translation, and the measure of quality. What changes from §3 on is only the **architecture**, the computation that turns the source and the target prefix (the target tokens before the current one) into the distribution of the next token.

The model and its training

Notation. The data are pairs of a **source** sentence $\mathbf{x} = x_{1:n}$ of n tokens and its **target**, a translation $\mathbf{y} = y_{1:m}$ of m tokens made by a person, also called the **reference**, from a parallel corpus. The target is followed by an end-of-sequence token $y_{m+1} = \text{EOS}$, so that the model also decides where the translation stops, and preceded by a beginning-of-sequence token $y_0 = \text{BOS}$. Source positions are indexed by j , target positions by t . $E(x)$ is the vector of token x , from a table that [Lecture 2 \(Section 6\)](#) called C . Vectors are rows and a linear map is $\mathbf{x}W$; [Lecture 2](#) wrote columns and $W\mathbf{x}$, so its formulas carry over with every matrix transposed.

The model is the chain rule of [Lecture 2 \(Section 1\)](#) with the source as extra context:

$$q_{\theta}(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{m+1} q_{\theta}(y_t \mid y_{<t}, \mathbf{x}),$$

a **conditional language model**: each target token is predicted from the whole source and the target tokens before it. It is trained by maximum likelihood, that is, by minimizing the

negative log-likelihood: the log loss $-\log q$ of [Lecture 2 \(Section 3\)](#), summed over the $m_i + 1$ target tokens of each pair i (EOS included) and averaged over the P training pairs:

$$\mathcal{L}(\theta) = -\frac{1}{P} \sum_{i=1}^P \sum_{t=1}^{m_i+1} \log q_\theta(y_t^{(i)} | y_{<t}^{(i)}, \mathbf{x}^{(i)}), \quad \hat{\theta} = \arg \min_{\theta} \mathcal{L}(\theta).$$

Perplexity is the exponential of the loss per target token, as in [Lecture 2](#).

Teacher forcing. In training, the **decoder** (the part of the model that writes the target, §3) reads the previous tokens of the reference, not its own guesses. This is **teacher forcing**, as in [Lecture 2 \(Section 7\)](#) ([Williams and Zipser, 1989](#)). The decoder's inputs are its targets shifted right by one position, with BOS in front ([Figure 3](#)).

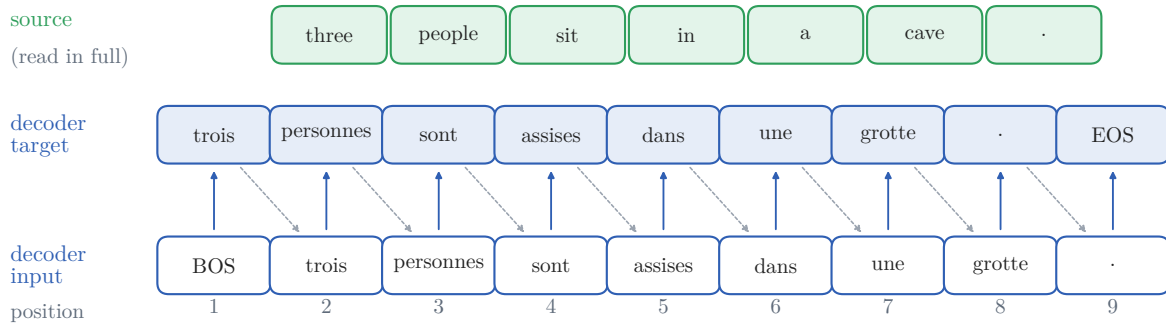


Figure 3. The decoder's inputs are its targets, shifted right by one position. A Multi30k test pair. The decoder reads BOS and the reference tokens (bottom row) and at each position is asked for the next reference token (top row), ending with EOS. In training every input is known in advance; in generation each input is the token chosen at the position before (gray arrows).

Three consequences follow:

1. **In training all inputs are known in advance**, but each prediction may still depend only on the inputs up to its position. The prediction of *assises* in [Figure 3](#) may read BOS, *trois*, *personnes*, and *sont*, but not *assises*, which is the next input.
2. **Whether the decoder can then compute all $m + 1$ positions at once is a property of the architecture.** A recurrent decoder cannot (§5); a convolutional decoder can (§8), and so can a Transformer (§10).
3. **In generation, the input at position $t + 1$ is the token chosen at position t** , so for every architecture the positions are produced one after another. The decoder then reads its own output so far, mistakes included, whereas in training it read only correct prefixes: a model trained by teacher forcing has never practiced recovering from an error.

Decoding

A trained model gives a probability to every target sentence, and translating means searching for a probable one. The most probable, $\arg \max_{\mathbf{y}} q(\mathbf{y} | \mathbf{x})$, cannot be found by trying every output: there are $|\mathcal{V}|^m$ outputs of length m over a target vocabulary $\mathcal{V}$. **Greedy decoding** writes the most probable token at each step, but a likely first word can lead to an unlikely sentence. **Beam search** keeps several partial translations, the **hypotheses**, each scored by its total log-probability, the sum of the log-probabilities of its tokens. [Figure 4](#) shows a beam of two at work, and the numbered steps below give the algorithm.

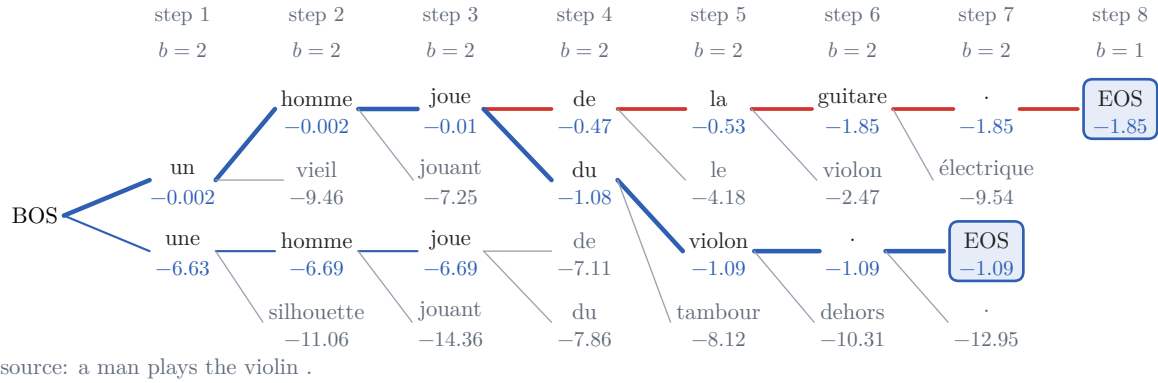


Figure 4. Beam search keeps a hypothesis that greedy decoding discards. A beam of two, with the attention model of §6, translating *a man plays the violin* . Each column is one step of the search, with its width b , and each word carries the total log-probability of its prefix. Every live hypothesis is shown with the extensions the search considers, its b best: blue ones are kept, gray ones dropped, and a boxed EOS is a finished translation. The bold blue path is the beam’s output; red marks the greedy path where it leaves the beam’s output. The first translation finishes at step 7, the width falls to one, and the second, the greedy one, finishes at step 8.

Beam search with a beam of size B runs as follows. The current width b starts at B and drops by one each time a translation finishes, so every finished translation uses up one place in the beam:

1. **Start.** There is one live hypothesis, BOS, with score 0; the list of finished translations is empty; the width is $b = B$.
2. **Extend.** Extend every live hypothesis by every token w ; the extension’s score is the hypothesis’s score plus $\log q(w \mid \text{hypothesis}, \mathbf{x})$. Only a hypothesis’s b best extensions can survive pruning (step 3), so in practice each hypothesis is extended only by its b most probable tokens.
3. **Prune.** Keep the b highest-scoring extensions and discard the rest, even those that end with EOS.
4. **Retire.** Move each kept extension that ends with EOS to the finished list, and lower b by one for each. The other kept extensions are the new live hypotheses.
5. **Repeat** steps 2–4 until $b = 0$. If a maximum length is reached first, the live hypotheses join the finished list.
6. **Output** the finished translation with the highest score.

Greedy decoding is the case $B = 1$. The search always ends with B finished translations, ranked by score, and each step ranks at most $b \times b$ extensions.

In Figure 4, greedy decoding and beam search diverge at step 4:

1. After *un homme joue*, greedy decoding takes *de*, the likelier word (prefix log-probability -0.47 , against -1.08 with *du*), and then continues with *la guitare*. The beam keeps both.
2. At step 7 the beam finishes *un homme joue du violon* . at -1.09 , and its width falls to one.
3. At step 8 its one live hypothesis, the greedy translation *un homme joue de la guitare* ., finishes at -1.85 . The width is zero, and the search stops.
4. The output is the higher-scoring translation, *un homme joue du violon* ., “a man plays the violin”, the right one. Greedy decoding writes “a man plays the guitar”.

The two finished translations have different lengths, seven tokens and eight, counting EOS. Over the 1,000 test captions, a beam of three, with the same model, changes 480 of the greedy

translations, finds a more probable one for 432, and raises BLEU (defined below) from 49.2 to 51.2. In the other 48 it ends with a less probable translation than greedy decoding, because the greedy path was pruned along the way: beam search, too, can miss the most probable translation. A translation can thus be wrong because of the model or because of the search; Lecture 8 returns to the search, including the sampling used for open-ended text.

Length normalization. Each token adds a negative log-probability, so longer translations have lower totals, and choosing the finished translation with the highest total favors short ones. Wider beams, which find more short translations, can then lower BLEU (Koehn and Knowles, 2017). Many systems choose instead by a score normalized for length,

$$\frac{\log q(\mathbf{y} \mid \mathbf{x})}{|\mathbf{y}|^\alpha},$$

where α , tuned on held-out data, lies between 0 (no normalization, as in Figure 4) and 1 (the average log-probability per token). Wu et al. (2016) divided by a smoothed length, $((5+|\mathbf{y}|)/6)^\alpha$, and added a penalty, computed from the attention weights of §6, for translations that seem to leave source words untranslated. Normalization changes only the final choice: the extensions ranked at any one step all have the same length, so normalizing would not change their order. In Figure 4 it would not change the output either: per token, the beam’s translation scores $-1.09/7 = -0.16$ and the greedy one $-1.85/8 = -0.23$.

Evaluation

The automatic measure of translation quality used throughout the years this lecture covers, BLEU (Papineni et al., 2002), compares a system’s output with the reference in three steps:

1. For each n from 1 to 4, the **modified precision** p_n is the share of the output’s n -grams (runs of n consecutive tokens) that also occur in the reference, each counted at most as many times as it occurs there: *the the the* earns one match against a reference with one *the*.
2. The four precisions fall roughly exponentially with n , so they are combined by their geometric mean, which averages their logarithms and gives each order equal weight.
3. A **breavity penalty** BP lowers the score of outputs shorter than the references; precision alone would reward a system that writes only the words it is sure of.

$$\text{BLEU} = \text{BP} \cdot \exp\left(\frac{1}{4} \sum_{n=1}^4 \log p_n\right), \quad \text{BP} = \min(1, e^{1-\ell_{\text{ref}}/\ell_{\text{out}}}),$$

where ℓ_{out} and ℓ_{ref} are the total lengths of the outputs and of the references. The counts are added over all the sentences of a test set before dividing, and BLEU is reported on a scale from 0 to 100; scores are comparable only on the same test set with the same tokenization.

Unigrams measure whether a translation has the right words, and longer n -grams whether they come in a fluent order. The highest order, 4, was chosen by experiment: Papineni et al. (2002) found that n -grams up to 4 agreed best with human judgments, with 3 and 5 giving comparable results. They also noted that, up to order 4, a precision of zero (which would make BLEU zero) should be rare on a test set of reasonable size.

For the pair of Figure 2, the attention model of §6 writes *une fille en tenue de karaté faisant un bâton avec un coup de pied* ., with *faisant*, “doing”, where the reference has *brisant*, “breaking”. The wrong word spoils every n -gram that contains it: 1 of the 15 unigrams, 2 of the 14 bigrams, 3 of the 13 trigrams, and 4 of the 12 four-grams. The output is as long as the reference, so BP = 1 and

$$\text{BLEU} = \left(\frac{14}{15} \cdot \frac{12}{14} \cdot \frac{10}{13} \cdot \frac{8}{12}\right)^{1/4} = 0.80,$$

reported as 80.0.

BLEU and the held-out log loss measure different things. The log loss scores the probabilities the model gives to the reference’s tokens, with the reference prefix given; it needs no search, and it is what training minimizes. BLEU scores the output of a search. The two can disagree: a better search raises BLEU without changing the model, as the beam did above, and a change to the model that raises the probability of a reference token it ranks second, without making it first, lowers the log loss but changes no greedy choice, so greedy BLEU stays the same. BLEU also rewards matching words, not meaning: a correct *une jeune fille* for the reference’s *une fille* loses points.

BLEU today. BLEU was the standard measure from 2002 through the years this lecture covers, and every published result quoted here is reported in it; these notes use it to compare those models on one scale. It is no longer the standard for judging translation. Learned metrics such as COMET (Rei et al., 2020) score an output with a neural network trained to predict human ratings, and they agree with human judgments much better: the 2022 study of evaluation metrics at WMT, the yearly Workshop (since 2016, Conference) on Machine Translation, was titled “Stop Using BLEU – Neural Metrics Are Better and More Robust” (Freitag et al., 2022). The 2024 WMT evaluation ranked systems by human judgments and by two learned metrics, “not using BLEU nor ChrF on purpose” (ChrF is a BLEU-like score over character n -grams) (Kocmi et al., 2024). BLEU remains common where a score must be comparable with earlier work.

3. Encoder-decoders

The neural translators of 2013–2014 divided the computation of $q(y_t \mid y_{<t}, \mathbf{x})$ between two networks with different jobs.

Two roles

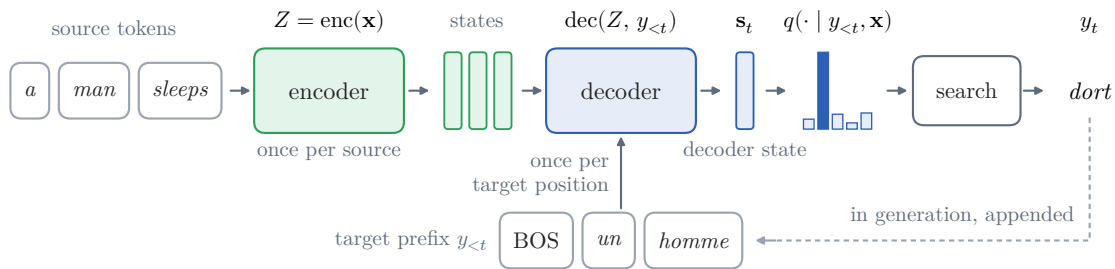


Figure 5. The encoder maps the source to states; the decoder maps the states and the target prefix to its own state, and that state to a distribution over the next token; a search chooses the token. In training the prefix is the reference’s (§2); in generation the chosen token is appended to it (dashed).

An **encoder** reads the source and computes a representation Z of it, and a **decoder** predicts the target from Z and the target prefix (Figure 5):

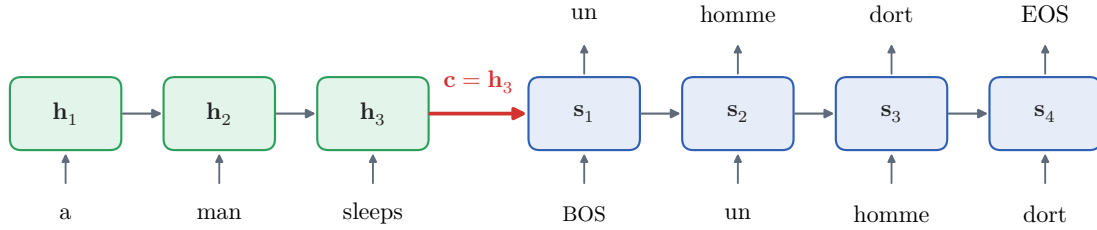
$$Z = \text{enc}(x_1, \dots, x_n), \quad q(\cdot \mid y_{<t}, \mathbf{x}) = \text{dec}(Z, y_0, \dots, y_{t-1}), \quad t = 1, \dots, m + 1.$$

The representation Z can be a single vector or one vector per source position. The encoder runs once per source; the decoder, a conditional language model of the target, runs once per target position. The two names describe what each part computes, not how. An encoder can be recurrent, convolutional, or built from attention, and so can a decoder, as long as its prediction

at step t depends only on Z and the prefix $y_{<t}$, never on y_t or the tokens after it. The decoder's **state** $\mathbf{s}_t$ is a vector from which it computes the distribution; the search of §2 chooses the token. Encoder states are written $\mathbf{h}_j$.

The recurrent encoder-decoder

(a) **c only at the start**



(b) **c also at every step**

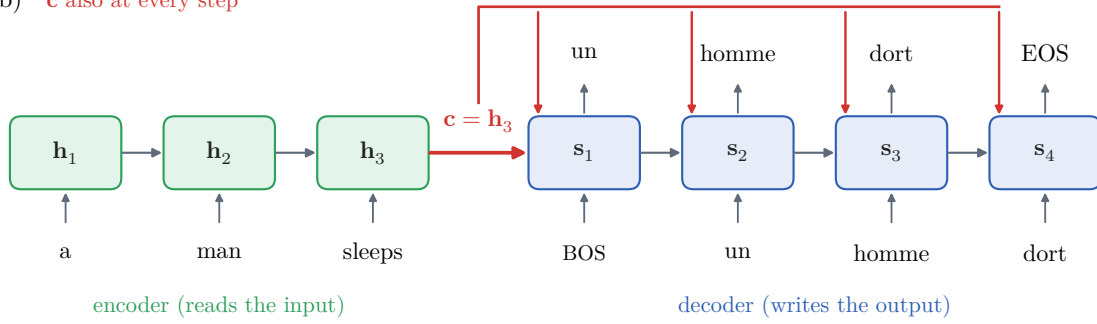


Figure 6. In a recurrent encoder-decoder, the whole source passes through one fixed-size summary. An encoder-decoder translating a three-word sentence. The encoder (green) reads the source; its last state is the context $\mathbf{c}$. (a) The context starts the decoder (blue), which writes the output one token at a time, reading each token it wrote as its next input. (b) The context also enters every step of the decoder and the decoder's output layer (not drawn).

In the recurrent encoder-decoder (Figure 6a), in the notation of Lecture 2 (Section 7):

1. The encoder, a recurrent network, reads the source one token at a time: $\mathbf{h}_j = f_{\text{enc}}(\mathbf{h}_{j-1}, E(x_j))$ for $j = 1, \dots, n$.
2. Its last state is the **context**, $\mathbf{c} = \mathbf{h}_n$. Here the representation Z is this one vector.
3. The decoder, a recurrent language model of the target, starts from the context, $\mathbf{s}_0 = \mathbf{c}$, and reads the target prefix: $\mathbf{s}_t = f_{\text{dec}}(\mathbf{s}_{t-1}, E(y_{t-1}))$ for $t = 1, \dots, m + 1$.
4. Each state gives the distribution of the next token, through a matrix U that maps a state to one score per target token: $q(y_t | y_{<t}, \mathbf{x}) = \text{softmax}(\mathbf{s}_t U)$.

The updates f_{enc} and f_{dec} are recurrent cells with their own weights. Training backpropagates the loss of §2 through both networks at once, so the encoder learns to put into $\mathbf{c}$ what the decoder needs. The size of $\mathbf{c}$ is fixed by the architecture: a sentence of 5 words and a sentence of 50 are summarized in the same number of numbers.

When the two networks' states differ in size, a learned matrix W_b , a **bridge**, maps one to the other: $\mathbf{s}_0 = \mathbf{c}W_b$. Cho et al. (2014a) and Bahdanau et al. (2015) also applied a tanh, $\mathbf{s}_0 = \tanh(\mathbf{c}W_b)$ (Bahdanau et al. bridged the right-to-left encoder state at the first word, §6), which keeps the starting state between -1 and 1 , the range of the states that their cells produce; both forms are used. The small models of §4 need no bridge: their decoder has as many units as the two directions of their encoder together.

The context at every step. Started from $\mathbf{c}$, the decoder sees the source once and must carry it through every update. [Cho et al. \(2014a\)](#) also gave the context to every update and to the output layer ([Figure 6b](#)):

$$\mathbf{s}_t = f_{\text{dec}}(\mathbf{s}_{t-1}, E(y_{t-1}), \mathbf{c}), \quad q(y_t | y_{<t}, \mathbf{x}) = \text{softmax}([\mathbf{s}_t; \mathbf{c}] U),$$

where U now maps the state and the context, joined into one row, to the scores. Every prediction is then one step from $\mathbf{c}$, however long the prefix. (Cho et al. placed one more layer, reading the state, the context, and the previous word's vector, before the softmax; the linear map written here is the simplest choice.) Neither version gives the decoder access to the separate source positions: both pass the whole source through $\mathbf{c}$.

A bidirectional encoder, a causal decoder. The source is given in full before translation begins, so the encoder may read it in both directions. A **bidirectional** encoder ([Schuster and Paliwal, 1997](#)) runs one recurrent network from left to right and another from right to left and joins their states, $\mathbf{h}_j = [\bar{\mathbf{h}}_j; \tilde{\mathbf{h}}_j]$, so that the state at position j summarizes the whole sentence around word j . The decoder may not do the same with the target: in generation the later tokens do not exist yet, and in training a decoder that read them would learn to copy its answer, since each target is the next position's input ([Figure 3](#)). A decoder whose prediction at each position depends only on the tokens before it is called **causal** or **autoregressive**. The constraint is on what each prediction may read, not on the order of computation: the convolutional decoders of §8 and the Transformer compute all positions of a decoder layer at once in training, without letting any of them read the positions after it.

Recurrent cells

The cells f_{enc} and f_{dec} can be any recurrent cell of [Lecture 2 \(Sections 7–8\)](#). The cell changes what each state holds and how information and gradients travel from step to step; the roles, the training, and the use of one fixed-size context stay the same. [Figure 7](#) opens one box of [Figure 6](#) for each of three cells.

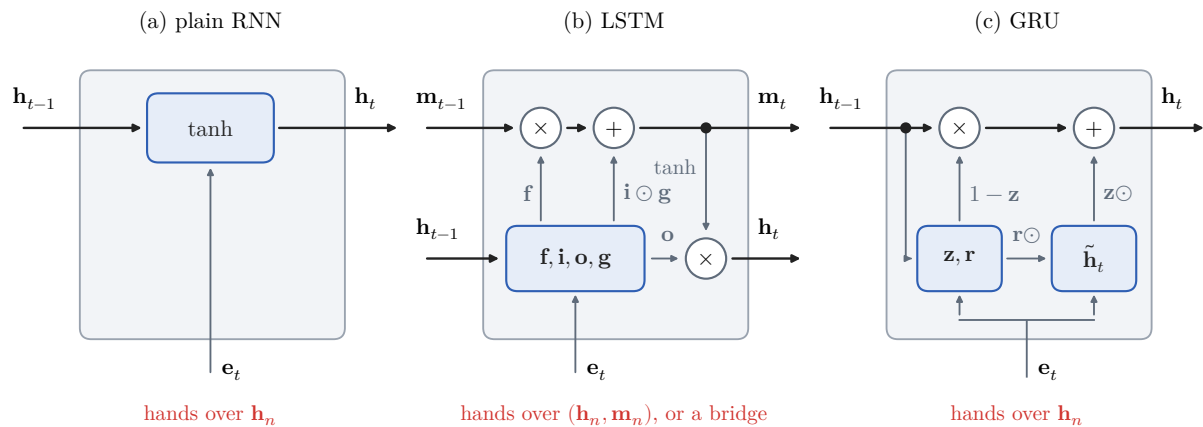


Figure 7. The same step, with three different cells inside. Each panel is one box of [Figure 6](#), opened: the state carried from step to step runs along the black lines, and the input vector $\mathbf{e}_t$ enters from below. (a) A plain RNN rewrites its state through one $\tanh$ layer. (b) An LSTM carries a memory cell $\mathbf{m}_t$ along the top line, only scaled by the forget gate and added to, and a hidden state $\mathbf{h}_t$ along the bottom line, read out from the cell through a $\tanh$ and the output gate. (c) A GRU carries one state; it keeps a share $1 - \mathbf{z}$ of it and replaces a share $\mathbf{z}$ by the candidate $\tilde{\mathbf{h}}_t$, which sees the old state through the reset gate $\mathbf{r}$. Below each panel, what the encoder's last step hands to the decoder.

A plain RNN rewrites its state through one $\tanh$ layer at every step, and information and gradients fade along a long sentence ([Lecture 2, Section 7](#)). An LSTM carries, besides its hidden

state $\mathbf{h}_t$, a memory cell $\mathbf{m}_t$ that is only scaled by the forget gate and added to, with no squashing nonlinearity in between,

$$\mathbf{m}_t = \mathbf{f}_t \odot \mathbf{m}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t, \quad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{m}_t),$$

with input, forget, and output gates $\mathbf{i}_t$, $\mathbf{f}_t$, $\mathbf{o}_t$ and a candidate $\mathbf{g}_t$, each computed from $\mathbf{h}_{t-1}$ and the input vector $\mathbf{e}_t$. (Lecture 2 wrote the cell $\mathbf{c}_t$; here $\mathbf{c}$ is the context.) The cell carries information, and gradients, over more steps than a plain RNN's state does. An LSTM encoder hands the decoder both its hidden state and its cell.

The **gated recurrent unit** (GRU) was introduced with one of the first encoder-decoders (Cho et al., 2014a) and named later (Chung et al., 2014). Cho et al. added gates because, in their words, “We were not able to get meaningful result with an oft-used tanh unit without any gating.” The GRU has no separate cell: its state is the memory. Two gates and a candidate are computed from the previous state and the input,¹

$$\begin{aligned} \mathbf{z}_t &= \sigma(\mathbf{h}_{t-1}W_z + \mathbf{e}_tV_z + \mathbf{b}_z) && \text{update gate: how much of each unit to replace} \\ \mathbf{r}_t &= \sigma(\mathbf{h}_{t-1}W_r + \mathbf{e}_tV_r + \mathbf{b}_r) && \text{reset gate: how much of the old state the candidate sees} \\ \tilde{\mathbf{h}}_t &= \tanh((\mathbf{r}_t \odot \mathbf{h}_{t-1})W + \mathbf{e}_tV + \mathbf{b}) && \text{candidate} \\ \mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t. \end{aligned}$$

The last line says that each unit's new value is a weighted average of its old value and the candidate, with weights set by the update gate. Four consequences:

1. A unit whose update gate stays near 0 keeps its value, and its gradient, across many steps, as an LSTM cell does when its forget gate stays near 1.
2. Writing and forgetting are tied: each unit forgets exactly the share $\mathbf{z}_t$ of its old value that it overwrites with the candidate, and the state stays between -1 and 1 without a further squashing step.
3. With the reset gate near 0 for all units, the candidate is computed from the input alone, so the state can start afresh.
4. There is no output gate: the whole state is visible to the rest of the network.

The GRU has three sets of weights where the LSTM has four; Lecture 2 (Section A.8) derives its gradient path.

The first encoder-decoders

Kalchbrenner and Blunsom (2013) encoded the source with a convolutional network and decoded with a recurrent language model to which they added a representation of the source at every step; they used the whole model to rescore (re-rank) the candidate translations of a statistical system.² Cho et al. (2014a), posted in June 2014, called their model the **RNN Encoder-Decoder**: one recurrent network “encodes a sequence of symbols into a fixed-length vector

¹Lecture 2, these notes, and Bahdanau et al. (2015) write the update as $(1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$, so that $\mathbf{z}_t$ is the share replaced. Cho et al.'s original paper puts the gate on the old state, so that there $\mathbf{z}_t$ is the share kept, and software libraries differ as well. The two are the same unit with the gate relabeled, but a formula from one source cannot be combined with a gate value from the other.

²Kalchbrenner and Blunsom's first model added one source vector to the decoder's hidden state at every step; their second, which gave the better results, predicted the length of the translation and expanded the source representation into one vector per target position. Used with a penalty on the number of words to rescore the 1,000 best candidates of a statistical system, the second model came within 0.1 BLEU of that system's own choices (21.7 against 21.8 on the 2012 English-French news test set). Cho et al.'s feature raised a phrase-based system's BLEU on the 2014 English-French test set from 33.30 to 33.87.

representation, and the other decodes the representation into another sequence of symbols”. Their name for the design stuck. They used it on phrases of a few words, to score the phrase pairs of a phrase-based system: the network’s score for each phrase pair became one more feature, an input to the system’s scoring of translations, and this feature improved the system’s translations slightly. [Sutskever et al. \(2014\)](#) made the network the whole translator (§4).

The three designs differ mainly in how the decoder receives the source and in what the network was used for:

Model	Encoder	Decoder receives the source	Used as
Kalchbrenner and Blunsom (2013)	Convolutional	At every step	A scorer for a statistical system
Cho et al. (2014a)	Recurrent (GRU)	At the start and at every step	A feature of a phrase-based system
Sutskever et al. (2014)	Deep LSTM, reversed source	At the start only	The whole translator (§4)

4. Sequence to sequence: a complete neural translator

In September 2014, [Sutskever et al. \(2014\)](#) translated English into French with a single network: a recurrent encoder–decoder, made deep and large. Their abstract presents it as “a general end-to-end approach to sequence learning that makes minimal assumptions on the sequence structure”, and nothing in the model is specific to translation.

The model

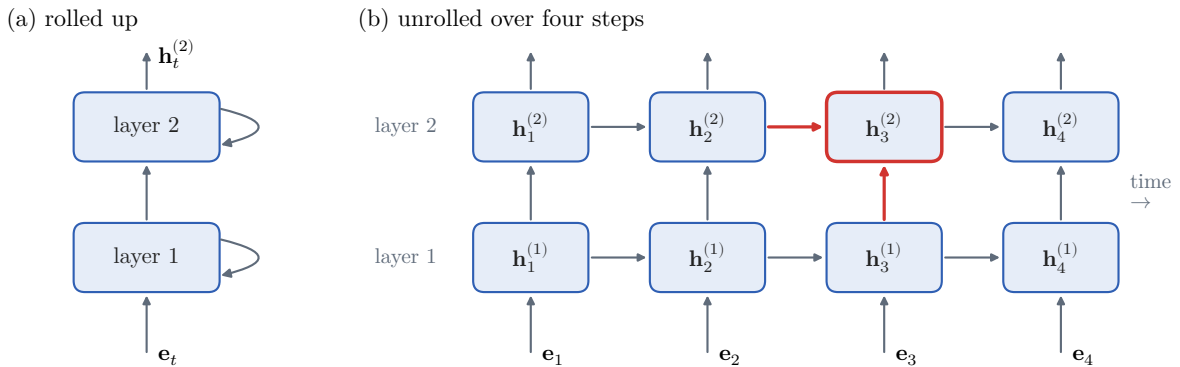


Figure 8. A deep recurrent network, rolled up and unrolled. (a) Two recurrent layers stacked: each has its own loop, and the layer above reads the output of the layer below. (b) The same network unrolled over four steps. The state of layer ℓ at step t is computed from the state of the layer below at the same step and from its own state at the step before; the two inputs of $\mathbf{h}_3^{(2)}$ are in red. In an LSTM each box also carries a memory cell.

The encoder and the decoder are both **deep**, or stacked, LSTMs: L LSTM layers, where at each step layer ℓ reads the hidden state that the layer below computed at that step (Figure 8). For a network that reads input vectors $\mathbf{e}_1, \mathbf{e}_2, \dots$ (the source words in an encoder, the previous target words in a decoder),

$$(\mathbf{h}_t^{(\ell)}, \mathbf{m}_t^{(\ell)}) = \text{LSTM}_\ell(\mathbf{h}_t^{(\ell-1)}, \mathbf{h}_{t-1}^{(\ell)}, \mathbf{m}_{t-1}^{(\ell)}), \quad \ell = 1, \dots, L, \quad \mathbf{h}_t^{(0)} = \mathbf{e}_t,$$

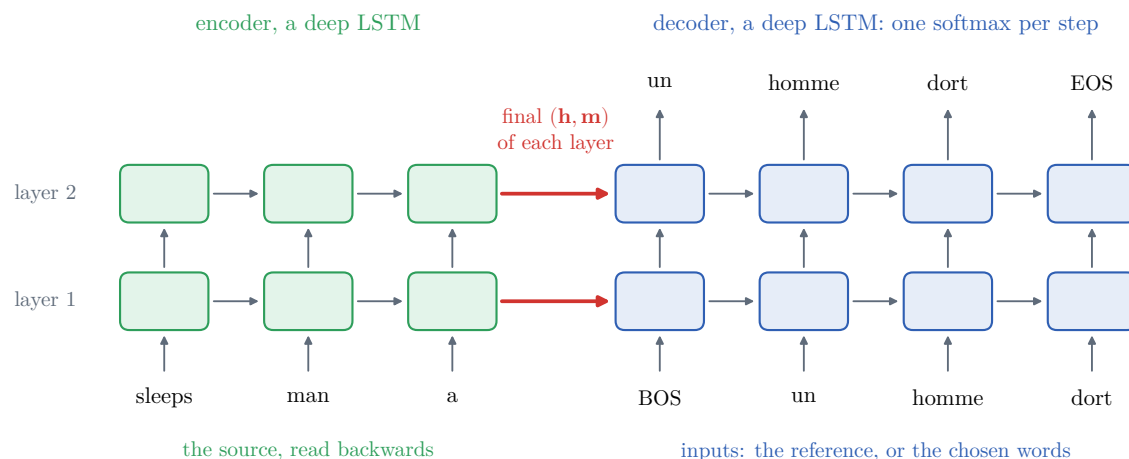


Figure 9. One sentence through a sequence-to-sequence model. A two-layer LSTM encoder (green) reads the source backwards; the hidden state and memory cell of each of its layers start the decoder layer at the same depth (red; presumably, see the text). The decoder (blue) reads BOS and then the target, and its top layer gives a distribution over the target vocabulary at every step. In training the inputs are the reference tokens; in generation each input is the token chosen at the step before. Sutskever et al. used four layers of 1,000 units in each network.

each layer with its own weights. Information moves along two axes, forward in time within each layer and upward through the layers at each step, and the top layer’s hidden state $\mathbf{h}_t^{(L)}$ is the output. Stacking recurrent layers was not new: Graves et al. (2013) had recognized speech with deep LSTMs. Sutskever et al. (2014) found that depth paid off in translation: deep LSTMs “significantly outperformed shallow LSTMs”, each further layer lowering the perplexity by nearly 10%; they suggested that the gain might come from the deeper network’s larger hidden state.

The translator is the recurrent encoder–decoder of §3 with a deep LSTM in each role (Figure 9), trained and decoded as in §2. It differs from the encoder–decoder of §3 in its size and in the way the source reaches the decoder. The encoder has four layers of 1,000 units. Its inputs are 1,000-dimensional word vectors over an English vocabulary of 160,000 words, and it reads the source backwards: *a man sleeps* is read as *sleeps, man, a*. The decoder, a second four-layer LSTM with its own weights, starts from the encoder’s final states. The paper does not say exactly how; presumably decoder layer ℓ begins where encoder layer ℓ ended.³ Its top layer gives a distribution over a French vocabulary of 80,000 words. There is no attention: everything the decoder knows about the source is in the states it started from. In Figure 9, the only route from *sleeps* to the prediction of *dort* runs through the two encoder steps after it (*man, a*), the encoder’s final states, and three decoder steps.

³The paper says that the decoder’s “initial hidden state is set to the representation” of the source, which is “the last hidden state of the LSTM”. It also says that the network “uses 8000 real numbers to represent a sentence”: four layers, each with a hidden state of 1,000 numbers and a memory cell of 1,000, make 8,000 numbers, which suggests that all four layers hand over both, although the paper does not say so explicitly.

Reading the source backwards

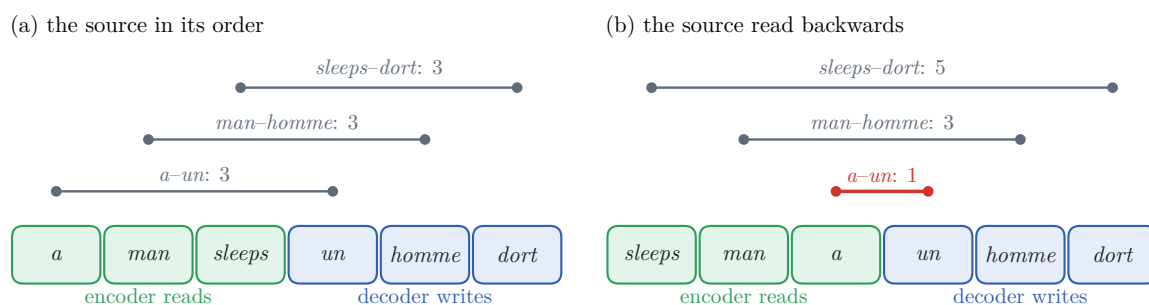


Figure 10. Reading the source backwards puts the first words of the two sentences next to each other. The encoder reads the source (green), then the decoder writes the target (blue); each bar joins a source word to its translation and is labeled with the number of steps between them. (a) In the original order every pair is three steps apart. (b) Read backwards, the pairs are one, three, and five steps apart: the same average, but the first words are adjacent.

The reversal was found by experiment: reading the source backwards lowered a single model’s test perplexity from 5.8 to 4.7 and raised its BLEU from 25.9 to 30.6. The authors offered an explanation, while saying they did not have “a complete explanation” (Figure 10):

1. The model reads the whole source before it writes the translation, so each source word is many steps from the word that translates it.
2. Reversing the source leaves the average distance between corresponding words unchanged, but puts the first source words next to the first target words.
3. With these short paths, they wrote, backpropagation “has an easier time” establishing communication between the source sentence and the target sentence.

They had expected reversal to help the beginning of the translation and hurt its end. Instead, the reversed models also translated long sentences better, which they took to mean that the LSTM used its memory better.

What the summary holds

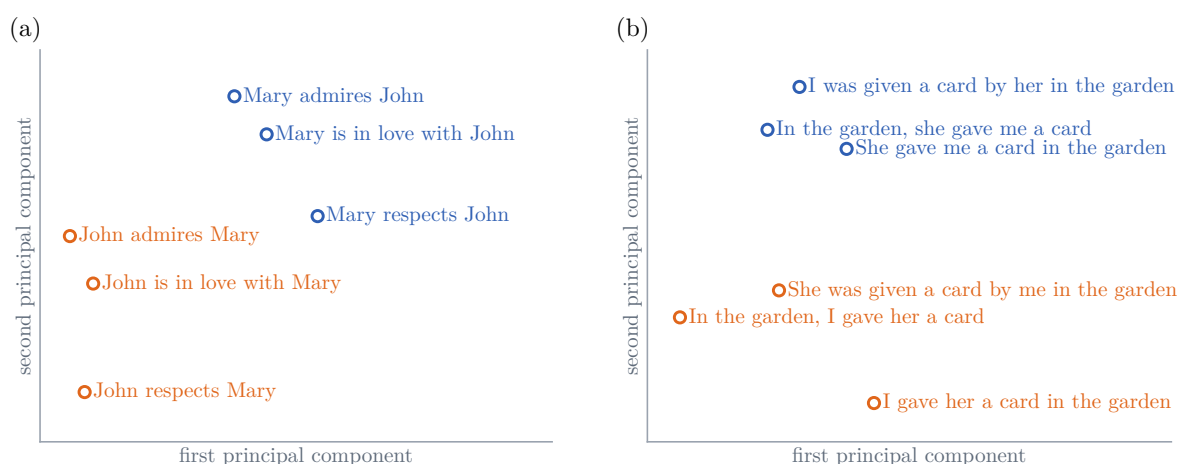


Figure 11. The summary depends on word order, much less on voice. The states of Sutskever et al.’s encoder after reading each phrase, projected onto their first two principal components, redrawn from the second figure of their paper. Blue and orange mark the two meanings in each panel. (a) The same words in a different order land far apart. (b) A sentence in the active voice and in the passive voice land in the same region.

What does the state of fixed size hold? Sutskever et al. ran a few phrases through their encoder and projected the final states onto two dimensions (Figure 11). Phrases with the same meaning landed close together, while phrases with the same words in another order, such as *John admires Mary* and *Mary admires John*, landed far apart: the summary is not a bag of words. In their words, the representations are “sensitive to the order of words, while being fairly insensitive to the replacement of an active voice with a passive voice”.

Results

Trained on 12 million sentence pairs of the English–French task of the 2014 Workshop on Machine Translation (WMT 2014), the model had 384 million parameters and took about ten days on eight GPUs. An ensemble of five reversed models, which combine their predictions, scored 34.8 BLEU on the test set with a beam of 12, above the 33.3 of the phrase-based baseline; a single model scored 30.6. Rescoring the baseline’s 1,000 best candidates, the ensemble reached 36.5, against 37.0 for the best system of that year’s evaluation. And, to the authors’ surprise, the model “did not suffer on very long sentences” (its BLEU dropped only slightly on the longest sentences, §5). A neural network trained on sentence pairs alone had beaten a phrase-based baseline. (The order of these notes is conceptual: the attention paper of §6 was posted nine days earlier.)

Two small encoder–decoders. Two small models, trained once each for these notes with the scripts in this lecture’s folder, make the drawbacks of §5 measurable. §6 trains them again with attention; this week’s exercise uses the translation model, with and without attention.

Setting	Reversal (seq2seq.py)	Translation (translate.py)
Task	Write a string of 1 to 40 random letters backwards, a task that needs memory and no knowledge of language	Multi30k English to French: 29,000 training, 1,014 validation, and 1,000 test captions, lower-cased and split into words
Encoder	Bidirectional LSTM, 64 units each way	Bidirectional LSTM, 128 units each way, on word vectors of 128 numbers
Decoder	LSTM of 128 units, started from the encoder’s final hidden states and cells, both directions side by side	LSTM of 256 units, started likewise; vocabularies of the 5,894 English and 6,430 French words that occur at least twice
Training	4,000 steps of 64 new random strings	3,000 steps of 64 pairs of similar length; Adam, learning rate 0.002 decaying to 0.0002; dropout 0.3; 4.0 million parameters; about 9 minutes on two processor cores

Unlike Sutskever et al.’s, both encoders read the source in both directions instead of backwards. Decoding greedily, the translation model scores 19.5 BLEU on the test captions. For the pair of Figure 2 it writes *une fille en tenue de bain rouge regardant un objet dans un champ* ., “a girl in a red swimsuit looking at an object in a field”: fluent French whose first words follow the source and whose rest does not.

5. Three drawbacks of the recurrent encoder–decoder

Sequence to sequence works, and at scale it works well. Its design still has three drawbacks: the whole source reaches the decoder through a **fixed-size summary**; the states are computed **in sequence**, one position after another; and related positions are joined by **long paths**. They are worth keeping apart, because the remedies of 2014–2017 addressed them separately, and a remedy for one can leave the others untouched.

The fixed-size summary

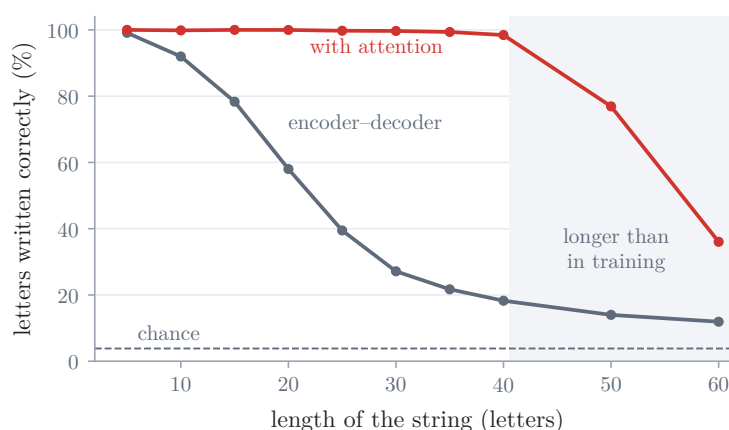


Figure 12. The fixed-size summary fails on long strings. The share of letters written correctly, against the length of the string, on the reversal task, for the LSTM encoder–decoder (gray) and the same network with attention (red, §6). Both were trained on strings of 1 to 40 letters; longer strings (shaded) were never seen in training.

However long the input, the decoder sees it through a fixed number of numbers. The reversal task of §4 isolates this limit, since reversing a string needs nothing but memory (Figure 12, gray). The encoder–decoder writes 92% of the letters of 10-letter strings in the right place, but only 58% of those of 20-letter strings and 18% of those of 40-letter strings, although it was trained on strings of up to 40 letters. The more there is to store, the less survives the passage through a state of fixed size.

Translation shows the same limit, to a degree that depends on the model. Cho et al. (2014b), comparing encoder–decoders with a phrase-based system, found that the neural model “performs relatively well on short sentences without unknown words, but its performance degrades rapidly as the length of the sentence and the number of unknown words increase”. Sutskever et al., with a much larger network reading reversed sentences, found no degradation on sentences of fewer than 35 words and only a minor one on the longest. A fixed-size summary strains a model more the longer the source; whether accuracy falls depends on the size of the summary, the training, and the data.

Sequential computation and long paths

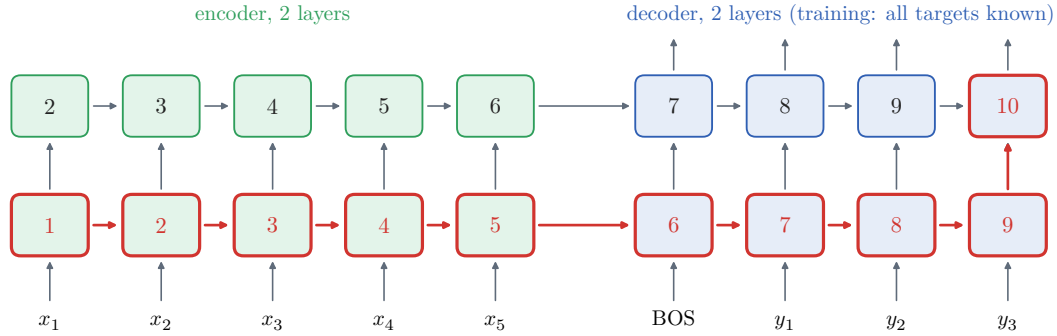


Figure 13. Recurrence fixes the order of computation. A two-layer encoder–decoder reads five words and, in training, runs the decoder over four positions, whose inputs are BOS and the three target words and whose outputs are the three target words and EOS. The number in each state is the earliest step at which it can be computed, even with one processor per state: each state needs the state to its left and the one below it. The longest chain of dependent states (red) has $n + r + L - 1 = 5 + 4 + 2 - 1 = 10$ steps.

The state at position j cannot be computed before the state at $j - 1$. A sentence of n tokens takes n steps in sequence in every layer of the encoder, and the decoder’s states must also be computed in order, even in training, when teacher forcing supplies every target token in advance (§2). Processors can share the work across the sentences of a batch or the layers of a network, but not along a sentence. Figure 13 counts the steps. Let the decoder run over $r = m + 1$ positions (the m target tokens and EOS), and let each of its L layers start from the final state of the encoder layer at the same depth, as in the figure and, presumably, in Sutskever et al.’s model. Then one training example takes

$$n + r + L - 1$$

dependent steps: one per source token, one per target position, and one per extra layer. The count grows with the lengths of the sentences and does not depend on the width of the layers. The Transformer paper put the cost in one sentence: “This inherently sequential nature precludes parallelization within training examples” (Vaswani et al., 2017). Google’s production system of 2016, with eight LSTM layers in the encoder and eight in the decoder, took about six days on 96 GPUs to train one English-to-French model (Wu et al., 2016).

The same chain makes the paths between positions long. A dependency between source position j and target position t travels through every recurrent step between them: forward, for the information, and backward, for the gradient that teaches the network to use it. In Figure 13, the first source word reaches the first prediction through every encoder step. Lecture 2 (Section 7) showed what happens to a gradient along such a path: it is multiplied by a Jacobian at every step, and usually shrinks.

Familiar remedies

Each remedy available in 2014 changed at most one of the three drawbacks:

Remedy	Fixed-size summary	Sequential computation	Long paths
Gated cell (LSTM, GRU)	Unchanged	Unchanged	Eased, not shortened
Larger state	Eased	Unchanged	Unchanged
Bidirectional encoder	Unchanged	Unchanged	Some routes shortened
Context at every step	Unchanged	Unchanged	Shortened between $\mathbf{c}$ and each prediction
Reversed source	Unchanged	Unchanged	Rearranged

Gating eases the long paths without shortening them: the path from the first source word to the decoder still has n steps, but the memory cell carries the signal across them with less loss. A bidirectional encoder's final states lie at both ends of the sentence (the left-to-right state at the last word, the right-to-left state at the first), but the words in the middle are still about $n/2$ steps from both. None of these remedies gives the decoder more than one fixed summary of the source, or lets the states of a sequence be computed at once. Attention frees the decoder from the single summary and shortens the paths from the source to the target; it leaves the computation sequential, which §§8–10 take up.

6. Attention

Can the decoder retrieve a different part of the source for each target word? Bahdanau et al. (2015), in a paper posted in September 2014, answered by giving up the single fixed-size summary. Their abstract states the diagnosis: “we conjecture that the use of a fixed-length vector is a bottleneck in improving the performance of this basic encoder-decoder architecture”. The encoder keeps a state for every position of the source, and at every step the decoder reads a weighted average of them, with weights computed by comparing its own state with each of them (Figure 14).

The read

At output step t , before predicting y_t , the decoder reads the source in three steps:

1. **Score** every source position against the decoder's previous state: $e_{tj} = a(\mathbf{s}_{t-1}, \mathbf{h}_j)$ for $j = 1, \dots, n$.
2. **Normalize** the scores into weights that sum to one: $\alpha_{tj} = \exp(e_{tj}) / \sum_{k=1}^n \exp(e_{tk})$.
3. **Average** the encoder states with these weights: $\mathbf{c}_t = \sum_{j=1}^n \alpha_{tj} \mathbf{h}_j$.

The context now has an index: it is recomputed at every step, from all n encoder states. The encoder is bidirectional (§3), so $\mathbf{h}_j = [\bar{\mathbf{h}}_j; \mathbf{h}_j]$ summarizes the whole source around word j .

The query. The vector that the source positions are scored against is called the **query**; here it is $\mathbf{s}_{t-1}$, the state from which the distribution of y_{t-1} was computed. It has read BOS, the target tokens up to y_{t-2} , and the earlier contexts, but not y_{t-1} , which the decoder reads only in the update that computes $\mathbf{s}_t$; §7 returns to this timing.

The states. Each $\mathbf{h}_j$ represents word x_j in the context of its sentence, and does two jobs: it is compared with the query, to decide how relevant position j is, and it is the content returned when position j is read. §7 separates the two jobs into a **key** and a **value**.

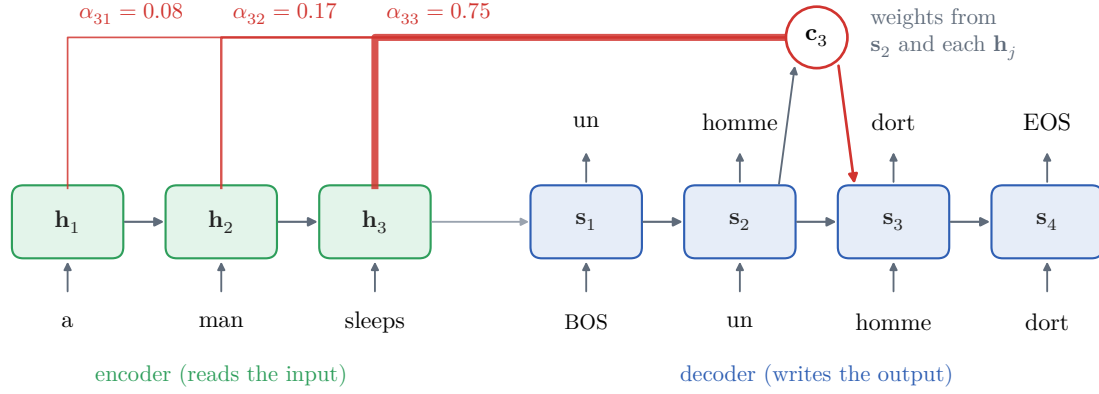


Figure 14. With attention, the decoder reads every source position at every step. The encoder keeps a state for every source position, and when writing each word the decoder reads a weighted average of them, with weights recomputed from its own previous state (shown for the third output word). Compared with Figure 6a, one context $\mathbf{c} = \mathbf{h}_n$ reaching the decoder once is replaced by every $\mathbf{h}_j$ reaching every step. The weights are illustrative, and the context also enters the decoder's output layer (not drawn). For the comparison, the encoder is drawn reading in one direction and handing its last state to the decoder (gray arrow); Bahdanau et al.'s encoder reads in both directions.

The score. Bahdanau et al. used a network with one hidden layer, called the **additive** score because the projected query and state are added inside the nonlinearity:

$$a(\mathbf{s}, \mathbf{h}) = \tanh(\mathbf{s}W_a + \mathbf{h}U_a) \mathbf{v}_a^\top,$$

with $W_a \in \mathbb{R}^{d_s \times d_a}$, $U_a \in \mathbb{R}^{d_h \times d_a}$, and $\mathbf{v}_a \in \mathbb{R}^{d_a}$, for decoder states of d_s numbers, encoder states of d_h , and a hidden layer of d_a units (Bahdanau et al. used $d_s = d_a = 1,000$ and $d_h = 2,000$). Its parameters are trained with the rest of the network. The products $\mathbf{h}_j U_a$ do not depend on t , so they are computed once per sentence.

The softmax. The weights are positive and sum to one, so the context is a weighted average of the encoder states, and has the size of one state whatever the length of the source. The positions compete for the weight: raising one score lowers the other weights, and adding the same constant to all the scores changes nothing.

The read sees a set. Given the query $\mathbf{s}_{t-1}$, the context $\mathbf{c}_t$ depends on the encoder states only as an unordered collection $\{\mathbf{h}_1, \dots, \mathbf{h}_n\}$: permuting the source positions permutes the scores and the weights in the same way, and leaves $\mathbf{c}_t$ unchanged. Nothing in the read knows where a state came from. Word order reaches the read only through the states themselves, which the recurrent encoder computed in order, so that $\mathbf{h}_j$ reflects the words around position j and where they stand. §9 returns to this property.

The context in the decoder. The new context enters the decoder's update and its output layer, as the fixed context did in §3:

$$\mathbf{s}_t = f_{\text{dec}}(\mathbf{s}_{t-1}, E(y_{t-1}), \mathbf{c}_t), \quad q(y_t | y_{<t}, \mathbf{x}) = \text{softmax}([\mathbf{s}_t; \mathbf{c}_t] U).$$

Bahdanau et al.'s decoder was a GRU whose candidate and both gates received $\mathbf{c}_t$ as an extra input. Like Cho et al., they placed one more layer before the softmax; it reads $\mathbf{s}_t$, $E(y_{t-1})$, and $\mathbf{c}_t$, and is made of maxout units, each the larger of two linear functions of its inputs.⁴ The decoder starts from the bridge of §3 applied to the right-to-left encoder state at the first position, $\mathbf{s}_0 = \tanh(\bar{\mathbf{h}}_1 W_b)$; that state has read the whole sentence.

⁴The paper's main text conditions the output on the new state $\mathbf{s}_t$, as written here; its appendix writes the previous state $\mathbf{s}_{t-1}$ in the output layer.

One read by hand

Take three source states of two numbers each, a decoder state, and an additive score with a hidden layer of two units ($d_s = d_h = d_a = 2$):

$$\mathbf{s}_{t-1} = (1, 2), \quad W_a = \begin{pmatrix} 1 & 0.5 \\ 0 & -0.75 \end{pmatrix}, \quad U_a = \begin{pmatrix} 0.5 & 0.5 \\ -1 & 1 \end{pmatrix}, \quad \mathbf{v}_a = (2, 1),$$

$$\mathbf{h}_1 = (2, 0), \quad \mathbf{h}_2 = (0, 1), \quad \mathbf{h}_3 = (1, 1).$$

Figure 15 shows where the computation ends: most of the weight on $\mathbf{h}_1$, and a context inside the triangle of the three states.

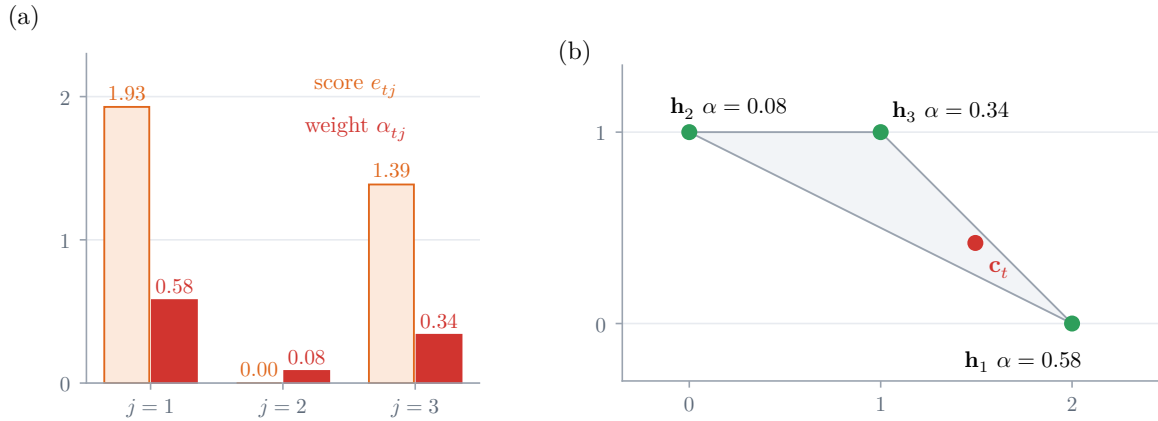


Figure 15. The read of the worked example. (a) Scores and weights for the three source positions: the additive scores e_{tj} and the weights α_{tj} after the softmax. (b) The source states as points in the plane. Every context is a weighted average of them, so it lies in their triangle; this one, $\mathbf{c}_t$, is closest to $\mathbf{h}_1$.

The query's part, $\mathbf{s}_{t-1}W_a = (1, -1)$, is computed once per step, and each state's part $\mathbf{h}_jU_a$ once per sentence; their sum is $\mathbf{z}_j = \mathbf{s}_{t-1}W_a + \mathbf{h}_jU_a$, and $e_{tj} = \tanh(\mathbf{z}_j) \mathbf{v}_a^\top$:

j	$\mathbf{h}_jU_a$	$\mathbf{z}_j$	$\tanh(\mathbf{z}_j)$	e_{tj}	α_{tj}
1	(1, 1)	(2, 0)	(0.964, 0)	1.928	0.579
2	(-1, 1)	(0, 0)	(0, 0)	0	0.084
3	(-0.5, 1.5)	(0.5, 0.5)	(0.462, 0.462)	1.386	0.337

The exponentials of the scores are 6.88, 1, and 4.00, which sum to 11.88, and the context is

$$\mathbf{c}_t = 0.579(2, 0) + 0.084(0, 1) + 0.337(1, 1) = (1.495, 0.421).$$

A plain dot product with the query would have ranked the states differently: $\mathbf{s}_{t-1}\mathbf{h}_j^\top$ is 2, 2, and 3, which puts $\mathbf{h}_3$ first. The additive score compares the query with each state through W_a , U_a , and $\mathbf{v}_a$, which training sets, not by their raw similarity. This week's exercise writes the read as a function of a batch of queries and source states, checks it on these numbers, and places it inside the trained translation model below.

Training without alignment labels

Attention learns where to read from the translation loss alone: nobody tells the network which source word corresponds to which target word, and the correspondences emerge without labels. The loss of the next word reaches every score through the context. Let $\gamma_t = \partial L / \partial \mathbf{c}_t$, the gradient

that backpropagation brings to the context from the rest of the decoder. The derivative of the softmax gives (Appendix A.1)

$$\frac{\partial L}{\partial e_{tj}} = \alpha_{tj} (\mathbf{h}_j - \mathbf{c}_t) \gamma_t^\top.$$

The gradient pushes up the score of each position whose state lies, as seen from the current context, in the direction that would lower the loss ($\mathbf{h}_j - \mathbf{c}_t$ points against γ_t), and pushes down the scores of positions that lie the other way. The pushes sum to zero, since $\sum_j \alpha_{tj} (\mathbf{h}_j - \mathbf{c}_t) = 0$: weight moves between positions and is never created. Each push is also scaled by the position's weight, so a position that the read ignores gets little signal through its own score. To first order, then, training moves each read toward the source positions whose states help to predict the next word. In translation these are mostly the words being translated, so the correspondences of Figure 2 appear as a by-product of predicting the next word; Figure 18 below shows them in the small model. Appendix A.1 works the gradient through the example above and on to the parameters of the score.

Results

Bahdanau et al. trained on the 348 million words of the WMT 2014 English–French data, with vocabularies of the 30,000 most frequent words in each language (the rest replaced by an unknown-word token), on one GPU for about five days per model. Their models scored (BLEU on the test set):

System	All sentences	No unknown words
Encoder–decoder, trained on sentences of up to 30 words	13.93	24.19
With attention, up to 30 words	21.50	31.44
Encoder–decoder, up to 50 words	17.82	26.71
With attention, up to 50 words	26.75	34.16
With attention, up to 50 words, trained longer	28.45	36.15
Phrase-based system (Moses)	33.30	35.63

With the same data, vocabularies, and training, the attention models scored 7.57 points (trained on sentences of up to 30 words) and 8.93 points (up to 50 words) above the encoder–decoders, and on the sentences without unknown words the best of them was ahead of the phrase-based system. The encoder–decoder's BLEU fell steadily with the length of the sentence, while the attention model trained on sentences of up to 50 words showed “no performance deterioration even with sentences of length 50 or more”. These numbers should not be ranked with Sutskever et al.'s, which came from larger models, larger vocabularies, ensembles, and different training; the evidence for attention is the comparison within Bahdanau et al.'s study.

Attention in the small models. The two small encoder–decoders of §4 were trained again with attention, with the same data, sizes, training steps, and decoding. Their decoder is an LSTM started from the encoder's final states; its query is its previous state and its score is additive, as in Bahdanau et al.; it reads $[E(y_{t-1}); \mathbf{c}_t]$ as its input, and its output layer is one tanh layer reading $\mathbf{s}_t$, $\mathbf{c}_t$, and $E(y_{t-1})$. Figures 16 and 17 show the two results.

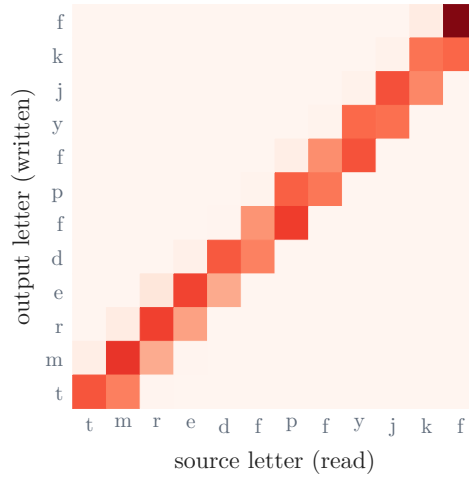


Figure 16. Attention traces the reversal. The attention weights of the reversal model for one 12-letter string: row t shows how the decoder divides its weight over the source letters (columns) when it writes letter t of the reversed string; darker is more weight.

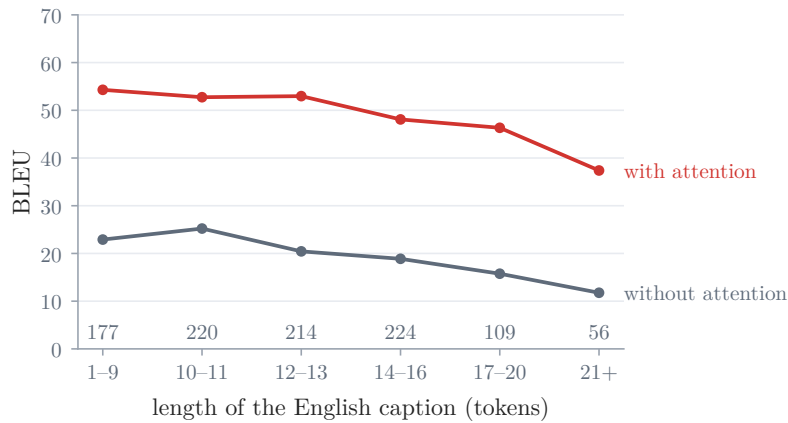


Figure 17. The small model without attention is far behind at every length. BLEU on the Multi30k test set by the length of the English caption, for the LSTM encoder-decoder of §4 and the same network with attention, both decoding greedily. Lengths count words and punctuation marks; the small numbers are the captions in each group, and the average caption has 13 tokens.

1. *Reversal.* With attention, the network writes more than 99% of the letters correctly for strings of up to 35 letters and 98% at 40, against 18% without it (Figure 12, red). The weights for one string (Figure 16) trace the anti-diagonal of a reversal: at each step the decoder divides its weight between the source position it must copy and the one just after it, whose left-to-right state has just read that letter. Nothing told the network to do this. Beyond the lengths it was trained on, it fails too, but more gradually than without attention: 77% of the letters at 50, 36% at 60.
2. *Translation.* The encoder-decoder with attention (4.4 million parameters, about 18 minutes of training) scores 49.2 BLEU on the Multi30k test captions, against 19.5 without attention. By the length of the English caption (Figure 17), both models translate long captions worse than short ones (which are also simpler, so length is not the only factor), but the model without attention is far behind at every length: 12 against 37 BLEU on the 56 captions of 21 tokens or more.

The comparison has a limit: the attention model also receives source information at every decoder step, and its output layer reads the context, so the comparison does not isolate how much of the difference comes from choosing what to read. A control that gave the fixed context

c to every step and to the output layer would separate the two; it has not been run. These are single training runs, and scores on short, lowercased captions are not comparable with scores on news text.

Reading the weights

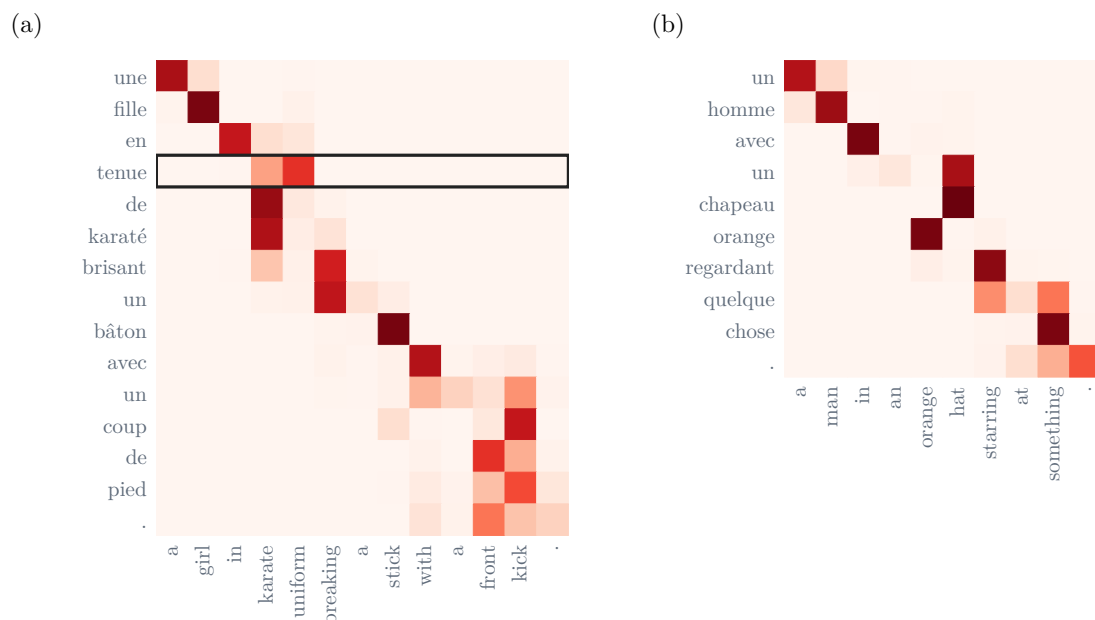


Figure 18. Attention weights follow the reordering of noun phrases. The weights of the small English-to-French translator with attention, on two Multi30k test captions. Each row is one step of the decoder, labeled with the French word it predicts; each column is an English word; the row shows how the decoder divides its weight over the English words at that step, darker for larger weights. Each row sums to one. (a) The pair of Figure 2; the row of *tenue* is outlined. (b) A second pair; *starring* is a misspelling of *staring* in the corpus.

The weights of all the reads made for one translation form a matrix, with one row for each target word and one column for each word of the source: row t holds the weights α_{tj} of the read made before predicting word t . Figure 18 shows two such matrices for the small English-to-French translator, computed with teacher forcing: the decoder is given the reference's words as its previous words, so row t shows where it reads before predicting the t -th word of the reference. Three things stand out.

1. **One English word per row, mostly.** Most rows put almost all their weight on the English word that the French word translates: *fille* on *girl* (0.96), *bâton* on *stick* (0.97).
2. **The reordering.** In these captions the French noun comes before the word that modifies it, and the weights follow. The outlined row, the step that predicts *tenue*, puts 0.66 on *uniform*, the fifth English word, and 0.33 on *karate*, the fourth; the rows of *de* and *karaté* go back to *karate*, with 0.90 and 0.84. In (b), the rows of *chapeau* and *orange* read *hat* and then *orange*, for *orange hat*.
3. **Reading ahead.** In (b), the step that predicts the second *un* puts 0.86 on *hat*, not on *an*. Plausibly, to choose between *un* and *une*, the decoder needs the gender of the noun that follows.

The weights show where the context came from, not why the decoder chose its word. Translating the caption of (a) by greedy decoding, the model writes *faisant*, “doing”, where the reference has *brisant*, “breaking” (§2). At that step it puts 0.73 of its weight on *breaking*, yet it is unsure

what to write: *faisant*, its most probable word, has a probability of only 0.10. It read the right word and still chose a poor one. The choice also depends on the decoder’s state, the previous word, and the output layer. And the encoder state it read at *breaking* carries the words around *breaking* as well as the word itself. To show that a source word caused a prediction, one has to intervene, by changing the word or forcing different weights, and see whether the prediction changes (Jain and Wallace, 2019; Wiegrefe and Pinter, 2019). Replacing *breaking* by *holding* in the source, for example, changes *faisant* to *tenant*, “holding”, and leaves the rest of the translation unchanged; this week’s notebook tries both kinds of intervention.

7. Attention variants

Between 2015 and 2017 attention appeared in many forms: additive and multiplicative, global and local, soft and hard, self-attention, multi-head attention. Each name answers a separate question about the read of §6, and a model makes one choice for each question. The first question, what is compared and what is returned, gives the vocabulary for all the others.

Queries, keys, and values

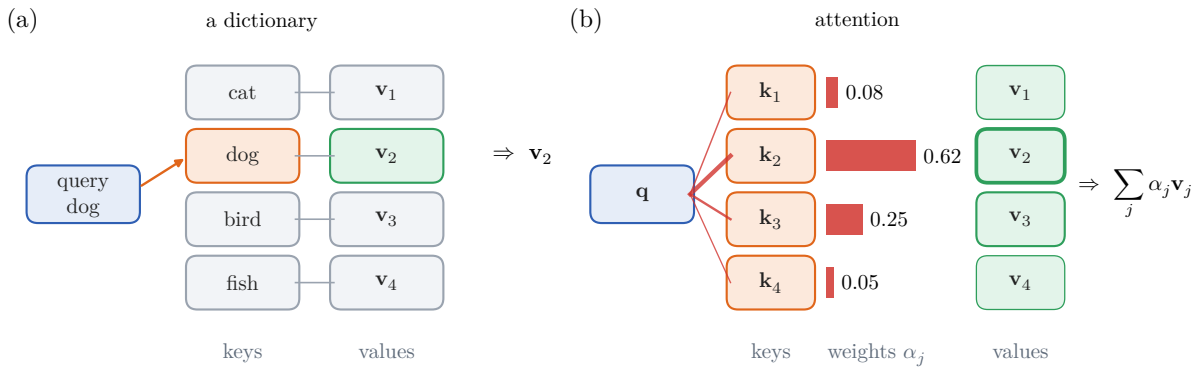


Figure 19. Attention is a lookup with weights instead of a match. (a) A dictionary returns the value whose key equals the query. (b) Attention compares the query with every key, turns the comparisons into weights that sum to one, and returns the weighted average of the values; here most of the weight goes to the second key, so the output is mostly, but not only, the second value.

A dictionary returns the value stored under the key that matches the query. Attention softens the match: it compares the query with every key, turns the comparisons into weights, and returns the weighted average of the values (Figure 19), as kernel regression averages the observed values at the inputs near a query point (Appendix A.4). With a query q , keys k_j , and values v_j ,

$$\text{read} = \sum_j \alpha_j v_j, \quad \alpha_j = \text{softmax}_j(\text{score}(q, k_j)).$$

The three roles:

- the **query** says what the reader is looking for;
- a **key** says what an item can be found by;
- a **value** says what the item returns when it is read.

In Bahdanau et al.’s attention the query is the decoder’s previous state s_{t-1} , and the key and the value of position j are the same vector, h_j . They need not be the same, since what makes a position relevant need not be what it should contribute once it is chosen. The terms *key* and

value come from networks with an external memory. The Neural Turing Machine, a recurrent network with a memory matrix that it can read and write (Appendix B.3), read its memory by emitting a “key vector”, compared with every row of the memory (Graves et al., 2014); end-to-end memory networks stored each sentence of a story twice, once to be compared with a question and once to be returned (Sukhbaatar et al., 2015); and key–value memory networks named the two parts (Miller et al., 2016). Separate keys and values reached language modeling (Daniluk et al., 2017) and translation (ConvS2S, the convolutional translator of §8), and the Transformer paper made the three roles the general description of attention, “mapping a query and a set of key-value pairs to an output”, with all three computed by learned maps (Lecture 4).

Cross or self. In translation the query comes from one sequence, the target, and the keys and values from another, the source: **cross-attention**. Nothing in the read requires two sequences. When the queries, keys, and values come from the same sequence, each position reading the others, the read is **self-attention**; it can use any of the scores below, and it need not be causal: an encoder’s positions may read positions on both sides, a decoder’s only the positions up to their own. §10 takes this step.

More than one read. One read returns one weighted average, and some predictions need more than one thing from the source. A model can read again with a query that already contains the result of the first read: memory networks read their memory in several rounds, called **hops**, so that what one read returns can help find a second fact; ConvS2S (§8) reads the source once in every decoder layer. Such repeated reads are sequential, each depending on the one before. A model can also make several independent reads at the same step, with different queries, as the read heads of the Neural Turing Machine did; the Transformer’s **multi-head attention** makes several reads of this kind in every layer (Lecture 4).

How to score

In translation the query is a decoder state $\mathbf{s}$ of d_s numbers and the keys are encoder states $\mathbf{h}_j$ of d_h numbers. Three scores were in common use:

- the **dot product**, $\mathbf{s} \mathbf{h}_j^\top$, which has no parameters and requires $d_s = d_h$;
- the **bilinear** score, $\mathbf{s} W \mathbf{h}_j^\top$, with a matrix $W \in \mathbb{R}^{d_s \times d_h}$ that learns which coordinates of the query to compare with which coordinates of the state;
- the **additive** score of §6, $\tanh(\mathbf{s} W_a + \mathbf{h}_j U_a) \mathbf{v}_a^\top$, a small network with d_a hidden units, whose $\tanh$ makes the score a nonlinear function of the query and the state.

The dot-product and bilinear scores are called **multiplicative**.

The dot product is large when the two vectors point the same way and are long. Unlike the cosine of the angle between them, it depends on their lengths, so a state with a large norm can win the softmax against a better-aligned one. The bilinear score is a dot product after a learned projection, and if W is factored as $W_Q W_K^\top$, with $W_Q \in \mathbb{R}^{d_s \times d_k}$ and $W_K \in \mathbb{R}^{d_h \times d_k}$, it becomes

$$\mathbf{s} W \mathbf{h}_j^\top = (\mathbf{s} W_Q)(\mathbf{h}_j W_K)^\top,$$

a dot product between a projected query and a projected state in d_k dimensions, with $(d_s + d_h)d_k$ parameters instead of $d_s d_h$. This is the form Lecture 4 builds on.

The costs differ in a way that later mattered. The dot and bilinear scores of many queries against many states form a single matrix product, $QH^\top$ or $QWH^\top$ with the queries and the states as the rows of Q and H , which is the operation that parallel hardware does fastest; additive scores need a $\tanh$ for every pair. Which score works best depends on the setting. Luong et al. (2015) found the dot product best for global attention and the bilinear score for

local attention (both defined in the next subsection). A larger study later found additive scores slightly better than multiplicative ones (Britz et al., 2017). The Transformer paper, citing that study, attributed the gap, tentatively, to dot products that grow large in many dimensions, and countered it by scaling them (Lecture 4).

Where and how to read

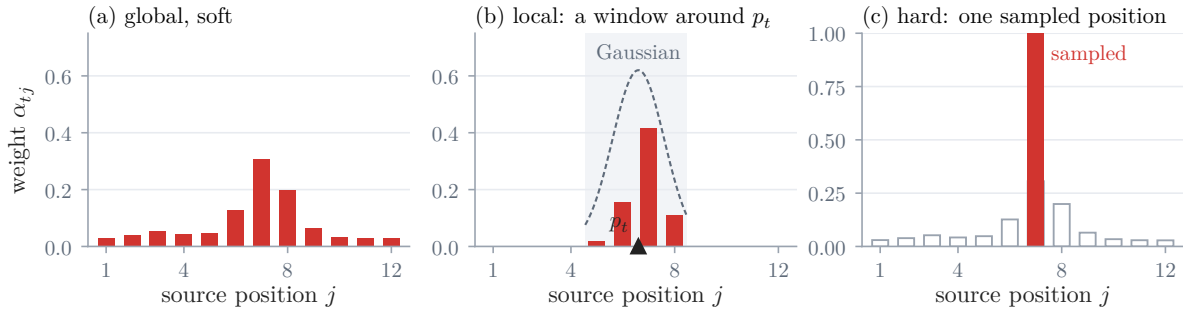


Figure 20. Three ways to read the same source. Illustrative weights of one decoder step over twelve source positions. (a) A global soft read gives some weight to every position. (b) A local read (Luong et al.’s local-p) is restricted to a window of half-width $D = 2$ around a predicted center p_t (shaded). The window holds four positions because p_t is not a whole number. Its weights are multiplied by a Gaussian (dashed), so they no longer sum to one. (c) A hard read samples one position from the soft weights (outlined) and reads that position alone.

Global or local. **Global** attention may read every source position, at a cost per step proportional to n (Figure 20a). **Local** attention reads only the positions in a window $[p_t - D, p_t + D]$ around a center p_t (Figure 20b). Luong et al. tried two centers: $p_t = t$ (*local-m*), which assumes that source and target run roughly in step, and a center predicted from the decoder’s state (*local-p*), with the weights in the window multiplied by a Gaussian bump around it (Appendix A.2). Either way the read is a differentiable weighted sum of the states in the window, so the model is trained by backpropagation as before. Local attention costs at most $2D + 1$ scores per step instead of n , but a word outside the window cannot be read at that step, which matters when word order differs a lot between the languages.

Soft or hard. The read of §6 is **soft**: it returns the weighted average of the states, a differentiable function of the scores. A **hard** read samples one position j with probability α_{tj} and returns $\mathbf{h}_j$ alone (Figure 20c). Xu et al. (2015), generating captions while attending over a grid of image regions, built both; Figure 21 shows where their model read. The soft version, they wrote, is “smooth and differentiable”, so “learning end-to-end is trivial by using standard back-propagation”. The hard version reads one actual region, but sampling has no derivative, so it needs an estimate of the gradient of the kind Lecture 6 develops for reinforcement learning (Appendix A.3). The soft version was simpler to train and became the standard. Global or local and soft or hard are independent choices: local attention is still soft, and a hard read could range over all positions.

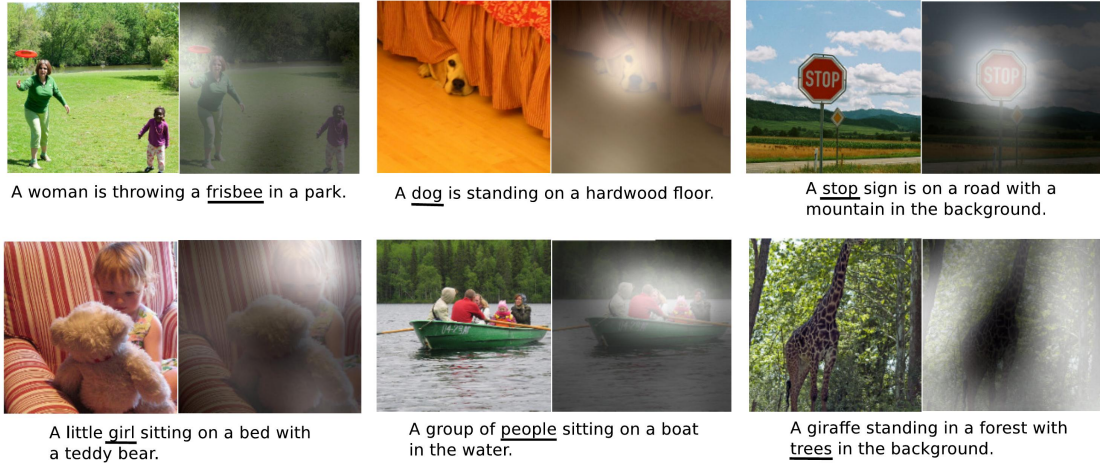
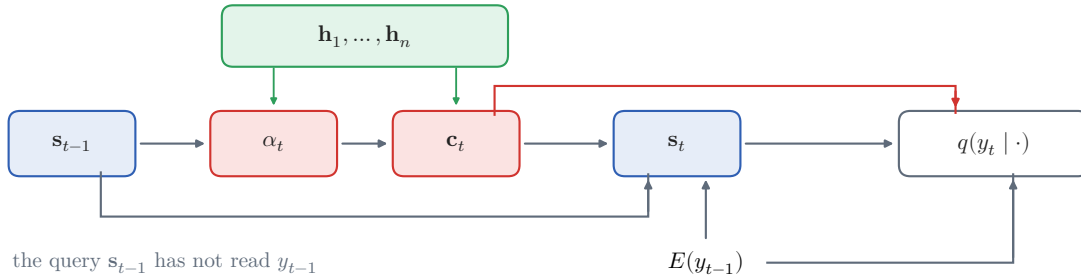


Figure 21. Attention over image regions. A caption generator that reads a grid of image regions before each word; for each photograph, the white area is where it read when it wrote the underlined word. Reproduced with attribution from the third figure of Xu et al. (2015).

When to read

(a) Bahdanau et al.: read, then update



(b) Luong et al.: update, then read

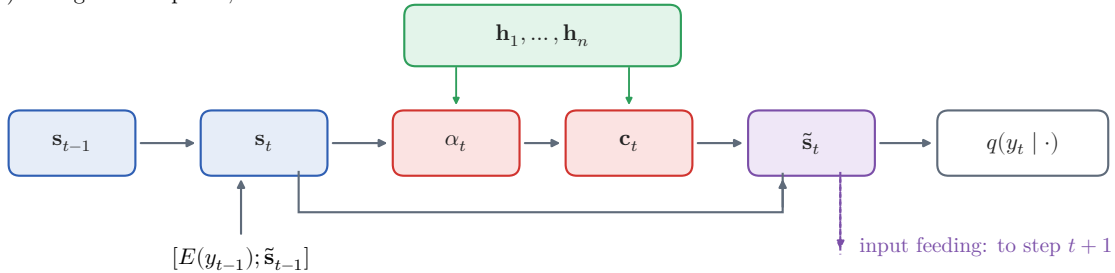


Figure 22. Two places for the read in a decoder step. (a) Bahdanau et al.: the previous state is the query, and the context enters the update and the output. (b) Luong et al.: the updated state is the query, and the context and the state are combined into the attentional state $\tilde{s}_t$, which gives the output and, with input feeding, is part of the next step's input (dashed).

The query of §6 is the decoder's previous state, and the read comes before the decoder's update. Luong et al. (2015) reversed the order (Figure 22). Their decoder first updates its state with the previous word, then reads with the new state s_t as the query, and predicts from the **attentional state** $\tilde{s}_t = \tanh([c_t; s_t]W_c)$, an extra layer that combines the read with the state. In this order the read does not enter the recurrence, so they joined $\tilde{s}_t$ to the next step's input, "to inform the model about past alignment decisions" (**input feeding**). Bahdanau et al.'s query has not read the previous word; Luong et al.'s has, but the context it reads enters the decoder's recurrent state only at the next step, through input feeding. The two orders are two placements of the same read, not two kinds of attention.

The choices together, with the papers that made them:

Choice	Alternatives	Examples
Query and memory	Different sequences (cross); the same (self)	Translation: cross. §10: self
Keys and values	The same vectors; separate	Bahdanau: the same. ConvS2S (§8): separate
Number of reads	One; repeated across depth; parallel heads	ConvS2S (§8): repeated. Transformer (§10): heads
Score	Additive; bilinear; dot product	Bahdanau: additive. Transformer (§10): scaled dot product
Eligible positions	All (global); a window (local)	Luong: both
Read	Weighted average (soft); one sampled position (hard)	Xu et al.: both, on images
Query timing	Previous decoder state; updated state	Bahdanau: previous. Luong: updated

Bahdanau et al.’s attention and the Transformer’s are both global soft reads; they differ in the score, the timing of the query, the roles of keys and values, the use of self-attention, and the number of reads. The choices can be made largely independently; the exceptions are minor, such as a plain dot product needing the query and the states to have the same size. In Bahdanau et al.’s and Luong et al.’s models, though, the encoder and the decoder are recurrent, and their states are still computed one position after another; §8 removes the recurrence.

8. Convolutional sequence models

Attention answered the first problem of this lecture: the decoder now reads every source state at every step. The second problem remains (§5). Every state of a recurrent network waits for the state before it, so even in training, when the whole target is known, a caption of 30 words takes 30 steps in each layer, one after another, and parallelism cannot help within a sentence.

Networks for images never had such a chain. A convolutional network computes the features at each pixel from a small patch of the layer below, so every position of a layer can be computed at once. Applied to a sentence, the same idea computes each state from a few neighboring positions of the layer below, instead of from the state before it. The roles of §3 stay as they are: the encoder computes states for the source, and the decoder predicts each word from the source and the target prefix. Only the way the states are computed changes. In 2016–2017 convolutional networks became the main alternative to recurrent ones for translation.

Convolution over a sequence

Take a window of width 3. A **convolution** computes the output at position i from the inputs at positions $i - 1$, i , and $i + 1$, each multiplied by a weight matrix of its own:

$$\mathbf{y}_i = \mathbf{b} + \mathbf{x}_{i-1}W_{-1} + \mathbf{x}_iW_0 + \mathbf{x}_{i+1}W_1.$$

Here $\mathbf{x}_1, \dots, \mathbf{x}_n$ are the input vectors of one layer and $\mathbf{y}_1, \dots, \mathbf{y}_n$ its outputs (not the source and the target), and zero vectors stand in for the inputs beyond either end. A window of any odd width k has one matrix W_u for each offset u :

$$\mathbf{y}_i = \mathbf{b} + \sum_{u=-(k-1)/2}^{(k-1)/2} \mathbf{x}_{i+u} W_u.$$

This **centered** window suits an encoder, which may look both ways. A decoder must not see the words it is about to predict (§3), so its convolutions use a **causal** window, which ends at the output's own position, with zero vectors in front of the first position:

$$\mathbf{y}_t = \mathbf{b} + \sum_{u=0}^{k-1} \mathbf{x}_{t-u} W_u.$$

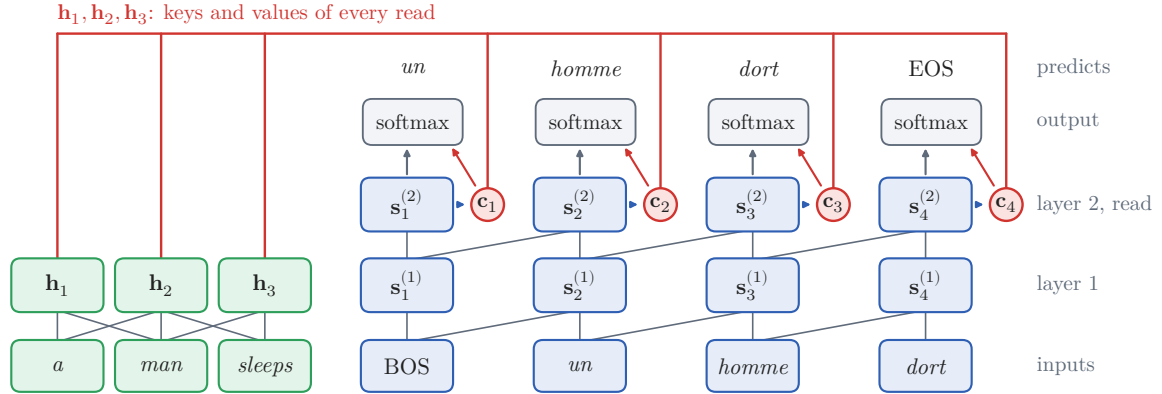
A stack of such layers, with a nonlinearity after each, is a convolutional network. Two of its properties matter here.

1. **The same weights at every position.** The matrices W_u , together called the **kernel**, are shared by all positions, as a recurrent network shares its weights across steps, so the network handles any length with a fixed set of parameters. Lecture 2's fixed-window language model was a convolution in disguise: the same network, applied to the last few words at every position.
2. **No dependence along the layer.** An output depends on k inputs of the layer below and on nothing in its own layer. When a layer's inputs are all available, all its outputs can be computed at once, so a stack of L layers takes L dependent steps however long the sequence, where a recurrent layer takes one step per position.

The idea is older than neural translation: time-delay neural networks recognized phonemes with convolutions over time (Waibel et al., 1989), Collobert et al. (2011) labeled the words of sentences with them, and the encoder of Kalchbrenner and Blunsom (2013) (§3) was convolutional.

A small convolutional translator

(a) Training: each layer computes all its positions at once



(b) Generation, step 3: only the new column is computed

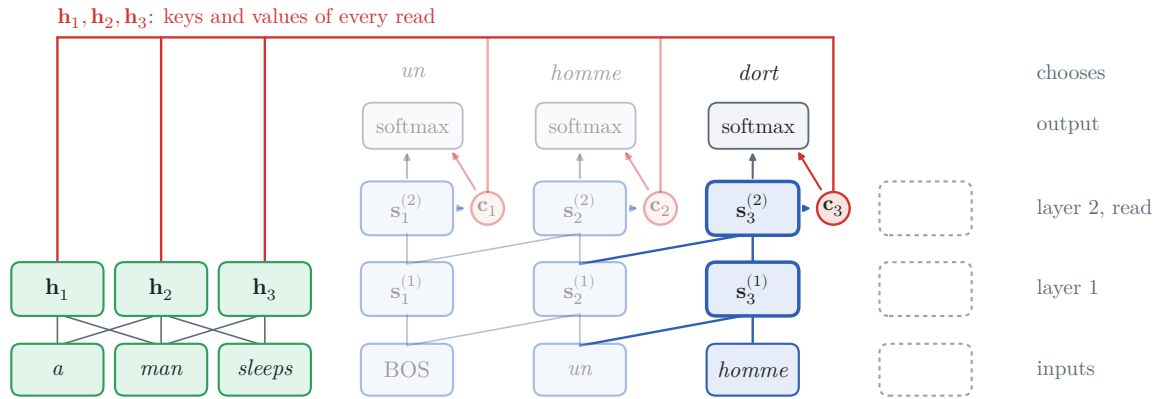


Figure 23. A convolutional encoder–decoder translating *a man sleeps*. The encoder (green) is one centered convolution of width 3 over the source words. The decoder (blue) has two causal convolutions of width 2. Each top state $s_t^{(2)}$ is the query of a read over the encoder states (red): the encoder states reach every read along the red line at the top, and the read returns c_t . The state and c_t give, through a softmax, the distribution of the word written above them. Gray lines connect each state to the states it is computed from. (a) In training, the inputs are BOS and the reference, and each layer, the reads, and the softmaxes compute all their positions at once. (b) In generation, the inputs are the words chosen so far. At step 3, the columns of steps 1 and 2 are already computed and do not change. The new column is computed from the column of step 2 and from *homme*, the word chosen at step 2, and gives the distribution from which *dort* is chosen. The column of step 4 does not exist yet.

Figure 23 follows a convolutional translator small enough to compute by hand, translating *a man sleeps* into *un homme dort*. It has the shape of the attention translator of §6, with every recurrence replaced by a convolution: the encoder computes the source states, and the decoder computes its own states, reads the source states with attention, and turns each state and its read into the distribution of the next word.

The encoder. One convolutional layer of width 3 computes a state for every source word from the word and its two neighbors: h_2 from *a*, *man*, and *slees*. No state depends on another, so the three are computed at once. These three states are all the decoder sees of the source.

The decoder in training. The decoder’s inputs are BOS and the reference, shifted right by one position (§2): BOS, *un*, *homme*, *dort*, at positions 1 to 4, so the input at position t is y_{t-1} . Each position t ends with the distribution of the word y_t , in four steps.

1. **Layer 1.** The state $\mathbf{s}_t^{(1)}$ is computed from the inputs at positions $t-1$ and t : $\mathbf{s}_3^{(1)}$ from *un* and *homme*.
2. **Layer 2.** The state $\mathbf{s}_t^{(2)}$ is computed from $\mathbf{s}_{t-1}^{(1)}$ and $\mathbf{s}_t^{(1)}$. It therefore depends on the inputs at positions $t-2$ to t , that is, on y_{t-3} , y_{t-2} , and y_{t-1} , and never on y_t itself.
3. **The read.** The state $\mathbf{s}_t^{(2)}$ is the query of an attention read over the encoder states, which serve as both the keys and the values. With the dot-product score of §7 (the encoder and decoder states of this model have the same size),

$$e_{tj} = \mathbf{s}_t^{(2)} \mathbf{h}_j^\top, \quad \alpha_{tj} = \text{softmax}_j(e_{tj}), \quad \mathbf{c}_t = \sum_{j=1}^3 \alpha_{tj} \mathbf{h}_j.$$

Each position reads the source with its own weights, because its query is its own state. The query has already read y_{t-1} , so the timing is Luong et al.'s (§7).

4. **The prediction.** The state and the read, joined into one row, give the distribution of y_t through an output matrix U , as in §3 and §6:

$$q(y_t \mid y_{<t}, \mathbf{x}) = \text{softmax}([\mathbf{s}_t^{(2)}; \mathbf{c}_t] U).$$

The training loss adds $-\log q(y_t \mid y_{<t}, \mathbf{x})$ over the four positions, for the reference words *un*, *homme*, *dort*, and EOS (§2).

Each of the four steps handles all four positions at once, since the inputs of every position are known. In training, the whole target therefore takes one dependent step per layer, plus one for the read and one for the prediction, however long it is.

The decoder in generation. Now the target words are not known in advance, and the decoder builds its columns one at a time. At step 1 it has only BOS: it computes $\mathbf{s}_1^{(1)}$, $\mathbf{s}_1^{(2)}$, and $\mathbf{c}_1$, then the distribution of the first word, and chooses *un*, the most probable word (greedy decoding; beam search would keep several hypotheses, §2). At step 2 it appends *un* to its inputs, computes the second column, and chooses *homme*; and so on, until it chooses EOS (Figure 23b). A new word never changes the earlier columns, because no position reads the positions after it, so each step computes only its new column, with one dependent operation per layer, plus the read and the prediction. The steps themselves still come one after another, as for every model, because each input is the word chosen at the step before (§2).

In this small model, $\mathbf{s}_t^{(2)}$ sees only the last three target words, and each $\mathbf{h}_j$ only three source words. How far can a convolutional network read?

How far a convolution reads

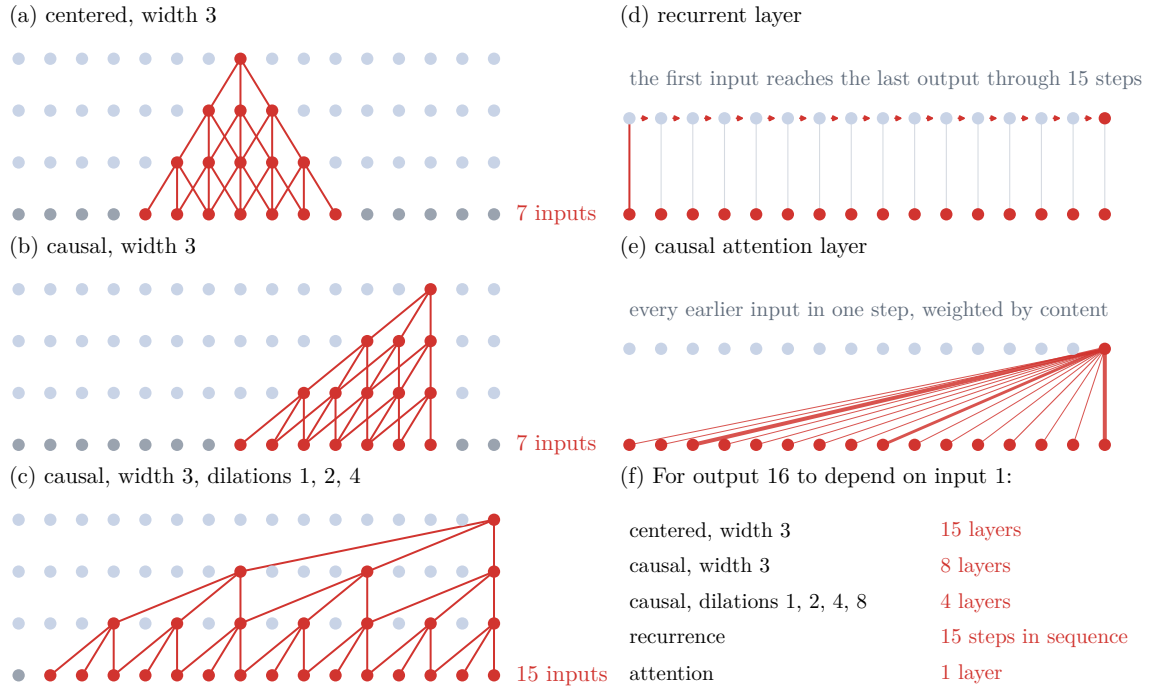


Figure 24. How far one output can read. Panels (a)–(e) show sixteen positions; the highlighted output at the top depends on the inputs marked at the bottom, through the paths drawn. (a) Three centered convolutions of width 3 see seven positions, three on each side. (b) Three causal convolutions of width 3 see seven positions, all at or before the output’s. (c) Causal convolutions of width 3 with dilations 1, 2, and 4 see fifteen positions. (d) A recurrent layer depends on every earlier position, but the path from the first position passes through every state in between. (e) A causal attention layer reads every earlier position directly, in one step, with weights computed from content. (f) What it takes for the output at position 16 to depend on the input at position 1.

The **receptive field** of an output is the set of inputs it depends on. One layer of width k sees k positions, and each further layer widens the field by $k - 1$, so L stacked layers see

$$1 + (k - 1)L$$

positions: the reach grows linearly with depth. A centered window reaches $(k - 1)L/2$ positions to either side of the output, and a causal window reaches $(k - 1)L$ positions back. With layers of width 3, making the output at position 16 depend on the input at position 1 thus takes 15 centered layers or 8 causal ones (Figure 24a, b, f).

A **dilated** convolution reaches farther with the same number of weights by spreading its window: with dilation ρ , a causal window of width 3 reads $\mathbf{x}_{i-2\rho}$, $\mathbf{x}_{i-\rho}$, and $\mathbf{x}_i$. Doubling the dilation from layer to layer, $\rho = 1, 2, 4, \dots, 2^{L-1}$, makes L layers see

$$1 + (k - 1)(2^L - 1)$$

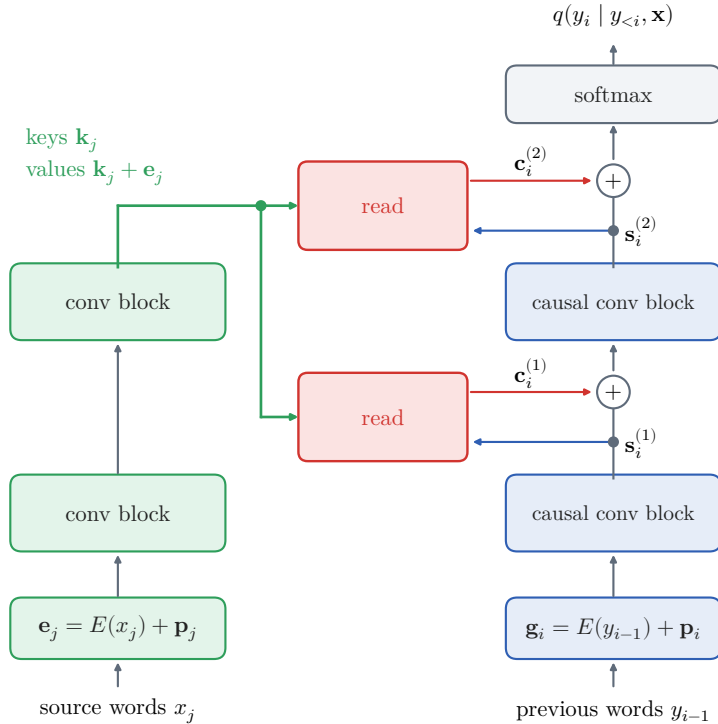
positions (Figure 24c). The field then roughly doubles with each layer, so connecting positions δ apart takes about $\log_2 \delta$ layers: ten causal layers of width 3 see 21 positions undilated and 2,047 dilated.

Either way, reaching farther costs layers: a dependency between two distant positions passes through as many layers as it takes to span the distance. A recurrent layer reaches every earlier position, but along a chain of one step per position (Figure 24d); an attention layer reaches every position in one step (Figure 24e).

Two convolutional translators

ByteNet: convolution only. Kalchbrenner et al. (2016) translated character by character with stacks of dilated convolutions and nothing else: no recurrence and no attention. The encoder’s convolutions are centered and the decoder’s causal, and the decoder is stacked on top of the encoder, position by position: at target position t it receives the encoder’s output at position t , whose receptive field covers a stretch of the source around it.⁵ The decoder’s access to the source is therefore positional, not chosen by content. On WMT 2014 English-to-German, character to character, ByteNet scored 23.75 BLEU.⁶ Its dilated causal convolutions came from WaveNet, a model of raw audio from the same group (van den Oord et al., 2016).

(a) The model, two blocks in each stack



(b) One causal block

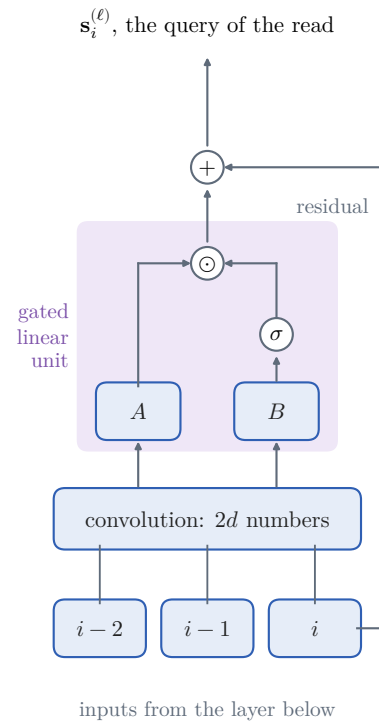


Figure 25. ConvS2S: convolutional stacks, and a read of the source in every decoder layer.

(a) The model with two blocks in each stack (L in general). The encoder (green) adds a position vector to each source word’s vector, giving e_j , and passes the result through its blocks; the top block’s outputs k_j are the keys of every read, and $k_j + e_j$ the values. The decoder (blue) forms g_i from the previous word in the same way and passes it through causal blocks. Each decoder block’s output $s_i^{(\ell)}$ is the query of a read of the encoder (red; the query also adds g_i , not drawn). The read $c_i^{(\ell)}$ is added to $s_i^{(\ell)}$, and the sum goes to the next layer; the top layer gives the distribution of the next word. (b) One causal block: a convolution over the inputs from the layer below at positions $i-2$, $i-1$, and i produces $2d$ numbers, split into A and B ; the gated linear unit returns $A \odot \sigma(B)$, and the residual connection adds the block’s input at position i .

ConvS2S: convolution with attention in every layer. Gehring et al. (2017) kept the design of the small translator above, an encoder and a decoder built from convolutions with a read between them, and changed three things (Figure 25).

⁵Since the target may be longer than the source, the encoder’s representation is extended to an estimated target length, 1.2 times the source’s for English to German, and the decoder continues until it writes EOS (*dynamic unfolding*). The translation model stacks 30 residual blocks in each network, with kernels of width 3 and dilations doubling from 1 to 16, then starting again.

⁶These are the numbers in the tables of the paper’s second version, of March 2017, which also give 26.26 on the 2015 test set; the ConvS2S paper cites this version. The text of the same version gives 22.85 and 25.53, and the first version, of October 2016, reported 18.9 and 21.7.

1. **Gated blocks.** Each block is a convolution of width 3 that produces $2d$ numbers per position, split into halves A and B . A **gated linear unit** (Dauphin et al., 2017) returns $A \odot \sigma(B)$, d numbers in which the gate $\sigma(B)$ decides, number by number, how much of A passes, and a residual connection adds the block’s input (Figure 25b). The decoder’s blocks are causal.⁷
2. **Position vectors.** A learned vector for each absolute position is added to each word’s vector: $\mathbf{e}_j = E(x_j) + \mathbf{p}_j$ in the encoder and $\mathbf{g}_i = E(y_{i-1}) + \mathbf{p}_i$ in the decoder. The convolutions already see relative order through their offsets, and in the authors’ English–German ablation, removing the position vectors cost only 0.5 BLEU.
3. **A read in every decoder layer, with separate keys and values.** Let $\mathbf{s}_i^{(\ell)}$ be the output of decoder block ℓ at target position i , and $\mathbf{k}_j$ the encoder’s top output at source position j . Each decoder layer forms its own query from its block’s output and the previous word, scores it against the encoder outputs, and reads the encoder outputs plus the input vectors:

$$\mathbf{q}_i^{(\ell)} = \mathbf{s}_i^{(\ell)} W_d^{(\ell)} + \mathbf{b}_d^{(\ell)} + \mathbf{g}_i, \quad \alpha_{ij}^{(\ell)} = \text{softmax}_j(\mathbf{q}_i^{(\ell)} \mathbf{k}_j^\top), \quad \mathbf{c}_i^{(\ell)} = \sum_j \alpha_{ij}^{(\ell)} (\mathbf{k}_j + \mathbf{e}_j).$$

The read is added to $\mathbf{s}_i^{(\ell)}$, and the sum goes to the next layer; the top layer gives the distribution of the next word through a softmax.⁸

The separate values let a read return both what the encoder computed about a position and which word stood there; the authors noted that the construction “resembles key-value memory networks”. And since a layer’s query contains what the layers below it read, the decoder reads the source several times, each read informed by the ones before, like the hops of memory networks (§7).

It worked. On WMT 2014, single ConvS2S models scored 25.16 BLEU on English-to-German and 40.51 on English-to-French (averaged over several training runs), above the single models of Google’s recurrent system (24.61 and 38.95), and the authors reported translation “at an order of magnitude faster speed, both on GPU and CPU”.⁹

Together, ByteNet and ConvS2S separate two choices that the recurrent translators made together: how each sequence is processed, and how the decoder reaches the source.

Model	Within each sequence	Decoder’s access to the source
Sutskever et al. (2014)	Recurrence	The encoder’s final state
Bahdanau et al. (2015)	Recurrence	Attention, once per step
ByteNet (2016)	Dilated convolution	The encoder’s output at the same position
ConvS2S (2017)	Convolution	Attention in every decoder layer
Transformer (2017)	Self-attention	Attention in every decoder layer

Convolution showed that translation does not need recurrence: ByteNet and ConvS2S computed every layer at all positions at once in training, and ConvS2S translated better than the recurrent systems it was compared with. It left two limits.

⁷The main models use width 3. The decoder’s convolutions are made causal by padding and trimming: “we pad the input by $k - 1$ elements on both the left and right side by zero vectors, and then remove k elements from the end of the convolution output”.

⁸The paper writes $\mathbf{h}_i^{(\ell)}$, $\mathbf{d}_i^{(\ell)}$, and $\mathbf{z}_j^u$ for $\mathbf{s}_i^{(\ell)}$, $\mathbf{q}_i^{(\ell)}$, and $\mathbf{k}_j$, with column vectors.

⁹Decoding greedily on one K40 GPU, ConvS2S was 9.3 times as fast as the recurrent system’s published speed on a K80, which the authors describe as essentially two K40s; on processors it was 9.3 times as fast with 48 cores against the recurrent system’s 88.

1. **Distance costs layers.** Distant positions of a sequence communicate only through a stack of layers (Figure 24). In the words of the Transformer paper, this growing path “makes it more difficult to learn dependencies between distant positions”.
2. **Offsets, not content.** Within a sequence, what one position contributes to another depends on their offset (W_u multiplies whatever stands u positions away), not on what either contains.

The Transformer, posted five weeks after ConvS2S, kept the parallel computation and let the positions of a sequence communicate by attention instead of through windows (§§9–10).

9. Translation from attention alone

Attention gave the decoder direct access to every source position, and convolution showed that a sequence model need not compute one position at a time. Can attention alone, with no recurrence and no convolution, translate? A toy model answers on a two-word phrase (Figure 26), and what it lacks is what the Transformer adds.

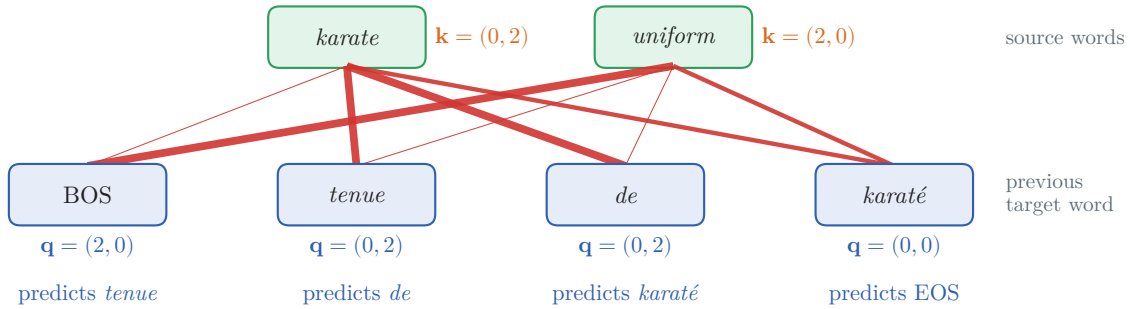


Figure 26. A translator built from attention alone. The toy of §9 on *karate uniform*. Each target position (bottom) reads the source words (top) with a query computed from the previous target word alone; the width of each line is the weight. No line joins two target positions or two source words, so in training, with the reference prefix given, the four reads are independent.

The model

The toy has four tables of vectors and an output layer, and nothing else. Each source word x_j has a key $\mathbf{k}_j = E_K(x_j)$ and a value $\mathbf{v}_j = E_V(x_j)$; each target word has a query vector, from a table E_Q , and a word vector, from a table E . At target position t the decoder

1. takes its query from the previous target word, $\mathbf{q}_t = E_Q(y_{t-1})$;
2. reads the source with a dot-product score: $\alpha_{tj} = \text{softmax}_j(\mathbf{q}_t \mathbf{k}_j^\top)$ and $\mathbf{c}_t = \sum_j \alpha_{tj} \mathbf{v}_j$;
3. predicts from the read and the previous word: $q(y_t | y_{<t}, \mathbf{x}) = \text{softmax}([E(y_{t-1}); \mathbf{c}_t] U)$.

There is no encoder computation, no decoder state, and no connection between target positions: each position reads the source by itself.

One translation by hand

Take *karate uniform*, whose translation is *tenue de karaté*, and two-dimensional keys and queries chosen by hand. The first coordinate marks a noun, the second a word that modifies a noun: $\mathbf{k} = (0, 2)$ for *karate* and $(2, 0)$ for *uniform*. The query of BOS asks for the noun, which comes first in French; the queries of *tenue* and *de* ask for the modifier; the query of *karaté* asks for nothing in particular. Scores and weights are given in the order (*karate*, *uniform*):

Position	Previous word	Query	Scores	Weights	Reads	Predicts
1	BOS	(2, 0)	(0, 4)	(0.02, 0.98)	<i>uniform</i>	<i>tenue</i>
2	<i>tenue</i>	(0, 2)	(4, 0)	(0.98, 0.02)	<i>karate</i>	<i>de</i>
3	<i>de</i>	(0, 2)	(4, 0)	(0.98, 0.02)	<i>karate</i>	<i>karaté</i>
4	<i>karaté</i>	(0, 0)	(0, 0)	(0.5, 0.5)	both	EOS

The first read goes to *uniform*, the second word of the source, because the query of BOS matches the key of a noun: the reordering comes from content, not from position. The output layer, not written out here, does the rest. Think of it as a table that maps the previous word and the word read to the next French word: BOS and *uniform* give *tenue*; *tenue* and *karate* give *de*; *de* and *karate* give *karaté*; *karaté* gives EOS, whatever was read.

In training, teacher forcing supplies every previous word in advance, so every query is known before anything is computed, and the reads of all target positions are a few matrix products. With the queries, keys, and values as the rows of Q (four rows), K , and V (two rows each),

$$S = QK^\top = \begin{pmatrix} 0 & 4 \\ 4 & 0 \\ 4 & 0 \\ 0 & 0 \end{pmatrix}, \quad A = \text{softmax}(S) = \begin{pmatrix} 0.02 & 0.98 \\ 0.98 & 0.02 \\ 0.98 & 0.02 \\ 0.5 & 0.5 \end{pmatrix}, \quad C = AV,$$

with the softmax taken along each row. Row t of C is $\mathbf{c}_t$: almost the value of *uniform* in row 1, almost the value of *karate* in rows 2 and 3, and the average of the two in row 4. No position waits for another, so a whole target takes a fixed number of dependent operations, however long it is, where the recurrent decoder of §5 needed one step per position.

What the toy cannot do

The toy translates its one phrase, but three kinds of information are missing from it:

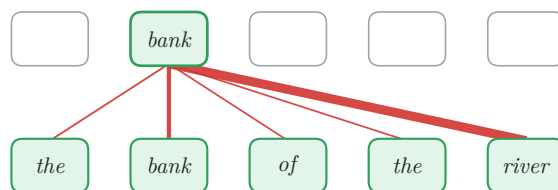
1. **Word order.** A read depends only on the unordered collection of source (key, value) pairs (§6), and here they come from the words alone. *Dog bites man* and *man bites dog* give the same reads, and the same translation. A recurrent or convolutional encoder put order into its states; without one, information about each word's **position** has to be added to its vectors.
2. **The context of each source word.** *Bank* has the same key and value in *the bank of the river* and in *the bank raised its rates*, and *kick* the same in *front kick* and *free kick*. The source words need representations computed from the words around them, which the encoder can supply by letting the source positions read one another: **self-attention** in the encoder.
3. **The target so far.** Each query sees only the previous word. The full translation of Figure 2 contains *un* twice, in *un bâton* and *un coup de pied*. After each, the toy has the same query, makes the same read, and predicts the same word: it cannot know what it has already written. Each target position needs to read the earlier target positions: **causal self-attention** in the decoder.

A single read per position is also limiting. Several reads at once, and layers that read what the reads below produced, let a model combine information, as ConvS2S's repeated reads did (§8). Adding these pieces to the toy, while keeping its property that no position waits for another in training, gives the Transformer.

10. Self-attention and the Transformer

The toy of §9 lacks positions, the context of each source word, and the target so far. The last two come from one operation: attention turned on a single sequence, each position reading the others (Figure 27).

(a) encoder self-attention: every position may read every position



(b) causal decoder self-attention: a position reads itself and the positions before it

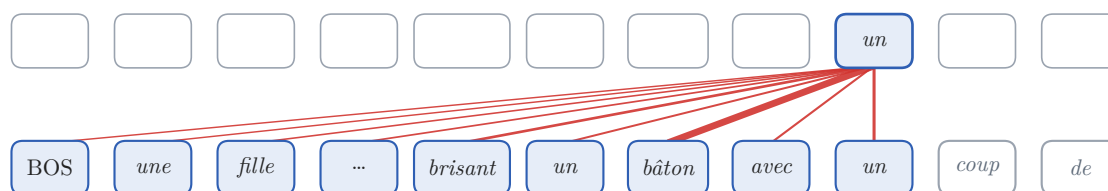


Figure 27. Self-attention: the positions of one sequence read one another. Each output position (top) reads the positions of the layer below (bottom); one output's read is drawn, with line width for the weight (illustrative weights). (a) In an encoder every position may read every position: *bank* reads *river*, which tells it which bank is meant. (b) In a decoder the reads are causal: the second *un* may read itself and the positions before it, including *bâton*, which tells it that the stick has been written; the later positions (gray) do not exist yet in generation and are hidden in training.

Self-attention

Let every position i of a sequence emit a query $\mathbf{q}_i$, and every position j offer a key $\mathbf{k}_j$ and a value $\mathbf{v}_j$. Position i 's output is the read

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{v}_j, \quad \alpha_{ij} = \text{softmax}_j(\text{score}(\mathbf{q}_i, \mathbf{k}_j)),$$

over all positions j in an encoder, and over $j \leq i$ in a causal decoder. This is **self-attention**: in the Transformer paper's words, "an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence". In an encoder, *bank* can read *river* in one sentence and *rates* in another, and gets a different output in each. In a decoder, the position whose input is the second *un* of the karate translation has *bâton* among its earlier positions, and the position of the first *un* does not, so their outputs differ: the decoder can know that the stick has been written.

A self-attention layer combines the strengths of convolution and of cross-attention (Figure 28):

1. **Like a convolutional layer**, it computes all its outputs at once when its inputs are available (for a source, and in training for a target), since each output depends on the layer below and not on the other outputs.
2. **Like the cross-attention of §6**, it lets any position read any other in one step, with weights computed from content, not fixed by an offset. In *the girl broke the stick with a kick, and it snapped*, the position of *it* can read *stick* directly, at whatever distance, if its query matches the key of *stick*.

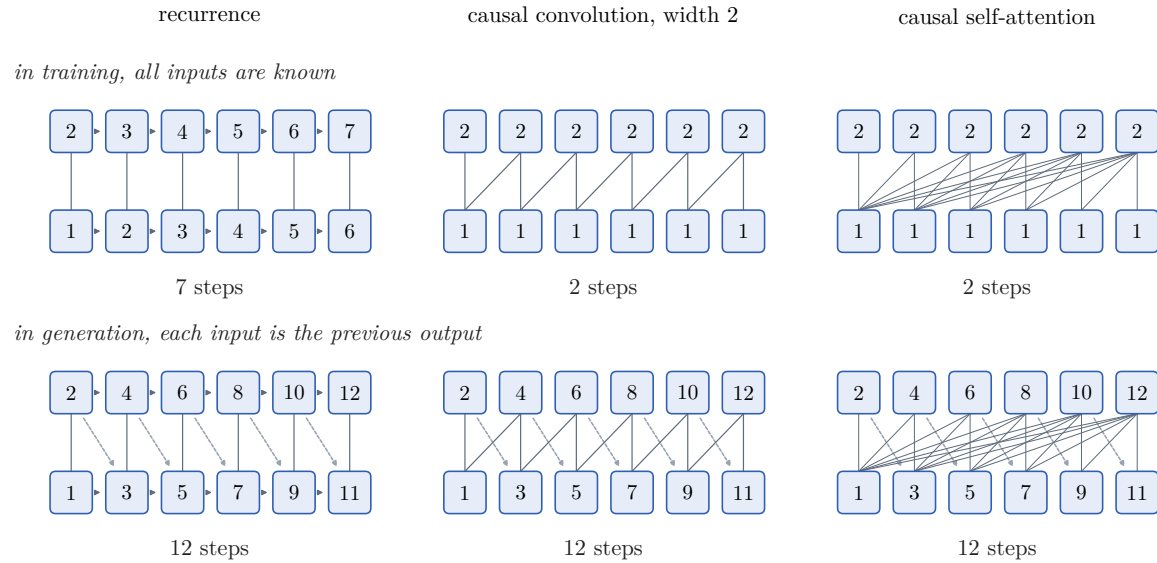


Figure 28. Recurrence, convolution, and self-attention, in training and in generation. Two layers over six positions; the number in each state is the earliest step at which it can be computed with one processor per state. Lines join each state to the states of the layer below that it is computed from: in the convolution, the state at its position and the one before; in self-attention, every state at or before its position. In training (top), all inputs are known: a recurrent stack needs one step per position along its chains, plus one per further layer (seven here), while convolutional and self-attention layers compute all positions at once, two steps for two layers. In generation (bottom), each input is the token chosen at the position before, so all three produce one position at a time.

The table sets the three mechanisms side by side, for a layer over n positions with vectors of d numbers:

	Recurrence	Convolution, width k	Self-attention
Dependent steps per layer, training	n	1	1
Dependent steps to generate $r = m + 1$ outputs, L layers	rL	rL	rL
Steps or layers to connect positions δ apart	δ steps	$\lceil \delta / (k - 1) \rceil$ causal layers	1 layer
What decides which positions communicate	Order in the chain	Offset in the window	Query–key scores
Multiplications per layer, roughly	nd^2	knd^2	n^2d , plus projections

The convolution column assumes undilated causal layers. The last row is the price of self-attention: its pairwise work grows with the square of the length.

Self-attention appeared before the Transformer. [Cheng et al. \(2016\)](#) gave an LSTM attention over its own earlier states, which they called **intra-attention**; [Paulus et al. \(2017\)](#) let a summarizer’s decoder attend to what it had already written, to avoid repeating itself; and [Parikh et al. \(2016\)](#) compared two sentences with attention and small networks applied to each word, without recurrence, and, with a ninth of the parameters, matched the accuracy of an LSTM-based model ([Appendix B.3](#)).

The original Transformer

[Vaswani et al. \(2017\)](#) kept the encoder–decoder organization of this lecture and built both parts from attention, with no recurrence and no convolution ([Figure 29](#)).

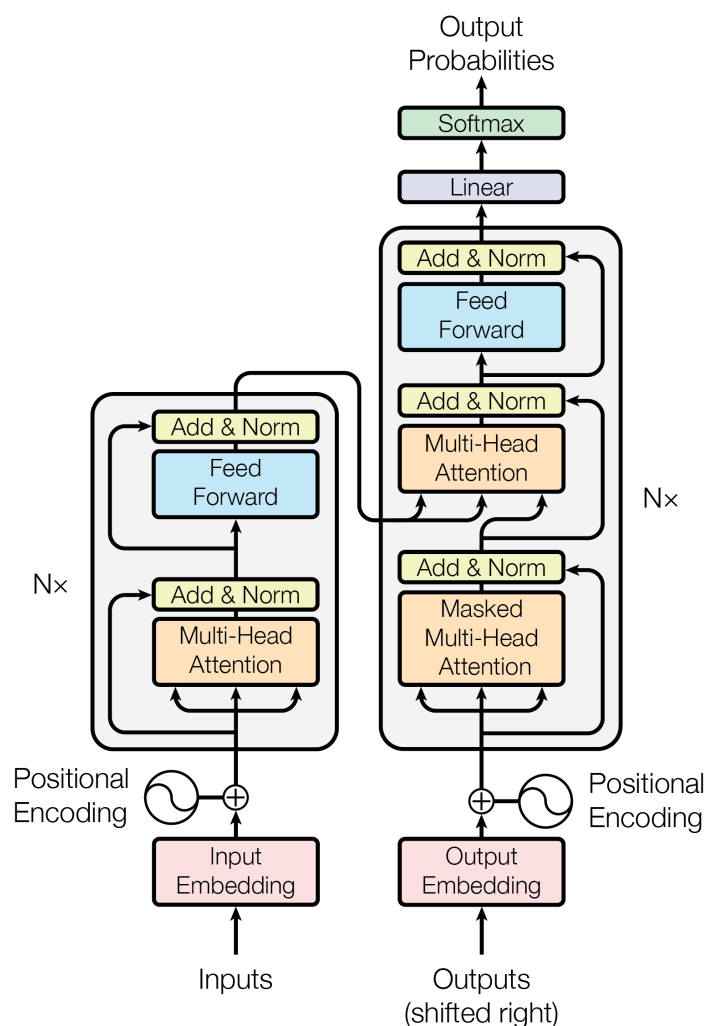


Figure 29. The Transformer, as drawn by Vaswani et al. (2017, their Fig. 1). Reproduced with attribution from *Attention Is All You Need* (arXiv:1706.03762v7). The encoder (left) and the decoder (right) are each a stack of $N = 6$ identical layers. The three uses of attention of this section are the encoder’s multi-head self-attention, the decoder’s masked multi-head self-attention, and the decoder’s multi-head attention over the encoder’s output, whose keys and values come from the encoder (its two left-hand inputs) and whose queries come from the masked self-attention sublayer below (its right-hand input). The embeddings, positional encodings, feed-forward sublayers, and residual connections with layer normalization (“Add & Norm”) are built in Lecture 4.

Attention appears in three places: the toy’s cross-attention, and the two self-attentions it lacked.

1. **Encoder self-attention:** each source position reads all source positions, so each source word is represented in its context.
2. **Causal decoder self-attention:** each target position reads those up to its own, so the decoder knows what it has written.
3. **Cross-attention:** each target position reads the encoder’s output, as the toy and the decoder of §6 did.

Order enters through **positional encodings**, vectors that depend on the position and are added to the word vectors. Each attention sublayer (the attention step inside a layer) makes several reads at once, one per **head**, and the encoder and the decoder are each a stack of six

layers. The score is a dot product of a query and a key, divided by a constant. Whenever all the queries of a layer are available, the scores of all its positions are computed at once:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

where the rows of Q , K , and V are the queries, keys, and values of all positions, d_k is the number of coordinates of a query or key, and the softmax runs along each row. Each row of the result is one position's read: the toy's $C = AV$, with the scores divided by $\sqrt{d_k}$.

The design removes the recurrent dependencies, so that in training all positions of a layer are computed at once, and it gives every pair of permitted positions a direct path. It does not remove every constraint: generation is still sequential (§2), and attention over all positions computes a score for every pair of them (the quadratic cost above). Lecture 4 builds what the formula leaves open: the learned maps that produce the queries, keys, and values; why the dot products are divided by $\sqrt{d_k}$; how masks enforce causality in one matrix product; why there are several heads; how the order of the positions enters; and what else a block needs for a deep stack of them to train.

11. Summary

- **Sequence tasks** classify, label, forecast, detect anomalies, or map one sequence to another, over words, numbers, vectors, or fields. Translation, the transduction these notes concentrate on, combines the main difficulties: lengths differ, order changes, and words correspond to phrases or to nothing.
- **Translation is conditional language modeling.** Each target token is predicted from the source and the target before it, ending with EOS; training uses maximum likelihood with teacher forcing; a search, greedy or beam, produces the translation; log loss and BLEU measure different things, and BLEU, the measure of the period, has since given way to learned metrics. Teacher forcing makes all target inputs available in advance, and the architecture decides whether their states can be computed in parallel; generation proceeds one token at a time.
- **Encoder and decoder are roles:** the encoder maps the source to states, the decoder maps the states and the target prefix to its own state, and that state to a distribution over the next token. The recurrent cell changes what a state holds, not the roles. In the recurrent encoder–decoder, the source reaches the decoder through one context of fixed size, given at the start or at every step.
- **Sequence to sequence** turned a deep LSTM encoder–decoder that reads the source backwards into a complete translator, competitive with phrase-based systems. Three drawbacks limit such designs: a fixed-size summary of the source, sequential computation, and long paths.
- **Attention** gives the decoder a fresh, learned read of the encoder states at each step, trained by the loss of the next word alone: the correspondences between words emerge without alignment labels. It frees the decoder from a single fixed summary and shortens the paths from source to target; the computation stays recurrent. The read sees the encoder states as a set, and its weights show where the context came from, not what caused a prediction. Its variants are largely independent choices: cross or self, keys and values, number of reads, score, eligible positions, soft or hard reading, and query timing.
- **Convolution** computes all positions of a layer at once when its inputs are available, with a reach that grows with depth: a convolutional decoder computes every target position at

once in training and one new column per step in generation. ByteNet and ConvS2S show that within-sequence computation and access to the source are separate choices.

- **Translation from attention alone** computes all target positions at once in training, and reorders by content, but it cannot see word order, the context of a source word, or what it has already written.
- **Self-attention** gives every permitted pair of positions a direct, content-dependent read. The Transformer uses it in its encoder and, causally, in its decoder, with cross-attention between them and positions added to the word vectors; dense attention's pairwise work grows with the square of the length.

Background reading

The basics of this lecture are developed in full elsewhere, in the chapters of the [Background Notes on Machine Learning and AI](#), standard textbooks, and courses and videos:

Topic	Background Notes	Textbook, course, or video
Recurrent networks and sequence models	DL 8	Goodfellow, Bengio, and Courville , Chapter 10
Machine translation, statistical and neural	NLP & LLMs 15	Jurafsky and Martin , <i>Machine Translation</i> ; Hutchins , <i>Machine translation over fifty years</i>
Encoder-decoders and attention	NLP & LLMs 15 , DL 9	Voita , <i>Sequence to Sequence (seq2seq) and Attention</i> ; Olah and Carter , <i>Attention and Augmented Recurrent Neural Networks</i>
Convolutional networks		Goodfellow, Bengio, and Courville , Chapter 9; Prince , <i>Understanding Deep Learning</i> , Chapter 10

For reading in the order of this lecture: [Sutskever et al. \(2014\)](#) and [Bahdanau et al. \(2015\)](#) are short, and read together show what attention changed; the appendix of Bahdanau et al. gives their model in full. Section 3 of [Luong et al. \(2015\)](#) lays out the choices of §7 in four pages. [Olah and Carter \(2016\)](#) illustrate attention and memory with interactive figures. Section 3 of [Gehring et al. \(2017\)](#) describes a convolutional translation model with attention in every layer, and Table 1 of [Vaswani et al. \(2017\)](#) compares recurrence, convolution, and self-attention as §10 does; Lecture 4 covers the rest of the paper.

References and further reading

Sequence modeling and transduction

- Ronald J. Williams and David Zipser (1989), A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. [Paper](#). *Neural Computation* 1(2). Teacher forcing.
- Alex Graves (2012), Sequence Transduction with Recurrent Neural Networks. [Paper](#). The RNN transducer.
- Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber (2006), Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks. [Paper](#).
- Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu (2002), BLEU: a Method for Automatic Evaluation of Machine Translation. [Paper](#).
- Ricardo Rei, Craig Stewart, Ana C Farinha, and Alon Lavie (2020), COMET: A Neural Framework for MT Evaluation. [Paper](#).
- Markus Freitag et al. (2022), Results of WMT22 Metrics Shared Task: Stop Using BLEU – Neural Metrics Are Better and More Robust. [Paper](#).
- Tom Kocmi et al. (2024), Preliminary WMT24 Ranking of General MT Systems and LLMs. [Paper](#).
- Yonghui Wu et al. (2016), Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. [Paper](#). GNMT; the length penalty of beam search.
- Philipp Koehn and Rebecca Knowles (2017), Six Challenges for Neural Machine Translation. [Paper](#). Wider beams find shorter, worse translations.
- Desmond Elliott, Stella Frank, Khalil Sima’an, and Lucia Specia (2016), Multi30K: Multilingual English-German Image Descriptions. [Paper](#). The French translations were added for the shared task of 2017: Desmond Elliott et al. (2017), Findings of the Second Shared Task on Multimodal Machine Translation and Multilingual Image Description. [Paper](#).

Translation before neural networks

- Warren Weaver (1949), Translation. [Memorandum](#), reprinted in Locke and Booth (eds.), *Machine Translation of Languages* (1955).
- W. John Hutchins (2004), The Georgetown-IBM Experiment Demonstrated in January 1954. [Paper](#).
- Automatic Language Processing Advisory Committee (1966), Language and Machines: Computers in Translation and Linguistics. [Report](#). The ALPAC report.
- Peter F. Brown et al. (1990), A Statistical Approach to Machine Translation. [Paper](#).
- Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer (1993), The Mathematics of Statistical Machine Translation: Parameter Estimation. [Paper](#). The IBM models.
- Philipp Koehn, Franz Josef Och, and Daniel Marcu (2003), Statistical Phrase-Based Translation. [Paper](#).

- Philipp Koehn et al. (2007), Moses: Open Source Toolkit for Statistical Machine Translation. [Paper](#).
- Franz Och (2006), Statistical machine translation live. [Post](#), Google Research blog.
- W. John Hutchins (2001), Machine translation over fifty years. [Paper](#). *Histoire, Épistémologie, Langage* 22(1).

Recurrent encoder–decoders

- Mike Schuster and Kuldeep K. Paliwal (1997), Bidirectional Recurrent Neural Networks. [Paper](#). *IEEE Transactions on Signal Processing* 45(11).
- Nal Kalchbrenner and Phil Blunsom (2013), Recurrent Continuous Translation Models. [Paper](#).
- Kyunghyun Cho et al. (2014a), Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. [Paper](#). The arXiv version (1406.1078) has the full model in an appendix.
- Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton (2013), Speech Recognition with Deep Recurrent Neural Networks. [Paper](#). Deep LSTMs.
- Ilya Sutskever, Oriol Vinyals, and Quoc V. Le (2014), Sequence to Sequence Learning with Neural Networks. [Paper](#).
- Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio (2014b), On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. [Paper](#). The decline with sentence length.
- Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio (2014), Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. [Paper](#). LSTM against GRU.
- Rafal Jozefowicz et al. (2016), Exploring the Limits of Language Modeling. [Paper](#).

Attention, 2014–2017

- Alex Graves (2013), Generating Sequences With Recurrent Neural Networks. [Paper](#). A soft window over the text being written.
- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio (2015), Neural Machine Translation by Jointly Learning to Align and Translate. [Paper](#). On arXiv in September 2014, at ICLR in 2015; the model in full is in its appendix.
- Andrej Karpathy (2024), The (true) story of development and inspiration behind the “attention” operator. [Post](#), with Bahdanau’s account.
- Minh-Thang Luong, Hieu Pham, and Christopher D. Manning (2015), Effective Approaches to Attention-based Neural Machine Translation. [Paper](#). Scores, the timing of the query, local attention, input feeding. The arXiv version (1508.04025) gives the window size and writes the additive score in full.
- Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le (2017), Massive Exploration of Neural Machine Translation Architectures. [Paper](#). Additive against multiplicative attention.
- Kelvin Xu et al. (2015), Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. [Paper](#). Soft and hard attention.

- Ronald J. Williams (1992), Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. [Paper](#). *Machine Learning* 8. REINFORCE.
- Sarthak Jain and Byron C. Wallace (2019), Attention is not Explanation. [Paper](#).
- Sarah Wiegreffe and Yuval Pinter (2019), Attention is not not Explanation. [Paper](#).
- Alex Graves, Greg Wayne, and Ivo Danihelka (2014), Neural Turing Machines. [Paper](#).
- Jason Weston, Sumit Chopra, and Antoine Bordes (2015), Memory Networks. [Paper](#).
- Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus (2015), End-To-End Memory Networks. [Paper](#).
- Alexander Miller et al. (2016), Key-Value Memory Networks for Directly Reading Documents. [Paper](#).
- Michał Daniluk, Tim Rocktäschel, Johannes Welbl, and Sebastian Riedel (2017), Frustratingly Short Attention Spans in Neural Language Modeling. [Paper](#). Separate keys, values, and outputs for prediction.
- Jan Chorowski et al. (2015), Attention-Based Models for Speech Recognition. [Paper](#).
- William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals (2016), Listen, Attend and Spell. [Paper](#).
- Oriol Vinyals et al. (2015a), Grammar as a Foreign Language. [Paper](#).
- Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly (2015b), Pointer Networks. [Paper](#).
- Jianpeng Cheng, Li Dong, and Mirella Lapata (2016), Long Short-Term Memory-Networks for Machine Reading. [Paper](#). Intra-attention.
- Ankur P. Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit (2016), A Decomposable Attention Model for Natural Language Inference. [Paper](#).
- Zhouhan Lin et al. (2017), A Structured Self-attentive Sentence Embedding. [Paper](#).
- Romain Paulus, Caiming Xiong, and Richard Socher (2017), A Deep Reinforced Model for Abstractive Summarization. [Paper](#).

Convolutional sequence models

- Alexander Waibel et al. (1989), **Phoneme Recognition Using Time-Delay Neural Networks**. [Paper](#). *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37(3).
- Ronan Collobert et al. (2011), **Natural Language Processing (Almost) from Scratch**. [Paper](#).
- Aäron van den Oord et al. (2016), **WaveNet: A Generative Model for Raw Audio**. [Paper](#). Dilated causal convolutions.
- Nal Kalchbrenner et al. (2016), **Neural Machine Translation in Linear Time**. [Paper](#). ByteNet; the numbers in these notes are from the tables of version 2 (March 2017).
- Łukasz Kaiser and Samy Bengio (2016), **Can Active Memory Replace Attention?** [Paper](#). The Extended Neural GPU.
- Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier (2017), **Language Modeling with Gated Convolutional Networks**. [Paper](#). Gated linear units.
- Jonas Gehring et al. (2017), **Convolutional Sequence to Sequence Learning**. [Paper](#). ConvS2S; the proceedings version, whose English–French score (40.51) differs slightly from the first arXiv versions⁷.

The Transformer

- Ashish Vaswani et al. (2017), **Attention Is All You Need**. [Paper](#). The subject of Lecture 4. The passages quoted here are in the 2017 versions: version 1 (June) and version 5 (December), which added the note on each author’s contribution.

Explanations

- Chris Olah and Shan Carter (2016), **Attention and Augmented Recurrent Neural Networks**. [Article](#).
- Lena Voita (2020), **Sequence to Sequence (seq2seq) and Attention**. [Lecture notes](#).
- E. A. Nadaraya (1964), **On Estimating Regression**. [Paper](#). Geoffrey S. Watson (1964), **Smooth Regression Analysis**. *Sankhyā, Series A* 26(4). Kernel regression ([Appendix A.4](#)).

Appendix A: Derivations

This appendix collects the derivations that the main text states, and the details of the attention variants that §7 describes in words.

A.1 The gradient of an attention read

The softmax's derivative. With $\alpha_i = e^{e_i} / \sum_j e^{e_j}$ for scores $e_1, \dots, e_n$,

$$\frac{\partial \alpha_i}{\partial e_j} = \alpha_i \delta_{ij} - \alpha_i \alpha_j, \quad \text{that is,} \quad \frac{\partial \alpha}{\partial \mathbf{e}} = \text{diag}(\alpha) - \alpha^\top \alpha$$

for a row α . This matrix is symmetric and positive semidefinite, its rows sum to zero (adding a constant to all scores changes nothing), and its diagonal entries are $\alpha_i(1 - \alpha_i) \leq \frac{1}{4}$. If one weight is $1 - \varepsilon$, every entry is $O(\varepsilon)$: a saturated softmax passes back almost no gradient to the scores.

From the context to the scores. For one read (dropping the index t), $\mathbf{c} = \sum_k \alpha_k \mathbf{h}_k$, so

$$\frac{\partial \mathbf{c}}{\partial e_j} = \sum_k \alpha_k (\delta_{kj} - \alpha_j) \mathbf{h}_k = \alpha_j (\mathbf{h}_j - \mathbf{c}), \quad \frac{\partial L}{\partial e_j} = \alpha_j (\mathbf{h}_j - \mathbf{c}) \gamma^\top,$$

with $\gamma = \partial L / \partial \mathbf{c}$ the gradient that backpropagation brings to the context from the rest of the decoder. The scores are activations, not parameters, so these are **local sensitivities**: how the loss would change if one score alone were nudged. Three things follow:

1. **Positions whose content would help are favored.** When $\mathbf{h}_j$ lies, as seen from the current context, in the direction that lowers the loss ($\mathbf{h}_j - \mathbf{c}$ points against γ), the sensitivity of e_j is negative: raising that score alone would lower the loss. For a position that lies the other way, it is positive.
2. **The sensitivities sum to zero.** Since $\sum_j \alpha_j (\mathbf{h}_j - \mathbf{c}) = \mathbf{c} - \mathbf{c} = 0$, raising all scores together would change nothing: attention is redistributed, not created.
3. **A small weight suppresses the direct signal.** Each sensitivity is proportional to α_j , so a position with almost no weight gets almost nothing through its own score at this step. Its state can still receive gradient by other routes: its state is read at other steps, and the parameters it depends on are shared.

The worked example. In the read of §6, suppose the rest of the decoder gives $\gamma = (0.5, -1)$: the loss would fall if the context moved toward $(-0.5, 1)$, up and to the left in Figure 15b. Then $(\mathbf{h}_j - \mathbf{c})\gamma^\top$ is 0.674, -1.326 , and -0.826 for the three positions, and multiplied by the weights 0.579, 0.084, and 0.337,

$$\frac{\partial L}{\partial e_1} = 0.390, \quad \frac{\partial L}{\partial e_2} = -0.112, \quad \frac{\partial L}{\partial e_3} = -0.278,$$

which sum to zero. Raising score 1 alone would raise the loss; raising score 2 or score 3 alone would lower it. State $\mathbf{h}_2$ lies furthest in the helpful direction, yet its sensitivity is the smallest of the three in size, because its weight, 0.084, scales it down.

On to the parameters. For the additive score, write $\mathbf{z}_j = \mathbf{s}W_a + \mathbf{h}_jU_a$, so that $e_j = \tanh(\mathbf{z}_j)\mathbf{v}_a^\top$. With $\delta_j = \partial L / \partial e_j$, backpropagation continues

$$\frac{\partial L}{\partial \mathbf{v}_a} = \sum_j \delta_j \tanh(\mathbf{z}_j), \quad \frac{\partial L}{\partial \mathbf{z}_j} = \delta_j \mathbf{v}_a \odot (1 - \tanh^2(\mathbf{z}_j)),$$

$$\frac{\partial L}{\partial W_a} = \mathbf{s}^\top \sum_j \frac{\partial L}{\partial \mathbf{z}_j}, \quad \frac{\partial L}{\partial U_a} = \sum_j \mathbf{h}_j^\top \frac{\partial L}{\partial \mathbf{z}_j},$$

and, for the inputs of the read,

$$\left. \frac{\partial L}{\partial \mathbf{s}} \right|_{\text{read}} = \sum_j \frac{\partial L}{\partial \mathbf{z}_j} W_a^\top, \quad \left. \frac{\partial L}{\partial \mathbf{h}_j} \right|_{\text{read}} = \underbrace{\alpha_j \gamma}_{\text{as content}} + \underbrace{\frac{\partial L}{\partial \mathbf{z}_j} U_a^\top}_{\text{as something compared}}.$$

The encoder state receives gradient by two routes: as the content that was read, in proportion to its weight, and as the thing the query was compared with, through the score. Summed over the decoder's steps, these are the signals that teach the encoder what to store at each position and how to make it findable.

What one update does. Every score's sensitivity contributes to the same $\partial L / \partial W_a$, $\partial L / \partial U_a$, and $\partial L / \partial \mathbf{v}_a$, so one step on these parameters moves all the scores together. Hold one read's query and states fixed and update only the score's parameters θ by gradient descent; with the sensitivities in the row δ and $J = \partial \mathbf{e} / \partial \theta$, a step of size η changes the scores, to first order, by $\Delta \mathbf{e} = -\eta \delta J J^\top$, and the loss by $\Delta \mathbf{e} \delta^\top = -\eta \|\delta J\|^2 \leq 0$. The argument shows no more than this. It is a first-order statement. Because $J J^\top$ mixes the scores, a single score can move against its own sensitivity. And in full training the update also changes the representations and combines the losses of many reads and examples.

A.2 Local attention

Luong et al. (2015) defined local attention around a center p_t , with the decoder's current state $\mathbf{s}_t$ as the query (§7). In *local-m*, $p_t = t$, assuming that source and target are roughly aligned in order. In *local-p* the center is predicted,

$$p_t = n \cdot \sigma(\tanh(\mathbf{s}_t W_p) \mathbf{v}_p^\top) \in [0, n],$$

with n the length of the source and W_p , $\mathbf{v}_p$ learned. The weights are computed by a softmax of the scores over the positions in the window $[p_t - D, p_t + D]$, and then multiplied by a Gaussian centered at p_t with standard deviation $D/2$,

$$\alpha_{tj} = \text{softmax}_j(\text{score}(\mathbf{s}_t, \mathbf{h}_j)) \cdot \exp\left(-\frac{(j - p_t)^2}{2(D/2)^2}\right),$$

for the integer positions j in the window; p_t is a real number. The weights so multiplied no longer sum to one, and Luong et al. do not renormalize them, so the read $\mathbf{c}_t = \sum_j \alpha_{tj} \mathbf{h}_j$ over the window is a weighted sum, not a weighted average. The Gaussian factor makes the weights differentiable in p_t , so the predicted center is trained by backpropagation with the rest of the network. They are differentiable almost everywhere, not everywhere: the window moves with p_t , and at the values of p_t where a position enters or leaves it, the softmax is taken over a different set of positions and the weights jump. The proceedings version says only that D “is empirically selected”; the arXiv version gives $D = 10$.

The three scores of the paper, with $\mathbf{s}_t$ and $\mathbf{h}_j$ as columns as in the paper, are $\mathbf{s}_t^\top \mathbf{h}_j$ (dot), $\mathbf{s}_t^\top W_a \mathbf{h}_j$ (general, our bilinear), and $\mathbf{v}_a^\top \tanh(W_a [\mathbf{s}_t; \mathbf{h}_j])$ (concat, our additive; the proceedings version prints it without $\mathbf{v}_a$ and the tanh, the arXiv version with them). A fourth, *location*-based, computes all the weights from the decoder's state alone, $\alpha_t = \text{softmax}(\mathbf{s}_t W_a)$ in our row notation, without looking at the source states.

With **input feeding**, the attentional state $\tilde{\mathbf{s}}_t$ is joined to the next input word's vector, so that the first layer of a decoder with d units per layer reads $2d$ numbers per step.

A.3 Hard attention and its gradient

A hard read samples a position $\tilde{j}$ from the weights α_t and returns $\mathbf{h}_{\tilde{j}}$. The log-likelihood of the target is then the logarithm of an average over all possible sequences of positions, $\log \sum_{\mathbf{a}} p(\mathbf{a}) p(\mathbf{y} \mid \mathbf{a})$, where $\mathbf{a}$ collects the positions sampled at every step and $p(\mathbf{a})$ is the product of the weights that chose them. Xu et al. (2015) maximized the lower bound given by Jensen's inequality,

$$L_s = \sum_{\mathbf{a}} p(\mathbf{a}) \log p(\mathbf{y} \mid \mathbf{a}) \leq \log \sum_{\mathbf{a}} p(\mathbf{a}) p(\mathbf{y} \mid \mathbf{a}).$$

Its gradient with respect to the parameters θ has two terms, since both $p(\mathbf{a})$ and $p(\mathbf{y} \mid \mathbf{a})$ depend on θ :

$$\frac{\partial L_s}{\partial \theta} = \sum_{\mathbf{a}} p(\mathbf{a}) \left[\frac{\partial \log p(\mathbf{y} \mid \mathbf{a})}{\partial \theta} + \log p(\mathbf{y} \mid \mathbf{a}) \frac{\partial \log p(\mathbf{a})}{\partial \theta} \right],$$

using $\partial p(\mathbf{a}) / \partial \theta = p(\mathbf{a}) \partial \log p(\mathbf{a}) / \partial \theta$. The sum over all sequences of positions is replaced by an average over a few sampled sequences. The second term is the **REINFORCE** estimator (Williams, 1992) with the log-likelihood of the target as the reward: a sampled sequence of positions is made more probable in proportion to how well the target is predicted from it. Its variance is high, so Xu et al. subtracted a running average of past rewards, $b \leftarrow 0.9b + 0.1 \log p(\mathbf{y} \mid \tilde{\mathbf{a}})$, from the reward, added an entropy term, and, with probability 0.5 for each image, replaced the sampled position by the soft read. Lecture 6 derives the estimator and the baseline in general.

A.4 Attention as kernel regression

Given observations (u_j, w_j) , the **Nadaraya–Watson** estimate of the regression function at a point u (Nadaraya, 1964; Watson, 1964) averages the observed values with weights given by a kernel κ :

$$\hat{f}(u) = \frac{\sum_j \kappa(u, u_j) w_j}{\sum_j \kappa(u, u_j)}.$$

Attention is this estimate with the query as the point, the keys as the observed inputs, the values as the observed outputs, and the kernel $\kappa(\mathbf{q}, \mathbf{k}) = \exp(\text{score}(\mathbf{q}, \mathbf{k}))$: the softmax is the normalization. For a dot-product score, $\exp(\mathbf{q}\mathbf{k}^\top) = \exp(-\frac{1}{2}\|\mathbf{q} - \mathbf{k}\|^2) \exp(\frac{1}{2}\|\mathbf{q}\|^2 + \frac{1}{2}\|\mathbf{k}\|^2)$; when all keys have the same length, the second factor is the same for every key and cancels, so attention is kernel regression with a Gaussian kernel, averaging the values whose keys lie near the query. What is new in attention is that the inputs and outputs of the regression are themselves computed by a network, trained with everything else, and that the observations themselves, the keys and values, change with every input.

Appendix B: History

The main text keeps to the line of argument; this appendix tells more of the history around it.

B.1 Translation before neural networks

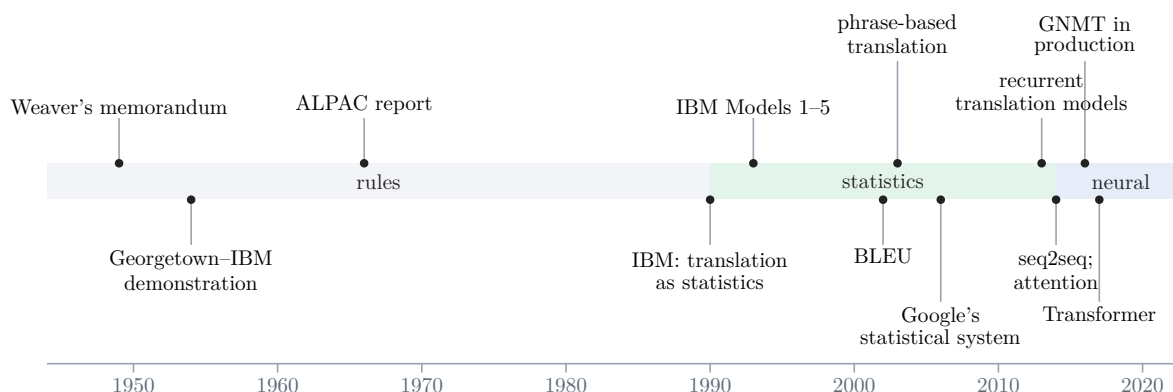


Figure 30. Seven decades of machine translation: rules, then statistics, then neural networks. The events of the main text and of this appendix, by year.

Translation was attempted with the methods of every period: rules, then statistics, then neural networks (Figure 30). In a memorandum of 1949, Warren Weaver suggested treating it as decoding a message: “When I look at an article in Russian, I say ‘This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode’” (Weaver, 1949, quoting a letter of his). The [first public demonstration](#), in 1954, translated Russian sentences on an IBM computer with a vocabulary of 250 stems and endings and six rules. Optimism followed, and so did funding, until a committee of the US National Academy of Sciences found in 1966 “no immediate or predictable prospect of useful machine translation” (the [ALPAC report](#)). Systems written by hand, as dictionaries and rules, continued in narrow uses, such as weather bulletins and technical manuals, and in commercial products.

From 1990 a group at IBM built systems that learned to translate from parallel corpora instead (Brown et al., 1990; 1993), with an approach brought over from speech recognition: a model of which words translate which, with the alignment as a hidden variable estimated by the EM algorithm, combined with an n-gram language model of the target language. Phrase-based systems (Koehn et al., 2003), built on these alignments, translated in units of a few words and were the best available method from 2003 until about 2016; Google put its own statistical system online in 2006 (Och, 2006). Neural networks entered translation in 2013–2014, first as parts of statistical systems and then as whole translators (§§3–4), and replaced phrase-based systems in production from 2016.

Speech recognition took a parallel path. Hidden Markov models with n-gram language models ran it for two decades, and neural networks entered it with methods for transductions whose alignment is monotonic, the output running in the same order as the input: connectionist temporal classification (CTC) sums the probabilities of all alignments between audio frames and labels (Graves et al., 2006), and the RNN transducer (RNN-T) extends it to outputs of any length (Graves, 2012). The RNN transducer paper put the task in one sentence: “Many machine learning tasks can be expressed as the transformation—or transduction—of input sequences into output sequences”. CTC and RNN-T both require a monotonic alignment, which fits speech but not translation’s reordered word correspondences. The neural answer that took hold in

translation was the design of §3: an encoder that reads the whole input, and a decoder that then writes the output.

B.2 The first neural translation models, 2013–2014

Sutskever et al. (2014) trained their deep LSTMs on the WMT 2014 English–French task, the shared task of the annual Workshop on Machine Translation, with plain stochastic gradient descent, a learning rate of 0.7 halved every half epoch after the fifth, batches of 128 pairs, and gradients rescaled when their norm exceeded 5. Each of the four LSTM layers ran on its own GPU, and four more GPUs computed the softmax. Decoded greedily, their ensemble scored 33.0 BLEU; with a beam of 2, 34.5, and with a beam of 12, 34.8.

Bahdanau et al. (2015) name the relation between their design and Cho et al.’s encoder–decoder: their attention model “becomes RNN Encoder–Decoder (Cho et al., 2014a), if we fix $\mathbf{c}_i$ to $\vec{\mathbf{h}}_{T_x}$ ”, the last forward state (their $\mathbf{c}_i$ and T_x are our $\mathbf{c}_t$ and n).

When the Transformer paper said in 2017 that recurrent networks, in particular LSTMs and gated recurrent networks such as the GRU, had been “firmly established as state of the art approaches in sequence modeling and transduction problems”, that was true of language modeling as well: an LSTM language model with a billion parameters, trained for three weeks on 32 GPUs, had set the record on a benchmark of a billion words, with a perplexity of 30.0 (Jozefowicz et al., 2016).

B.3 Attention, 2014–2017

Figure 31 places the papers of §§6–10 and of this appendix in time.

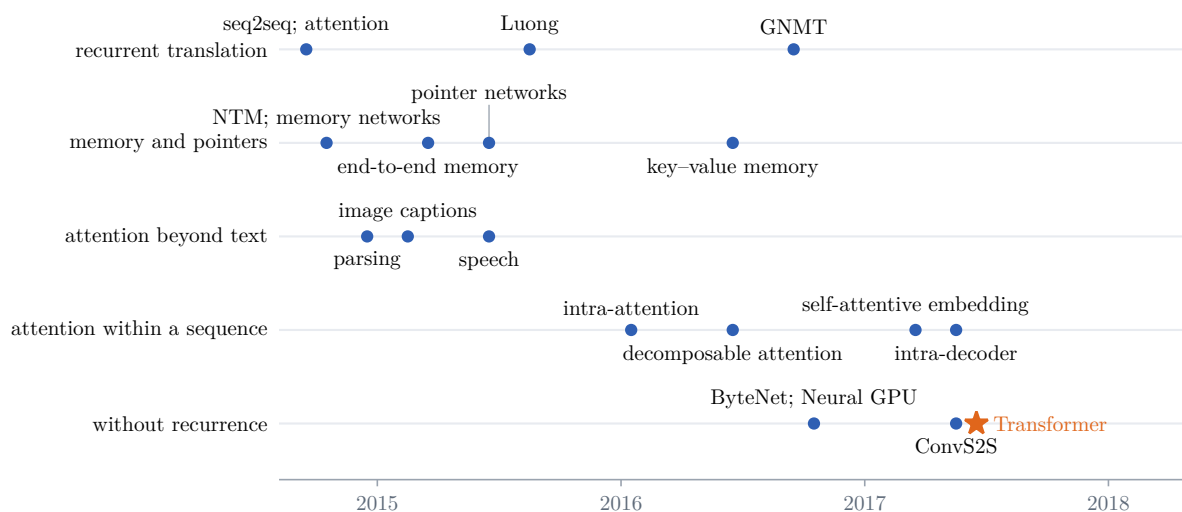


Figure 31. Attention spread from translation to memory, other inputs, and single sequences before the Transformer. The papers of §§6–10 and of this appendix, by the month they first appeared, in five lines of work. The Transformer (star) appeared in June 2017.

The name. Bahdanau, then an intern in the Montreal group that had built the RNN Encoder–Decoder, posted the paper with Cho and Bengio on 1 September 2014, nine days before Sutskever et al.’s; it appeared at ICLR the following spring. The authors called their model RNNsearch, for the soft search it performs over the input, and wrote in one paragraph that it “implements a mechanism of attention in the decoder”. In Bahdanau’s later account, the word *attention* came from Bengio in one of the final passes over the paper (Karpathy, 2024, who posted Bahdanau’s email). Graves (2013) had generated handwriting from text with a recurrent network that read the text through a window of Gaussian bumps whose centers could

only move forward; Bahdanau et al. cited it and called the forward-only movement “a severe limitation” for translation, which reorders words.

Reading from a memory. Seven weeks after Bahdanau et al., the Neural Turing Machine (Graves et al., 2014) gave a recurrent network an external memory, a matrix of rows that it could read and write. To read, the network

1. emits a key;
2. scores each row of the memory by its cosine similarity with the key, multiplied by a learned sharpness;
3. turns the scores into weights with a softmax;
4. returns the weighted average of the rows.

Graves et al. called this **content-based addressing**: attention with the encoder states replaced by a memory that the network writes itself. Memory networks applied the same reading to question answering, first with hard choices trained with labels (Weston et al., 2015), then, in end-to-end memory networks, with the soft, multi-hop reads of §7 and a learned vector for each sentence’s position in time (Sukhbaatar et al., 2015).

Other inputs. Besides the image captions of Xu et al., attention-based encoder–decoders recognized speech (Chorowski et al., 2015; Chan et al., 2016) and parsed sentences into trees written as strings (Vinyals et al., 2015a). Pointer networks used the attention weights themselves as the output, a pointer to an input position (Vinyals et al., 2015b). By 2016 an encoder–decoder with attention was the default model for most tasks whose output is a sequence.

Recurrence at scale. Google’s Neural Machine Translation system (Wu et al., 2016), GNMT, was the attention model of §6 made large: eight LSTM layers in the encoder, the first of them bidirectional, and eight in the decoder, residual connections between the layers, attention from the decoder to the top of the encoder, and a vocabulary of word pieces, which removed the unknown words that had held back Bahdanau’s model. A single model scored 39.0 BLEU on English-to-French and 24.6 on English-to-German. In September 2016 it replaced the phrase-based system for Chinese to English, and in ratings of isolated sentences by people it closed between 58% and 87% of the gap between the phrase-based system and human translators, depending on the language pair. Its authors found that without residual connections “simple stacked LSTM layers work well up to 4 layers, barely with 6 layers, and very poorly beyond 8 layers”.

Removing the recurrence. Besides ByteNet and ConvS2S (§8), the Extended Neural GPU (Kaiser and Bengio, 2016) replaced attention by an “active memory” that updates all positions in parallel with convolutions at every step. Several earlier papers (Cheng et al., 2016; Parikh et al., 2016; Lin et al., 2017; Paulus et al., 2017) each anticipated part of the Transformer’s self-attention, attention within a single sequence. Parikh et al. went furthest. To decide whether one sentence follows from another, their model compared the two sentences with attention between their words and applied small networks to each word separately, so that every step was computed at all positions at once (“trivially parallelizable”); with 382,000 parameters it matched the accuracy of an LSTM-based model with attention that had 3.4 million (86.3% on the SNLI benchmark). A variant also let each word attend to the words of its own sentence. The paper’s last author, Jakob Uszkoreit, is the author of the Transformer paper who, in its contribution note, “proposed replacing RNNs with self-attention”.