Chapter 7: Sequence-to-Sequence and Decoding

Learning Objectives

After reading this chapter, the reader should be able to:

1. Describe the encoder-decoder architecture for conditional text generation and explain how the encoder produces representations that the decoder conditions on via attention.
2. Explain the training-inference discrepancy caused by teacher forcing, identify exposure bias as its consequence, and describe scheduled sampling as a mitigation strategy.
3. Implement and compare decoding strategies – greedy search, beam search, top-k sampling, and nucleus (top-p) sampling – and explain the quality-diversity tradeoff each embodies.
4. Compute BLEU score for a generated translation and critically evaluate the strengths and weaknesses of automated evaluation metrics for text generation.

In Chapter 6, we developed the attention mechanism: the decoder can now dynamically focus on relevant parts of the source sequence instead of relying on a single compressed vector. But attention is only one component of a complete generation system. We built a flashlight – now we need to build the entire search party. How do we train an encoder-decoder model end to end? How do we turn a probability distribution over the vocabulary into actual text? And once we have that text, how do we measure whether it is any good? These three questions – training, decoding, and evaluation – define the practical engineering of text generation, and answering them is the subject of this chapter. We assemble the attention mechanism into a full encoder-decoder framework, confront the training challenge of teacher forcing and its uncomfortable consequence called exposure bias, present decoding strategies ranging from the naive (greedy search) to the modern (nucleus sampling), and introduce the evaluation metrics that let us measure progress. Throughout, machine translation serves as our running case study – the task that drove the development of both attention and sequence-to-sequence models, and the testing ground where most of these ideas were first validated before spreading to summarization, dialogue, and every other text generation application.

This chapter marks a transition in the book. Until now, we have been concerned with assigning probabilities to text – the prediction paradigm established in Chapter 1, pursued through n-gram counts, neural networks, and attention mechanisms. Here, the model stops merely predicting and starts producing. The decoder is the bridge from probability to language, from theory to application, from the question "what word is likely?" to the act of writing a sentence. That bridge turns out to be surprisingly difficult to cross, and the strategies we develop for crossing it – beam search, nucleus sampling, temperature scaling – remain at the core of every modern language generation system, from chatbots to machine translation engines to large language models.

7.1 Encoder-Decoder Architecture

7.1.1 The Encoder: From Input to Representations

How does a neural network read an entire input sentence and compress it into a form that another network can use for generation?

In 2014, two research groups independently proposed the same idea: use one neural network to read an input sequence and another to write an output sequence. Sutskever, Vinyals, and Le [2014] at Google and Cho et al. [2014] at the University of Montreal both demonstrated that this simple decomposition – an encoder that reads and a decoder that writes – could perform machine translation without any of the hand-engineered rules, phrase tables, or linguistic features that had dominated the field for a decade. The encoder's job is conceptually straightforward: take a sequence of tokens $x_1, x_2, \ldots, x_T$ and produce a set of representations that capture the meaning of each token in context. In practice, the encoder is typically a bidirectional LSTM (Section 5.3) that processes the input left-to-right and right-to-left, concatenating the forward hidden state $\overrightarrow{\mathbf{h}}_j$ and backward hidden state $\overleftarrow{\mathbf{h}}_j$ at each position to produce a context-sensitive representation $\mathbf{h}_j = [\overrightarrow{\mathbf{h}}_j; \overleftarrow{\mathbf{h}}_j] \in \mathbb{R}^{2d}$ that encodes position j with information about both its left and right contexts. The full encoder output is the matrix $\mathbf{H} = [\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T]$ – one vector per input token, each enriched with contextual information from the entire source sentence.

A critical point that frequently confuses students is what constitutes the "output" of the encoder. In the original architecture of Sutskever et al. [2014], the encoder output was a single vector $\mathbf{h}_T$ – the final hidden state, which had to compress the entire source sentence into a fixed-dimensional representation. As we established in Chapter 6 (Section 6.1), this creates an information bottleneck that degrades performance on long sentences. With attention, the encoder output is no longer a single vector but the entire sequence of hidden states $\mathbf{H}$. Every position is preserved. Nothing is thrown away. The attention mechanism (Section 7.1.3) will determine which positions matter for each step of generation. The encoder's responsibility is to produce the richest possible representation at every position and leave the selection problem to the decoder.

7.1.2 The Decoder: Autoregressive Generation

The decoder generates the output sequence one token at a time, left to right, in a process called *autoregressive generation*. At each step i, the decoder produces a probability distribution over the entire vocabulary V and selects (or samples) the next token y_i. The distribution at step i depends on three inputs: the previously generated token y_{i-1} (embedded via a learned embedding matrix), the decoder's own hidden state $\mathbf{s}_{i-1}$ (which summarizes the target-side history), and a context vector $\mathbf{c}_i$ computed via attention over the encoder states. The decoder hidden state is updated via an LSTM cell that takes as input the concatenation of the previous token embedding and the context vector, producing the new hidden state $\mathbf{s}_i = \text{LSTM}(\mathbf{s}_{i-1}, [\text{emb}(y_{i-1}); \mathbf{c}_i])$. The output distribution is then computed as $P(y_i \mid y_{<i}, \mathbf{x}) = \text{softmax}(\mathbf{W}_o[\mathbf{s}_i; \mathbf{c}_i] + \mathbf{b}_o)$, where $\mathbf{W}_o$ and $\mathbf{b}_o$ are learnable parameters.

The word *autoregressive* means that each prediction depends on all previous predictions. This creates a chain of dependencies: the choice of y_1 affects the distribution over y_2, which affects y_3, and so on through the entire output sequence. There is no going back – once a token is generated, it becomes part of the conditioning context for all future tokens. This sequential dependency is both the source of the decoder's power (it can model complex long-range dependencies in the output) and its greatest practical challenge (it makes generation inherently sequential and, as we will see in Section 7.2, creates a fundamental mismatch between training and inference). The decoder stops generating when it produces a special end-of-sequence token $\langle\text{EOS}\rangle$ or when a maximum length limit is reached. Generation always begins with a special start-of-sequence token $\langle\text{SOS}\rangle$ as y_0.

7.1.3 Conditioning via Attention

Think of the decoder as a translator working from notes. Attention is the bridge between encoder and decoder. At each decoder step i, the attention mechanism computes a *context vector* $\mathbf{c}_i$ that summarizes the most relevant parts of the encoder output for generating the current target token. We developed the mathematics of this mechanism in detail in Chapter 6 (Sections 6.2 and 6.3), so here we focus on how the pieces fit together within the full encoder-decoder system. At decoder step i, the decoder state $\mathbf{s}_{i-1}$ serves as the *query* – it represents "what the decoder is looking for." The encoder hidden states $\mathbf{h}_1, \ldots, \mathbf{h}_T$ serve as both *keys* and *values*. The attention mechanism computes alignment scores $e_{ij} = \text{score}(\mathbf{s}_{i-1}, \mathbf{h}_j)$ using either the additive (Bahdanau) or multiplicative (Luong) scoring function from Chapter 6, normalizes them with softmax to get attention weights $\alpha_{ij} = \text{softmax}(e_{ij})$, and produces the context vector $\mathbf{c}_i = \sum_{j=1}^{T} \alpha_{ij}\mathbf{h}_j$ as a weighted sum of encoder states. The context vector $\mathbf{c}_i$ is then concatenated with the decoder state $\mathbf{s}_i$ and fed through the output layer to produce the vocabulary distribution.

This is *cross-attention*: the decoder attends to the encoder. The decoder's hidden state tracks what has been generated so far on the target side, while the context vector provides relevant information from the source side. Together, they give the output layer the complete picture it needs to predict the next token. A useful way to think about this division of labor is that the hidden state is the translator's working memory of the translation so far, while the context vector is the translator's glance at the relevant passage of the source document for the current sentence being translated. Neither is sufficient alone – a translator who remembers what they have written but cannot consult the source will drift from the original meaning, while a translator who can see the source but forgets what they have already written will produce incoherent output. The attention mechanism ensures that both sources of information are available at every generation step, and the network learns to combine them appropriately.

7.1.4 The General Seq2Seq Framework

The encoder-decoder architecture is not specific to machine translation – it is a general framework for any task that maps a variable-length input sequence to a variable-length output sequence. Translation maps French to English. Summarization maps a document to a summary. Dialogue maps a conversation history to a response. Code generation maps a natural language description to a program. Question answering maps a question-context pair to an answer. In every case, the structure is the same: an encoder reads the input and produces contextualized representations, and a decoder generates the output one token at a time, conditioned on the encoder representations via attention. This generality is why the framework became so influential. When Raffel et al. [2020] introduced the T5 model (which we study in Chapter 9), they took this idea to its logical extreme by casting *every* NLP task as a text-to-text problem: the input is always a text string, the output is always a text string, and the model is always an encoder-decoder. Classification becomes "input: review text, output: positive." Summarization becomes "input: article, output: summary." The architecture does not change – only the training data differs.

Sidebar: Sutskever's Trick

In their landmark 2014 paper, Ilya Sutskever, Oriol Vinyals, and Quoc Le at Google reported a finding that struck many researchers as bizarre: simply reversing the order of the source sentence before encoding it improved BLEU scores by nearly 5 points on English-French translation. Reading "the cat sat on the mat" as "mat the on sat cat the" made the translations measurably better. The explanation was elegant and,

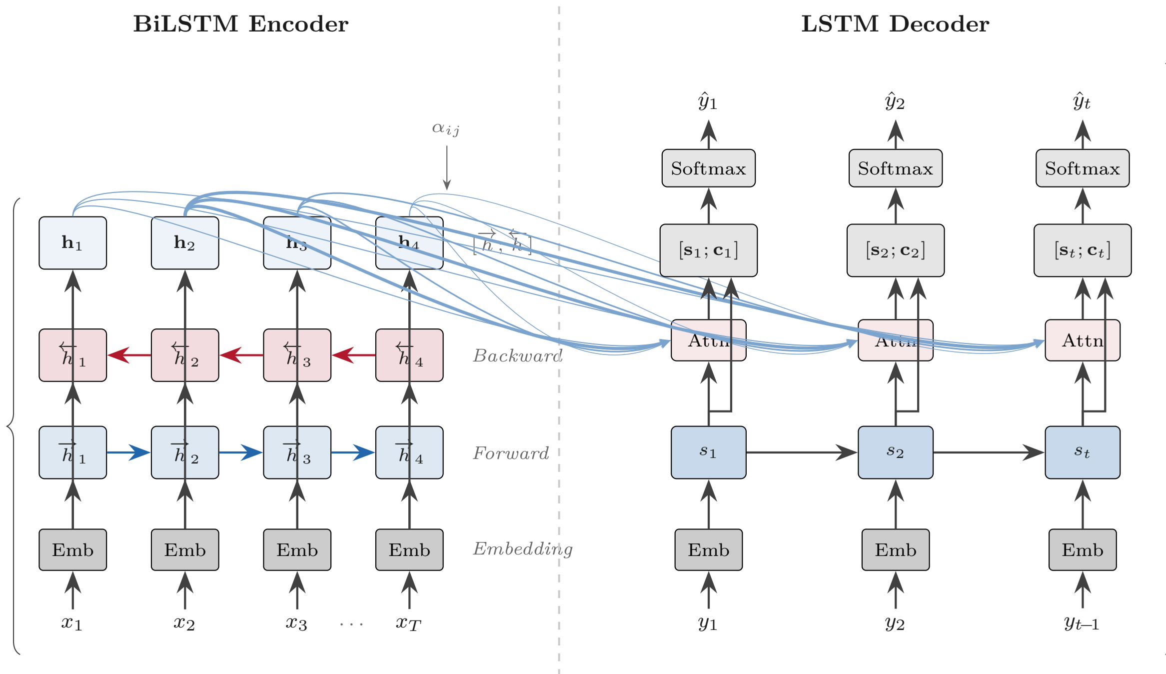

Figure 1: Figure 7.1 – The complete encoder-decoder architecture with attention

in retrospect, obvious. In a non-attention encoder-decoder, the decoder must generate the first target word from the encoder's final hidden state $\mathbf{h}_T$. If the source is read in natural order, the first source word (which often corresponds to the first target word in language pairs with similar word order) is the most distant from $\mathbf{h}_T$ in the recurrent chain. Reversing the source places the first source word last in the encoder, making it "freshest" in the final hidden state and shortening the average distance between corresponding source-target words. This clever hack – simple, unprincipled, and surprisingly effective – was enormously influential for about a year. Then Bahdanau's attention mechanism arrived, giving the decoder direct access to all encoder positions regardless of distance, and the reversal trick became unnecessary. It remains a beautiful illustration of a broader pattern in machine learning research: sometimes the best solution to a hard problem is a simple hack that works surprisingly well, until a principled solution arrives to replace it.

7.2 Teacher Forcing and Exposure Bias

7.2.1 Teacher Forcing: Fast Training, Hidden Cost

Consider the sentence pair ("Le chat est sur le tapis", "The cat is on the mat") used to train a translation model. During training, when the decoder is asked to predict the third target word "is", what does it receive as input? There are two options: (a) the correct previous word "cat" from the ground-truth target sequence, or (b) whatever word the model actually predicted at the previous step, which might have been "dog" or "feline" or any other token. Nearly every sequence-to-sequence model uses option (a). This training strategy is called *teacher forcing* [Williams and Zipser, 1989]: at

each decoder step i, the model receives the ground-truth token y_{i-1} as input, not its own prediction $\hat{y}_{i-1}$. The "teacher" (the ground-truth sequence) forces the correct input at every step, regardless of what the model would have generated on its own.

Teacher forcing makes training dramatically faster and more stable. Because the decoder always receives the correct prefix, the loss at each position can be computed independently. The cross-entropy loss decomposes as $\mathcal{L} = -\sum_{i=1}^{S} \log P(y_i \mid y_1, \ldots, y_{i-1}, \mathbf{x})$, where the conditioning on $y_1, \ldots, y_{i-1}$ uses the ground-truth tokens. This means that the gradients at different decoder positions do not depend on each other through the predicted tokens – the model does not need to propagate through its own sampling decisions. Training is therefore embarrassingly parallel across decoder positions (given the ground-truth prefix), and the loss landscape is smoother because the model always operates from correct contexts. Without teacher forcing, the model would need to generate its own predictions, sample from them, and backpropagate through the sampling process – a problem that is both computationally expensive and notationally unpleasant due to the non-differentiability of discrete sampling.

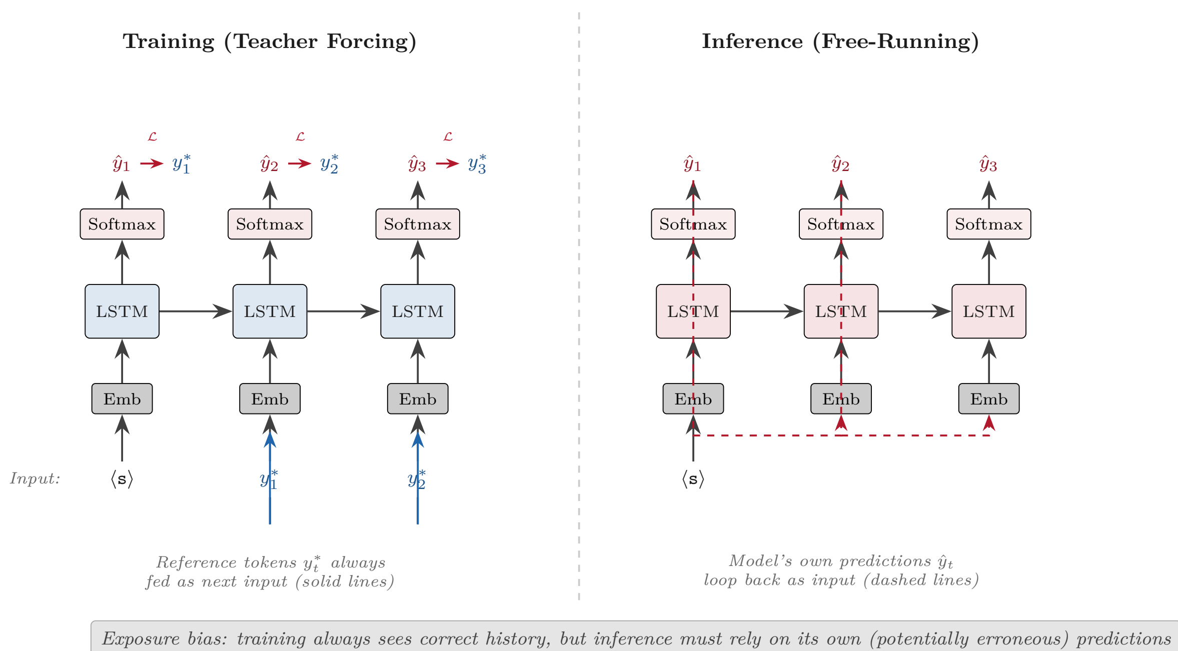

Figure 2: Figure 7.2 – Teacher forcing versus free-running inference

7.2.2 Exposure Bias: The Train-Test Gap

Why does a model trained on perfect inputs often fail catastrophically when it encounters its own imperfect outputs?

The hidden cost of teacher forcing has a name: *exposure bias*. During training, the decoder is exposed only to ground-truth prefixes – perfect sequences without errors. During inference, the decoder must condition on its own predictions, which inevitably contain errors. The model has never seen an incorrect prefix during training and has therefore never learned to recover from one. When the model generates "The dog is on the mat" instead of "The cat is on the mat," every subsequent

token must be generated conditioned on a context that includes "dog" – a context the model has never encountered during training. The result is often a cascade of errors: one mistake leads to another, and the generated sequence diverges further and further from anything the model saw during training.

A common misconception is that exposure bias is a minor issue. It is not – it is the primary failure mode of teacher-forced models in practice. Ranzato et al. [2016] demonstrated that exposure bias causes sequence-level errors that accumulate over the length of the output. The longer the generated sequence, the more opportunities for error accumulation, and the worse the degradation. The analogy is a GPS navigation system that was trained only on routes starting from the correct current position. Such a system works flawlessly as long as the driver follows directions perfectly. But if the driver misses a turn, the GPS has no experience with "wrong" starting positions and gives nonsensical directions instead of rerouting. A robust GPS must be trained on diverse starting conditions, including incorrect ones. Similarly, a robust decoder must be exposed to imperfect prefixes during training – which is precisely what scheduled sampling attempts.

We are not entirely sure whether exposure bias deserves its status as the dominant explanation for all decoder failure modes. Some researchers have argued that the problem is overstated, that large-scale pre-training (Chapter 9) implicitly mitigates it by exposing models to such vast linguistic diversity that any conceivable prefix becomes reasonably in-distribution. The debate continues, and it is worth noting that the most successful modern generation systems (GPT-4, Claude, Gemini) use teacher forcing during pre-training without explicit exposure bias mitigation, relying instead on sheer scale and diversity of training data. The problem may be less about the training strategy and more about the size of the world the model is trained on.

7.2.3 Scheduled Sampling

Scheduled sampling [Bengio et al., 2015] offers a principled compromise between teacher forcing and fully autoregressive training. The idea is simple: at each decoder step i during training, flip a biased coin. With probability ϵ, use the ground-truth token y_{i-1} as input (teacher forcing). With probability $1 - \epsilon$, use the model's own prediction $\hat{y}_{i-1}$ as input (autoregressive). Start training with ϵ close to 1.0 (mostly teacher forcing, for stability) and gradually decrease ϵ toward 0 over the course of training (mostly autoregressive, for robustness). This creates a curriculum: the model begins learning from perfect inputs (easy) and gradually transitions to learning from its own imperfect inputs (hard), building the ability to recover from errors incrementally rather than being thrown into the deep end.

Three decay schedules are commonly used. Linear decay decreases ϵ at a constant rate: $\epsilon_t = \max(0, 1 - t/T_{\text{total}})$. Exponential decay decreases ϵ multiplicatively: $\epsilon_t = k^t$ for some $k < 1$. Inverse sigmoid decay provides a smooth S-shaped transition: $\epsilon_t = k/(k + \exp(t/k))$. The choice of schedule and its hyperparameters affect training dynamics: too rapid a decrease in ϵ destabilizes training (the model has not yet learned enough to produce useful predictions), while too slow a decrease provides insufficient exposure to imperfect inputs. Bengio et al. [2015] reported improvements of 1 to 2 BLEU points on image captioning and machine translation tasks, modest but consistent gains that demonstrated the value of addressing exposure bias even partially. However, scheduled sampling introduces its own theoretical concern: the training objective is no longer a valid maximum likelihood estimator because the input distribution at each step is a mixture of the data distribution and the model distribution, and this mixture changes over training. The estimator can be biased, and the precise nature of this bias is not well understood analytically.

7.2.4 Other Mitigation Strategies

Several alternatives to scheduled sampling have been proposed, each attacking exposure bias from a different angle. Sequence-level training with REINFORCE [Ranzato et al., 2016] sidesteps the problem entirely by optimizing sequence-level rewards (such as BLEU score) directly using policy gradient methods from reinforcement learning. Instead of maximizing the likelihood of individual tokens conditioned on ground-truth prefixes, the model generates complete sequences and receives a reward signal based on how good the complete sequence is. This eliminates the train-test discrepancy because the model generates from its own predictions during training, just as it does during inference. The catch is variance: REINFORCE gradients are notoriously noisy, requiring large batch sizes and variance reduction techniques (such as baseline subtraction) to train stably. Minimum risk training [Shen et al., 2016] takes a similar approach but minimizes the expected loss (risk) under the model's own output distribution rather than maximizing a reward, providing a slightly different optimization landscape with similar practical challenges.

Data augmentation offers a simpler alternative. By injecting noise into the decoder inputs during training – replacing tokens with random alternatives, dropping tokens, or permuting short spans – the model encounters imperfect prefixes without the computational overhead of autoregressive training. This approach is less principled than scheduled sampling or REINFORCE but often effective in practice, particularly when combined with other regularization techniques. It is worth noting that the exposure bias problem has become somewhat less central in the era of large pre-trained models. The Transformer architecture (Chapter 8) trains with teacher forcing just as RNN-based models do, and massive pre-training (Chapter 9) exposes the model to such enormous linguistic diversity that the gap between training and inference distributions shrinks substantially. The most impactful mitigation for exposure bias may ultimately have been not any clever training trick but simply training on more data. Nevertheless, the concept remains important for understanding why smaller models and models trained on limited data exhibit characteristic failure patterns – repetitive loops, incoherent digressions, and sudden topic shifts – that can be traced back to the model encountering contexts at inference time that it never saw during training.

7.3 Decoding Strategies

7.3.1 Greedy Decoding

27 BLEU on English-German. That was the score of a well-trained encoder-decoder model with attention in 2016 – and the entire difference between 27 and 32 BLEU came not from a better model, but from a better way of choosing tokens from the probability distribution the model already produced. The decoding strategy matters enormously, and the simplest possible strategy – greedy decoding – illustrates why.

Greedy decoding selects the highest-probability token at each step: $y_i = \arg\max_{w \in V} P(w \mid y_{<i}, \mathbf{x})$. It is fast, requiring only $\mathcal{O}(S \cdot |V|)$ operations for a sequence of length S, and it is deterministic – the same input always produces the same output. But it is fundamentally myopic. By committing to the locally best token at each step, greedy decoding can miss globally optimal sequences that require choosing a less likely token early to enable much better tokens later. Consider a model generating text after the prompt "The president of France." Suppose $P(\text{"is"} \mid \text{context}) = 0.35$ and $P(\text{"Emmanuel"} \mid \text{context}) = 0.25$. Greedy decoding picks "is." But the sequence "Emmanuel Macron announced" might have a much higher total probability than "is the leader of," because "Emmanuel"

leads to a highly concentrated distribution over "Macron" while "is" leads to a flat distribution over many plausible continuations. Greedy decoding cannot look ahead to discover this. It is a chess player who always captures the first available piece without considering positional consequences three moves later.

7.3.2 Beam Search

Can we find better output sequences by exploring multiple candidates simultaneously instead of committing to one token at a time?

Beam search addresses the myopia of greedy decoding by maintaining B candidate sequences (called *hypotheses* or *beams*) at each step, where B is the *beam width.* At each step i, each of the B current hypotheses is extended by every token in the vocabulary, producing $B \cdot |V|$ candidate extensions. These candidates are scored by their cumulative log-probability, and the top B are retained. The process continues until all beams produce the end-of-sequence token or a maximum length is reached, at which point the highest-scoring complete hypothesis is selected as the output.

The score of a hypothesis $\mathbf{y} = (y_1, \ldots, y_S)$ given input $\mathbf{x}$ is the sum of log-probabilities of each token:

$$\text{score}(\mathbf{y} \mid \mathbf{x}) = \sum_{t=1}^{S} \log P(y_t \mid y_{<t}, \mathbf{x}) \tag{7.1}$$

This formulation has a systematic bias toward shorter sequences. Because log-probabilities are negative (probabilities are between 0 and 1), adding more terms makes the score more negative. A two-word sequence will almost always outscore a twenty-word sequence simply because it accumulates fewer penalties. To correct this, length normalization divides the score by the sequence length raised to a power α:

$$\text{score}_{\text{norm}}(\mathbf{y} \mid \mathbf{x}) = \frac{1}{S^\alpha} \sum_{t=1}^{S} \log P(y_t \mid y_{<t}, \mathbf{x}) \tag{7.2}$$

The hyperparameter α controls the strength of the length normalization. Setting $\alpha = 0$ recovers the unnormalized score (which favors short sequences). Setting $\alpha = 1$ divides by the raw length (which can overly favor long sequences). Values in the range $\alpha \in [0.6, 0.7]$ work well empirically across most translation tasks [Wu et al., 2016]. The Google Neural Machine Translation system used $\alpha = 0.6$ in their production system, a value that has since become something of a default in the research community.

Let us walk through a concrete example. Suppose our vocabulary is $V = \{\text{the, cat, sat, on, mat, EOS}\}$ and the beam width is $B = 3$. At step 1, the model computes $P(w \mid \text{SOS})$ for all $w \in V$. Suppose the top 3 are "the" ($\log p = -0.2$), "a" ($\log p = -1.1$), and "cat" ($\log p = -1.5$). These become our three beams. At step 2, each beam is extended by all tokens in V, producing 18 candidates. We compute the cumulative log-probability for each and keep the top 3. Perhaps "the cat" (score $-0.2 + (-0.3) = -0.5$), "the sat" (score $-0.2 + (-0.8) = -1.0$), and "a cat" (score $-1.1 + (-0.1) = -1.2$). Notice that the beam "cat" from step 1 was entirely eliminated – none of its extensions scored in the top 3. This pruning is the essence of beam search: maintain diverse hypotheses but ruthlessly discard unpromising ones.

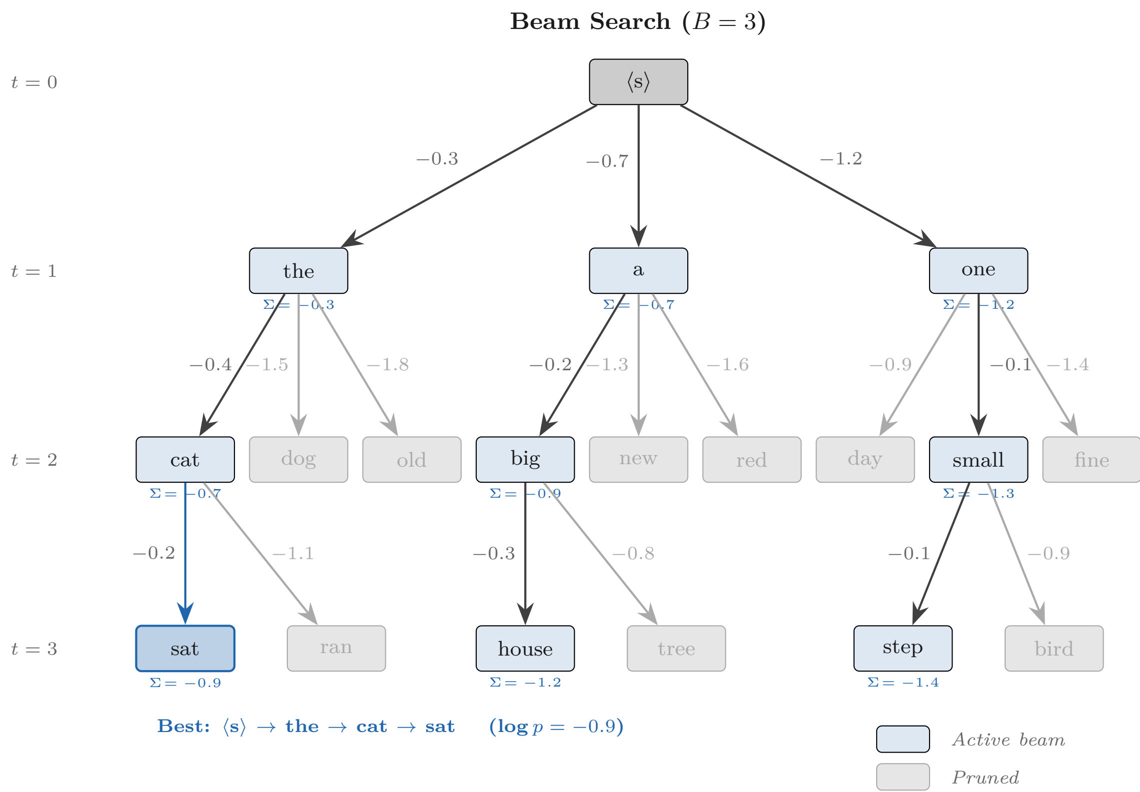

Figure 3: Figure 7.3 – Beam search tree with beam width $B = 3$ over 3 time steps

The time complexity of beam search is $\mathcal{O}(B \cdot |V| \cdot S)$ – a factor of B more expensive than greedy decoding. For typical values ($B = 5$, $|V| = 32{,}000$, $S = 50$), this is manageable. Increasing B improves quality with diminishing returns: the jump from $B = 1$ to $B = 5$ is substantial, the jump from $B = 5$ to $B = 10$ is modest, and beyond $B = 20$ the improvements are typically negligible for translation tasks. Koehn and Knowles [2017] observed that very large beam widths can actually degrade translation quality, a counterintuitive finding that suggests beam search may be optimizing a criterion (sequence probability) that does not perfectly align with translation quality.

```python
import torch
import torch.nn.functional as F

def beam_search(log_prob_fn, bos_id, eos_id, beam_width=3, max_len=10):
    """Beam search over a toy next-token log-probability function."""
    # Each beam: (score, token_list)
    beams = [(0.0, [bos_id])]
    completed = []
    for _ in range(max_len):
        candidates = []
        for score, tokens in beams:
            if tokens[-1] == eos_id:
                completed.append((score, tokens))
                continue
            log_probs = log_prob_fn(tokens)          # shape: (vocab_size,)
            topk_lp, topk_ids = log_probs.topk(beam_width)
            for lp, tid in zip(topk_lp.tolist(), topk_ids.tolist()):
                candidates.append((score + lp, tokens + [tid]))
        if not candidates:
            break
        candidates.sort(key=lambda x: x[0], reverse=True)
        beams = candidates[:beam_width]
    all_hyps = completed + beams
    all_hyps.sort(key=lambda x: x[0] / len(x[1]) ** 0.6, reverse=True)
    return all_hyps[0]  # best length-normalized hypothesis

# --- Toy demo: fixed log-probs for a 6-token vocabulary ---
VOCAB = {0: "SOS", 1: "the", 2: "cat", 3: "sat", 4: "mat", 5: "EOS"}
toy_logits = torch.tensor([[-5., 0.2, -1., -2., -3., -4.],   # after SOS
                           [-5., -3., -0.3, -0.8, -2., -4.],  # after "the"
                           [-5., -3., -3., -1.0, -0.2, -0.5]]) # after "cat"
def toy_log_prob_fn(tokens):
    step = min(len(tokens) - 1, len(toy_logits) - 1)
    return F.log_softmax(toy_logits[step], dim=-1)

best_score, best_tokens = beam_search(toy_log_prob_fn, 0, 5, beam_width=3)
print("Best:", " ".join(VOCAB[t] for t in best_tokens))
print(f"Score (length-normalized): {best_score:.3f}")
# Best: SOS the cat mat EOS
# Score (length-normalized): -0.612
```

7.3.3 Top-k Sampling

What if we want a language model to be creative rather than predictable?

Beam search finds high-probability sequences, but high probability is not always what we want. For machine translation, where there is typically one correct answer, maximizing probability makes sense. For open-ended generation tasks – story writing, dialogue, brainstorming – the most probable sequence is often generic and boring. “The meaning of life is to be happy” is more probable than “The meaning of life is to wrestle a philosophical crocodile,” but the latter is more interesting, and in many applications, interesting matters more than probable. Sampling-based decoding methods introduce controlled randomness to produce diverse, creative outputs while avoiding the incoherence of sampling from the full vocabulary distribution.

Top-k sampling [Fan et al., 2018] restricts the sampling distribution to the k most probable tokens and renormalizes. At each step i, the model computes the full distribution $P(w \mid y_{<i}, \mathbf{x})$, zeroes out all tokens except the top k by probability, and renormalizes the remaining probabilities to sum to 1. A token is then sampled from this truncated distribution. For $k = 1$, top-k reduces to greedy decoding. For $k = |V|$, it is unrestricted sampling. Values like $k = 40$ or $k = 50$ are common in practice, keeping enough tokens for variety while excluding the long tail of implausible continuations – typos, ungrammatical fragments, tokens in the wrong language – that would degrade output quality if sampled. The limitation of top-k is that the optimal value of k varies wildly across contexts. After “The capital of France is,” only one or two tokens are sensible (the nucleus of the distribution is tiny). After “I enjoy eating,” dozens of foods are plausible (the nucleus is wide). A fixed k is either too restrictive in some contexts or too permissive in others. This limitation motivated the development of nucleus sampling.

7.3.4 Nucleus (Top-p) Sampling

In 2020, Ari Holtzman and colleagues published a paper whose title – “The Curious Case of Neural Text Degeneration” – articulated a problem that practitioners had known about for years but nobody had named: neural language models, when decoded with likelihood-maximizing strategies like beam search, produce text that is repetitive, generic, and eventually degenerates into loops. The paper’s central insight was that the problem lies not with the model but with the decoding strategy. Human language is not the most probable sequence of tokens. Human writers make surprising, context-dependent word choices that beam search would prune away. The solution Holtzman et al. proposed was *nucleus sampling* (also called *top-p sampling*): instead of a fixed number of tokens, sample from the smallest set of tokens whose cumulative probability exceeds a threshold p.

Formally, the nucleus V_p at a given step is the smallest subset of the vocabulary such that the cumulative probability of tokens in the subset exceeds p:

$$V_p = \arg\min_{V' \subseteq V} \left\{ |V'| : \sum_{w \in V'} P(w \mid \text{context}) \geq p \right\} \tag{7.3}$$

The key advantage is adaptivity. When the model is confident – as after “The capital of France is” – the nucleus is tiny, perhaps just {Paris} at $p = 0.9$. When the model is uncertain – as after “I like to eat” – the nucleus expands to include many plausible foods. The sampling distribution adjusts automatically to the entropy of the model’s prediction at each step, without requiring the practitioner to guess how many tokens are appropriate. In practice, values of $p \in [0.9, 0.95]$ work

well for most open-ended generation tasks. The resulting text is more diverse than beam search output, more coherent than unrestricted sampling, and – crucially – closer in statistical profile to actual human writing. Holtzman et al. [2020] showed that the token-level probability distribution of text generated by nucleus sampling closely matches the distribution of human-written text, while beam search text is far more concentrated (lower entropy) than human text.

```python
import torch
import torch.nn.functional as F

def nucleus_sample(logits, p=0.9, temperature=1.0):
    """Sample from the nucleus (top-p) of a logit distribution."""
    logits = logits / temperature
    probs = F.softmax(logits, dim=-1)
    sorted_probs, sorted_idx = probs.sort(descending=True)
    cumulative = sorted_probs.cumsum(dim=-1)
    # Find cutoff: first index where cumulative prob >= p
    mask = cumulative - sorted_probs >= p   # exclude tokens beyond nucleus
    sorted_probs[mask] = 0.0
    sorted_probs /= sorted_probs.sum()      # renormalize
    # Sample from the truncated distribution
    sample_idx = torch.multinomial(sorted_probs, num_samples=1)
    return sorted_idx[sample_idx].item()

# --- Demo: sample 8 tokens from a synthetic 50-token distribution ---
torch.manual_seed(42)
logits = torch.randn(50)   # random logits for 50 tokens
print("p=0.5:", [nucleus_sample(logits, p=0.5) for _ in range(8)])
print("p=0.9:", [nucleus_sample(logits, p=0.9) for _ in range(8)])
print("p=0.95:", [nucleus_sample(logits, p=0.95) for _ in range(8)])
# p=0.5:  [13, 13, 13, 45, 13, 45, 13, 13]   <- tight nucleus, low diversity
# p=0.9:  [13, 45, 30, 13, 2, 45, 13, 30]     <- moderate diversity
# p=0.95: [13, 45, 30, 2, 17, 13, 45, 6]      <- wider nucleus, more diversity
```

Sidebar: The Curious Case of Repetitive Text

Before nucleus sampling, practitioners working with neural language models had a dirty secret: the models could assign reasonable probabilities to tokens, but when asked to generate long text with beam search, the output often collapsed into repetitive loops. A model might generate "The cat sat on the mat. The cat sat on the mat. The cat sat on the mat." ad infinitum, or produce variations on the same sentence with slightly different word choices that converged back to the same phrases. Holtzman et al. [2020] gave this phenomenon a name – *neural text degeneration* – and provided a compelling diagnosis. The probability of any individual token in human text is, on average, much lower than the probability of the token selected by beam search. Human writers routinely choose words that are surprising in context – metaphors, unusual collocations, topic shifts – and it is precisely this surprise that makes text interesting to read. Beam search, by systematically selecting high-probability tokens, produces text that is technically fluent but lacks the variability and surprise of human writing. The repetition problem emerges because once a high-probability phrase is generated, the same phrase remains

high-probability in the updated context, creating a positive feedback loop. Nucleus sampling breaks this loop by allowing the model to sample from its own uncertainty: when the model is confident, the output is predictable; when the model is uncertain, the output is diverse. This simple insight changed the default decoding strategy for an entire field.

7.3.5 Temperature and the Quality-Diversity Tradeoff

Temperature is a single scalar τ that controls how "peaked" or "flat" the output distribution is, and it is orthogonal to the choice of decoding strategy – you can apply temperature scaling before greedy decoding, before beam search, before top-k, or before nucleus sampling. The temperature-scaled distribution divides the raw logits z_w by τ before applying softmax:

$$P_\tau(w \mid \text{context}) = \frac{\exp(z_w/\tau)}{\sum_{w'} \exp(z_{w'}/\tau)} \tag{7.4}$$

When $\tau = 1$, the distribution is unchanged. When $\tau < 1$ (low temperature), the logits are amplified: large logits become proportionally even larger, and the distribution sharpens. The highest-probability token concentrates more mass, and the output becomes more deterministic and repetitive. In the limit $\tau \to 0$, the distribution collapses to a point mass on the argmax token – temperature zero is greedy decoding. When $\tau > 1$ (high temperature), the logits are compressed: the gap between the highest and lowest logit shrinks, and the distribution flattens. More tokens become plausible, and the output becomes more diverse but less coherent. In the limit $\tau \to \infty$, all logits approach zero and the distribution becomes uniform over the vocabulary – every token is equally likely, and the output is random noise.

Temperature interacts with nucleus sampling in an important way. Lowering the temperature shrinks the nucleus (fewer tokens are needed to reach cumulative probability p) while raising it expands the nucleus. A common practical configuration is $\tau = 0.7$ or $\tau = 0.8$ combined with $p = 0.9$ or $p = 0.95$, which slightly sharpens the distribution before applying the adaptive truncation of nucleus sampling. This combination gives the practitioner two orthogonal dials: temperature controls the overall "creativity" of the distribution, and p controls how much of the tail is cut off. The quality-diversity tradeoff is fundamental to all text generation and cannot be eliminated, only navigated. Formal, factual text (legal documents, medical reports, translations) calls for low temperature and low p – stay close to the most probable output. Creative, exploratory text (stories, dialogue, brainstorming) calls for higher temperature and higher p – embrace the model's uncertainty and let surprising tokens through.

7.4 Evaluation of Generated Text

7.4.1 BLEU Score

How can we automatically measure whether a machine translation is any good, without asking a human to read it?

How do you know whether "The black cat sits on the rug" is a good translation of "Le chat noir est assis sur le tapis"? A human can tell at a glance, but human evaluation is expensive, slow, and difficult to reproduce. In 2002, Kishore Papineni and colleagues at IBM Research proposed an

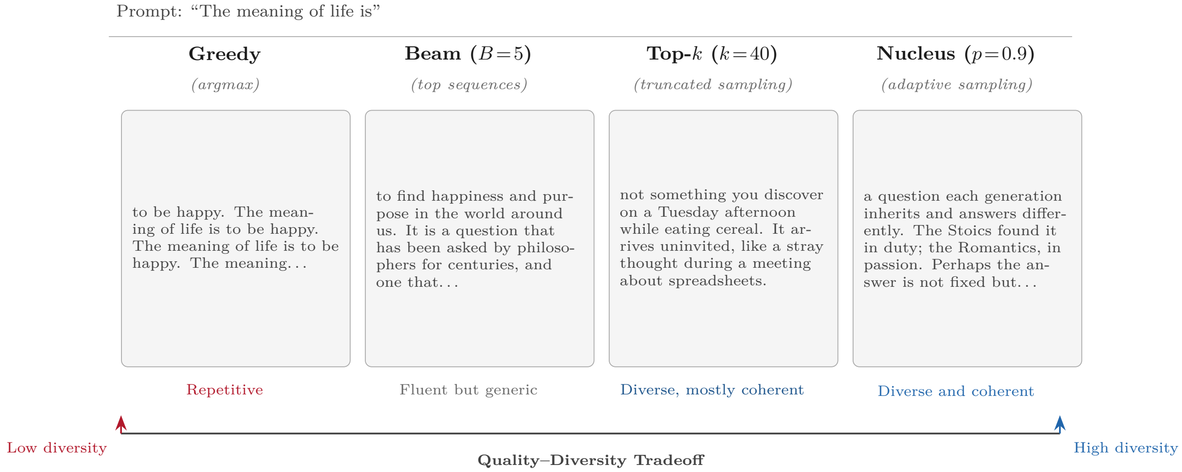

Figure 4: Figure 7.4 – Decoding strategies text comparison

automated metric that transformed the field: BLEU (Bilingual Evaluation Understudy). BLEU measures the overlap between a candidate (machine-generated) translation and one or more reference (human) translations, using n-gram precision with a brevity penalty. Despite its simplicity and known limitations, BLEU has been the dominant evaluation metric for machine translation for over two decades and remains the most widely reported number in MT research papers.

BLEU is built on *modified n-gram precision*. For each n-gram in the candidate translation, we count how many times it appears in both the candidate and the reference(s). The "modified" part is crucial: each n-gram's count in the candidate is clipped to its maximum count in any reference, preventing a degenerate candidate that consists of a single word repeated S times from achieving perfect unigram precision. Formally, the modified precision for n-grams of length n is:

$$p_n = \frac{\sum_{\mathbf{y}\in\hat{Y}} \sum_{\text{n-gram}\in\mathbf{y}} \min(C_{\text{pred}}(\text{n-gram}),\ C_{\text{ref}}(\text{n-gram}))}{\sum_{\mathbf{y}\in\hat{Y}} \sum_{\text{n-gram}\in\mathbf{y}} C_{\text{pred}}(\text{n-gram})} \tag{7.5}$$

where C_{pred} and C_{ref} are the counts of each n-gram in the candidate and reference, respectively, and the sums are taken over all sentences in the test corpus $\hat{Y}$. The final BLEU score combines modified precisions for $n = 1, 2, 3, 4$ via a weighted geometric mean and applies a brevity penalty BP that penalizes candidates shorter than the reference:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right), \quad \text{BP} = \min\left(1,\ \exp\left(1 - \frac{r}{c}\right)\right) \tag{7.6}$$

where $w_n = 1/N$ (equal weights, typically $N = 4$), r is the total reference length, and c is the total candidate length. The brevity penalty equals 1 when the candidate is at least as long as the reference and decreases exponentially as the candidate gets shorter, penalizing systems that achieve high precision by producing terse output.

Let us compute a concrete example. Reference: "The cat is on the mat." Candidate: "The cat on the mat." Unigram precision: the candidate has 5 unigrams, all of which appear in the reference, so

$p_1 = 5/5 = 1.0$. Bigram precision: the candidate bigrams are ("the cat", "cat on", "on the", "the mat"); of these, "the cat", "on the", and "the mat" appear in the reference but "cat on" does not, so $p_2 = 3/4 = 0.75$. Trigram precision: candidate trigrams are ("the cat on", "cat on the", "on the mat"); only "on the mat" matches, so $p_3 = 1/3 \approx 0.333$. Four-gram precision: candidate 4-grams are ("the cat on the", "cat on the mat"); neither matches, so $p_4 = 0/2 = 0$. Here we encounter a problem: $\log(0) = -\infty$, so the BLEU score is technically zero whenever any $p_n = 0$. This is common for individual sentences – corpus-level BLEU, which aggregates counts across all sentences before computing precisions, is much more stable and is the standard way BLEU is reported. The brevity penalty: $r = 6$, $c = 5$, so $\text{BP} = \exp(1 - 6/5) = \exp(-0.2) \approx 0.819$. If we ignore the zero 4-gram precision, the geometric mean of p_1 through p_3 is $\exp((0 + \log 0.75 + \log 0.333)/3) \approx 0.63$, and $\text{BLEU} \approx 0.819 \times 0.63 \approx 0.516$. This worked example illustrates that even a candidate missing a single word ("is") suffers a noticeable BLEU penalty.

```
import sacrebleu

# Example 1: close translation
refs = [["The cat is on the mat."]]
cand = ["The cat on the mat."]
bleu = sacrebleu.corpus_bleu(cand, refs)
print(f"Example 1 - BLEU: {bleu.score:.1f}")
print(f"  Precisions: {[f'{p:.1f}' for p in bleu.precisions]}")
print(f"  BP: {bleu.bp:.3f}, ratio: {bleu.sys_len}/{bleu.ref_len}")
# Example 1 - BLEU: 46.7
#   Precisions: ['100.0', '75.0', '50.0', '0.0']
#   BP: 0.819, ratio: 5/6

# Example 2: valid paraphrase, low BLEU
refs2 = [["The cat is on the mat."]]
cand2 = ["A feline rests upon the rug."]
bleu2 = sacrebleu.corpus_bleu(cand2, refs2)
print(f"\nExample 2 - BLEU: {bleu2.score:.1f}")
print(f"  Precisions: {[f'{p:.1f}' for p in bleu2.precisions]}")
# Example 2 - BLEU: 6.2
#   Precisions: ['33.3', '0.0', '0.0', '0.0']
# A perfectly valid translation scores near zero - BLEU's key limitation.
```

7.4.2 ROUGE and METEOR

BLEU asks: "are the n-grams in the candidate correct?" It is a precision-oriented metric. But precision alone tells an incomplete story. A candidate that produces a single correct n-gram and nothing else would achieve high precision but obviously be a terrible translation. Recall asks the complementary question: "are the n-grams in the reference captured by the candidate?" ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [Lin, 2004] measures exactly this. ROUGE-N computes the recall of reference n-grams in the candidate: $\text{ROUGE-N} = \frac{\text{number of reference n-grams appearing in candidate}}{\text{total number of reference n-grams}}$. ROUGE-1 (unigram recall) and ROUGE-2 (bigram recall) are most commonly reported. ROUGE-L uses the longest common subsequence (LCS) between candidate and reference, capturing sentence-level structure without requiring contiguous n-gram matches. ROUGE was originally developed for summarization evaluation, where recall matters more

than in translation: a good summary should capture the key information from the source document, and ROUGE measures whether the important reference n-grams appear in the generated summary.

METEOR (Metric for Evaluation of Translation with Explicit ORdering) [Banerjee and Lavie, 2005] attempts to address the limitations of both BLEU and ROUGE by incorporating synonym matching, stemming, and a penalty for word order differences. Where BLEU gives zero credit to a candidate that uses "feline" instead of "cat" (because the n-grams do not match literally), METEOR recognizes that "feline" and "cat" are synonyms and awards partial credit. METEOR computes a harmonic mean of precision and recall at the unigram level (with synonyms and stems counted as matches), then applies a fragmentation penalty that penalizes candidates whose matching unigrams appear in a different order than in the reference. In practice, METEOR correlates more strongly with human judgments than BLEU at the sentence level, though BLEU remains more widely used due to its simplicity and the massive body of existing results reported in BLEU.

7.4.3 Limitations of Automated Metrics

Every automated metric based on n-gram overlap shares a fundamental blind spot: it cannot distinguish between a fluent, meaningful sentence and a word salad that happens to contain the right n-grams. A candidate translation that scrambles the word order of the reference – producing "mat the on is cat the" – would achieve non-trivial unigram precision with BLEU despite being completely unintelligible. Conversely, a perfectly valid paraphrase that uses different vocabulary – "A feline rests upon the rug" for "The cat is on the mat" – scores near zero because the n-grams do not overlap. Callison-Burch et al. [2006] conducted a thorough analysis demonstrating that BLEU scores at the sentence level correlate only moderately with human judgments of translation quality, and that BLEU can systematically prefer a worse translation system over a better one when the systems use different lexical choices. The correlation improves substantially at the corpus level (averaging over hundreds or thousands of sentences), which is why BLEU is considered reliable for comparing systems on the same test set but unreliable for evaluating individual translations.

Additional failure modes compound the problem. Automated metrics are insensitive to factual correctness: "Paris is the capital of Germany" and "Paris is the capital of France" receive identical BLEU scores when compared to a reference about France, because the metrics measure only surface overlap, not semantic truth. They cannot assess discourse coherence: a paragraph of individually correct sentences in the wrong order receives the same n-gram scores as a well-organized paragraph. They penalize legitimate diversity in word choice: there are many ways to say the same thing, and metrics that require specific n-grams systematically undervalue creative or non-literal translations. The limitations of automated metrics are not merely academic – they have practical consequences for model development. If researchers optimize for BLEU during model selection and hyperparameter tuning, they implicitly bias their systems toward the specific lexical choices in the reference translations, potentially at the expense of fluency, naturalness, and diversity. This has motivated a sustained research effort toward better automated metrics, including BERTScore [Zhang et al., 2020], COMET [Rei et al., 2020], and other model-based metrics that compare representations rather than surface strings, and these newer metrics do correlate more strongly with human judgments, though none has fully replaced BLEU as the community standard.

7.4.4 Human Evaluation and LLM-as-Judge

Human evaluation remains the gold standard for assessing generation quality because humans can evaluate dimensions that no automated metric captures: adequacy (does the output convey the

correct meaning?), fluency (does it read naturally in the target language?), style (is the register appropriate?), and factual correctness (are the claims true?). Standard protocols include pairwise comparison (which of two outputs is better?), Likert-scale rating (rate this output from 1 to 5 on adequacy and fluency), and direct assessment (assign an absolute quality score). These protocols are expensive – evaluating a single translation system on a test set of a few hundred sentences requires dozens of annotator hours – and subject to inter-annotator variability, where different evaluators disagree on quality judgments, particularly for outputs that are adequate but not perfectly fluent or that make debatable stylistic choices.

The emerging paradigm of *LLM-as-judge* offers a compelling middle ground between the speed of automated metrics and the nuance of human evaluation. The idea is simple: provide a strong language model (such as GPT-4 or Claude) with the source, the reference, and the candidate, and ask it to judge the quality of the candidate along specified dimensions. Recent studies [Zheng et al., 2023] have shown that GPT-4 judgments correlate above 80% with human annotator judgments on translation and summarization tasks, approaching the inter-annotator agreement rate among humans themselves. LLM-as-judge is faster, cheaper, and more reproducible than human evaluation while capturing nuances that n-gram metrics miss entirely: a language model can recognize that "A feline rests upon the rug" is a valid paraphrase of "The cat is on the mat" even though the n-gram overlap is minimal. The approach is not without limitations – LLM judges exhibit their own biases, including a preference for verbose outputs, sensitivity to the order in which candidates are presented, and occasional confident errors on factual claims – but it represents the most promising direction in evaluation methodology since the introduction of BLEU two decades ago. We should be honest about the state of affairs: no evaluation method is fully satisfactory, and the field remains in search of a metric that is simultaneously cheap, fast, reproducible, and aligned with the full spectrum of human quality judgments.

7.5 Machine Translation as a Case Study

7.5.1 From Phrase-Based to Neural MT

Machine translation is where sequence-to-sequence models were born, where attention proved its worth, and where every evaluation metric in Section 7.4 was first validated. The history of MT provides the context necessary to appreciate why the encoder-decoder framework was such a breakthrough and why it swept away two decades of accumulated engineering virtually overnight. From the early 1990s through 2016, the dominant approach to machine translation was *statistical machine translation* (SMT), which decomposed the translation problem into a series of learned components: a phrase table mapping source phrases to target phrases (with associated probabilities estimated from parallel corpora), a language model scoring the fluency of candidate translations, a distortion model capturing word reordering patterns, and a decoder that searched over combinations of phrase translations to find the highest-scoring output. The phrase-based SMT system of Koehn et al. [2003] represented the mature form of this paradigm, and systems built on its architecture (particularly the open-source Moses system) dominated machine translation benchmarks and commercial applications for over a decade.

The neural revolution arrived in two waves. The first wave was Sutskever et al. [2014], who showed that a simple LSTM encoder-decoder – with no phrase tables, no explicit alignment model, no language model, and no hand-engineered features – could achieve competitive translation quality on English-French, reaching 34.8 BLEU on the WMT'14 benchmark. This result was shocking not

because 34.8 was the best score ever reported, but because a system with zero linguistic engineering matched systems that had been refined for years. The second wave was Bahdanau et al. [2015], whose attention mechanism (Chapter 6) eliminated the fixed-size bottleneck and pushed neural MT decisively past phrase-based systems on long sentences. The tipping point came in 2016, when Google replaced their production phrase-based translation system with a neural one (GNMT, [Wu et al., 2016]), reporting a 60% reduction in translation errors on some language pairs. Within two years, virtually every major technology company had followed suit. The transition was like the shift from horse-drawn carriages to automobiles: the old technology was competitive right up until the moment it was not, and the switch, once it began, was swift and total.

7.5.2 A Complete Translation Example

Let us trace through a complete translation to see how all the pieces of this chapter fit together. The source sentence is French: "Le chat noir est assis sur le tapis." The target sentence is English: "The black cat is sitting on the mat." The encoder – a bidirectional LSTM – processes the French tokens left-to-right and right-to-left, producing hidden states $\mathbf{h}_1$ through $\mathbf{h}_8$, one for each French token (including the period). Each hidden state encodes its token in the context of the full sentence. The decoder begins with the start-of-sequence token ⟨SOS⟩ and generates the English sentence one token at a time. At step 1, the decoder state $\mathbf{s}_0$ (initialized from the encoder) queries the encoder states via attention. The attention weights concentrate on $\mathbf{h}_1$ ("Le") and $\mathbf{h}_3$ ("noir"), and the decoder generates "The." At step 2, attention shifts to $\mathbf{h}_3$ ("noir") and the decoder generates "black." At step 3, attention focuses on $\mathbf{h}_2$ ("chat") and the decoder generates "cat."

Notice what happened between steps 2 and 3. In French, the adjective "noir" follows the noun "chat" – "le chat noir" is literally "the cat black." In English, the adjective precedes the noun: "the black cat." The attention mechanism handled this reordering automatically. At step 2, the decoder attended to the French word in position 3 ("noir") to generate the English word in position 2 ("black"). At step 3, it attended to position 2 ("chat") to generate the English word in position 3 ("cat"). No explicit reordering rules were programmed. The attention weights simply learned, from thousands of training examples, that English adjectives precede nouns while French adjectives often follow them. This automatic handling of word order differences across languages is one of the most compelling features of the attention-based encoder-decoder model and one of the primary reasons it superseded phrase-based systems that required explicit distortion models to handle reordering.

7.5.3 Attention Alignment in Translation

The attention weights from Section 7.5.2 can be visualized as a heatmap, with source tokens on one axis and target tokens on the other, and the intensity at position (i, j) representing the attention weight α_{ij} – how much decoder step i attended to encoder position j. For our French-to-English example, this heatmap reveals a striking pattern: a roughly diagonal structure interrupted by local reorderings. Most English words attend to French words in approximately the same position (because English and French have broadly similar word order for simple sentences), but the adjective-noun swap produces a visible off-diagonal crossing where "black" attends to the third French position and "cat" attends to the second. Function words like "the" and "on" tend to show more diffuse attention, distributing weight across several source positions rather than concentrating on a single one, which makes linguistic sense: function words carry relational rather than content meaning and draw their identity from the broader syntactic context.

These attention patterns are not perfect word alignments. Traditional word alignment in statistical

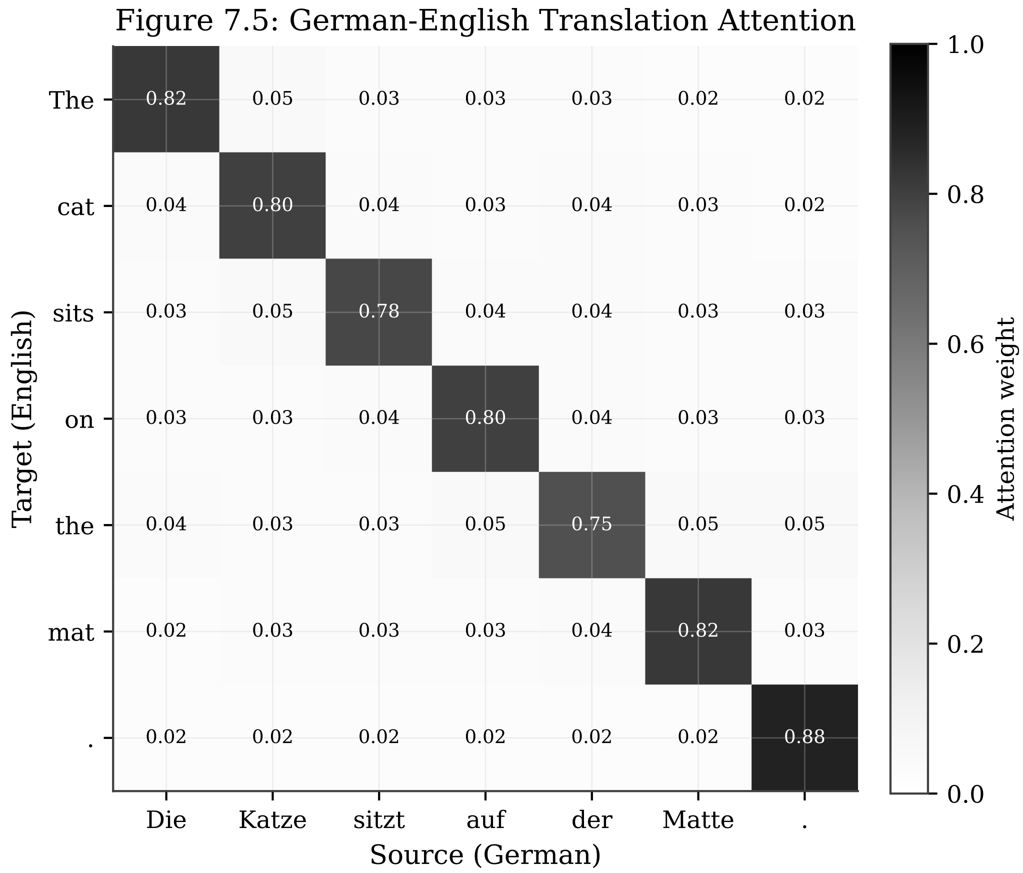

Figure 5: Figure 7.5 – Attention alignment heatmap for the French-to-English translation “Le chat noir est assis sur le tapis” to “The black cat is sitting on the mat

MT produced hard, one-to-one (or one-to-many) alignments; attention produces soft, many-to-many, fractional weights. A target word may attend to several source words simultaneously (as "sitting" attends to both "est" and "assis" because the concept of the progressive aspect requires both the auxiliary and the participle in French). A source word may receive attention from multiple target words (the French "le" receives attention from both "The" and "the" in the English output, though at different positions). This soft alignment is richer and more nuanced than traditional alignment, and it provides a form of interpretability that black-box neural models otherwise lack. By inspecting the attention heatmap, we can understand *why* the model generated each target word and diagnose errors by checking whether the model attended to the correct source positions. This diagnostic capability was one of the practical arguments that accelerated adoption of attention-based models in production translation systems.

7.5.4 The Transition to Transformers

We now have a complete sequence-to-sequence pipeline: an encoder that represents the input as a sequence of contextualized hidden states, an attention mechanism that bridges encoder and decoder by computing step-specific context vectors, a decoder that generates output tokens autoregressively, decoding strategies that turn probability distributions into actual text, and evaluation metrics that let us measure the quality of the result. Every component of this pipeline will carry forward into the Transformer architecture that we study in Chapter 8. But one major bottleneck remains, and it is the bottleneck that motivated the Transformer's existence: the recurrence. The encoder processes tokens sequentially – $\mathbf{h}_1$, then $\mathbf{h}_2$, then $\mathbf{h}_3$ – because each hidden state depends on its predecessor. For a 100-token sentence, the encoder requires 100 sequential LSTM steps. No amount of parallel hardware can compress these sequential steps, because each depends logically on the one before it. The decoder has the same limitation during inference (each token depends on all previously generated tokens), though during training with teacher forcing the computation can be parallelized.

The question that launched the Transformer is, in retrospect, obvious: if attention already gives us direct access to all positions, why do we need the sequential chain of hidden states that produced them? Self-attention, which we introduced in Section 6.4, allows every position to attend to every other position in a single parallel operation. If we replace the recurrent encoder with layers of self-attention, every position can be computed simultaneously, and the $\mathcal{O}(T)$ sequential bottleneck becomes $\mathcal{O}(1)$ depth (at the cost of $\mathcal{O}(T^2)$ pairwise comparisons, which GPUs handle efficiently for moderate sequence lengths). The encoder-decoder structure stays the same. Teacher forcing, beam search, nucleus sampling, BLEU – all the concepts from this chapter carry over unchanged. What changes is the engine: recurrence is replaced by self-attention, and the propulsion of the aircraft is upgraded from propeller to jet while the fuselage remains the same. The Transformer architecture that implements this idea is the subject of Chapter 8.

The power and limitations of attention-augmented recurrent models provide a natural transition to Chapter 8, where we introduce the Transformer architecture that replaces recurrence entirely with self-attention.

Exercises

Exercise 7.1 (Greedy as a special case of beam search). Prove that greedy decoding is a special case of beam search with beam width $B = 1$. Formally define both algorithms in terms of the

candidate set maintained at each step and the selection criterion, and show that they produce identical output for any input sequence and any model.

Exercise 7.2 (Temperature limiting behavior). Show that as the temperature parameter τ approaches 0, sampling from the temperature-scaled distribution $P_\tau(w) = \text{softmax}(z_w/\tau)$ converges to greedy decoding (argmax over the vocabulary). Then show that as $\tau \to \infty$, the distribution converges to the uniform distribution $P(w) = 1/|V|$ for all w. Derive both limiting cases from the softmax formula, and explain the implications for the quality-diversity tradeoff.

Hint: For the $\tau \to 0$ case, consider what happens to the ratio z_i/τ when z_i is the largest logit versus when it is not.

Exercise 7.3 (Beam search complexity). Analyze the time complexity of beam search as a function of beam width B, vocabulary size $|V|$, and target sequence length S. Express the total number of score computations. Compare with greedy decoding and explain at what point increasing B yields diminishing returns. Your analysis should address both the scoring step (computing $B \cdot |V|$ candidate scores) and the selection step (finding the top B candidates).

Parameter	Typical Value	Your Analysis
B	1–20	
$\|V\|$	30,000–50,000	
S	10–100	
Total score computations	?	

Exercise 7.4 (Beam search with a pre-trained model). Using a pre-trained HuggingFace translation model (e.g., `Helsinki-NLP/opus-mt-en-de`), translate the following five English sentences into German using (a) greedy decoding (`num_beams=1`) and (b) beam search (`num_beams=5`): (1) "The weather is nice today." (2) "Machine translation has improved dramatically in recent years." (3) "The committee decided to postpone the vote until further notice." (4) "She opened the window and let the fresh air fill the room." (5) "Despite the rain, thousands of fans gathered outside the stadium." Compare the outputs qualitatively and compute BLEU scores against reference translations using `sacrebleu`.

Exercise 7.5 (Sampling strategies comparison). Implement top-k sampling and nucleus sampling from scratch in Python using PyTorch. Given a pre-trained GPT-2 model from HuggingFace, generate 10 continuations of the prompt "The meaning of life is" using: (a) greedy decoding, (b) beam search with $B = 5$, (c) top-k with $k = 50$, and (d) nucleus sampling with $p = 0.9$. For each method, generate 50 tokens. Rate each continuation on a 1-to-5 scale for both quality (grammaticality, coherence) and diversity (novelty, surprise). Report the mean and standard deviation of your ratings for each method.

Exercise 7.6 (BLEU analysis). Compute BLEU scores for a set of 20 English-German translation pairs (you may use any parallel test set or create your own). Plot BLEU score versus sentence length (in tokens). Identify at least three cases where BLEU disagrees with your own judgment of translation quality – cases where a good translation receives a low BLEU score or a mediocre translation receives a high one. For each case, explain why BLEU and your judgment diverge. Compute the Pearson correlation between your human ratings (on a 1-to-5 adequacy scale) and the sentence-level BLEU scores.

Exercise 7.7 (Sequence reversal with attention). Build a minimal encoder-decoder model with Bahdanau attention for a toy task: reversing a sequence of digits (e.g., “1 2 3 4” → “4 3 2 1”). Use a 1-layer LSTM encoder and decoder with hidden dimension 64. Train with teacher forcing on sequences of length 5 to 10 for 50 epochs. Test with greedy decoding on held-out sequences of lengths 5, 10, and 15. Visualize the attention weights for at least two test examples and verify that the model learns to attend in reverse order (the attention pattern should approximate an anti-diagonal). Report accuracy (exact sequence match) at each test length.

Exercise 7.8 (Teacher forcing versus scheduled sampling). Using the sequence reversal model from Exercise 7.7, train two versions: one with pure teacher forcing and one with scheduled sampling (with ϵ decaying linearly from 1.0 to 0.1 over 50 epochs). Compare: (a) training loss curves for both models, (b) test accuracy on sequences of length 10 and 20, (c) robustness to errors: inject one random error into the first three tokens of the decoder input and measure how well each model recovers to produce the correct remaining tokens. Discuss whether your results are consistent with the exposure bias hypothesis.

References

- Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *Proceedings of ICLR 2015*.
- Banerjee, S., & Lavie, A. (2005). METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and Summarization*, 65–72.
- Bengio, S., Vinyals, O., Jaitly, N., & Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in NeurIPS*, 1171–1179.
- Callison-Burch, C., Osborne, M., & Koehn, P. (2006). Re-evaluating the role of BLEU in machine translation research. In *Proceedings of EACL*, 249–256.
- Cho, K., van Merrienboer, B., Gulcehre, C., et al. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of EMNLP*, 1724–1734.
- Fan, A., Lewis, M., & Dauphin, Y. (2018). Hierarchical neural story generation. In *Proceedings of ACL*, 889–898.
- Holtzman, A., Buys, J., Du, L., Forbes, M., & Choi, Y. (2020). The curious case of neural text degeneration. In *Proceedings of ICLR*.
- Koehn, P., Och, F. J., & Marcu, D. (2003). Statistical phrase-based translation. In *Proceedings of NAACL-HLT*, 48–54.
- Koehn, P., & Knowles, R. (2017). Six challenges for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, 28–39.
- Lin, C.-Y. (2004). ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, 74–81.
- Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). BLEU: A method for automatic evaluation of machine translation. In *Proceedings of ACL*, 311–318.
- Raffel, C., Shazeer, N., Roberts, A., et al. (2020). Exploring the limits of transfer learning with a unified text-to-text Transformer. *JMLR*, 21(140), 1–67.
- Ranzato, M., Chopra, S., Auli, M., & Zaremba, W. (2016). Sequence level training with recurrent neural networks. In *Proceedings of ICLR*.
- Rei, R., Stewart, C., Farinha, A. C., & Lavie, A. (2020). COMET: A neural framework for

MT evaluation. In *Proceedings of EMNLP*, 2685–2702.
- Shen, S., Cheng, Y., He, Z., He, W., Wu, H., Sun, M., & Liu, Y. (2016). Minimum risk training for neural machine translation. In *Proceedings of ACL*, 1683–1692.
- Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 3104–3112.
- Williams, R. J., & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2), 270–280.
- Wu, Y., Schuster, M., Chen, Z., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.
- Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., & Artzi, Y. (2020). BERTScore: Evaluating text generation with BERT. In *Proceedings of ICLR*.
- Zheng, L., Chiang, W.-L., Sheng, Y., et al. (2023). Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in NeurIPS*.