Recurrent Networks: Language Models, Gates, and Translation

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

---

Prerequisites: ***NN_BASICS*** *(training loop, softmax + cross-entropy, vanishing gradients) and the* ***CNN*** *note (weight sharing). Optional companion:* ***RNN_ADVANCED****.*

A recurrent network reads a sequence one step at a time while carrying a memory. We use it as a **language model** (§1–§6) and as a Spanish→English **translator** (§7–§9).

1 Why Sequences Need Memory

The data are not independent any more. In every earlier model the datapoints were IID: the order of the rows of X did not matter. In a sequence it is everything. A weather forecast is good for tomorrow and poor for next month; tomorrow's stock price depends on today's; the next word of a sentence depends on all the words before it. What we want is the probability of a whole sequence, and the chain rule of probability writes it as a product of next-step predictions:

$$p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}). \tag{1}$$

In words: to model a sequence, it is enough to predict each step from everything before it. The problem is that "everything before it" grows with t.

Plan A: limited memory (Markov). Look back only τ steps, $p(x_t \mid x_{t-\tau}, \ldots, x_{t-1})$. With $\tau = 1$ and words this is a *bigram* model: count how often each word follows each other word. It works for short-range patterns and fails for anything further back than τ ("she" three sentences after "Lily"). Making τ large is hopeless: the number of possible histories grows like $|\mathcal{V}|^\tau$.

Plan B: a summary that is carried along. Keep a fixed-size vector H_t, the **hidden state**, that summarizes the history, and update it with each new input:

$$H_t = f(H_{t-1}, X_t), \qquad p(x_{t+1} \mid x_1, \ldots, x_t) \approx g(H_t).$$

The hidden state is "the context at time t" (your board). The same function f is used at every step, so the model has a fixed number of parameters however long the sequence is. An RNN is Plan B with f and g being small neural networks.

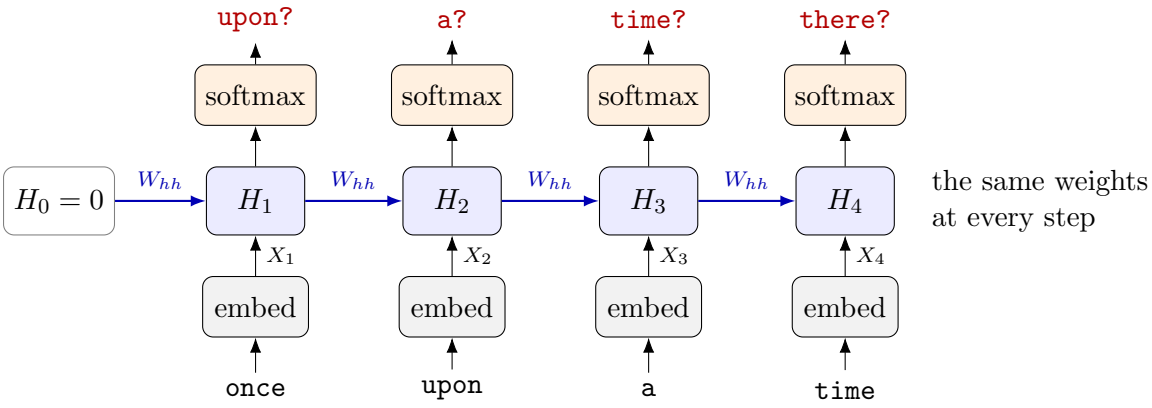


Figure 1: The big picture: an RNN language model unrolled over the start of a TinyStories story. At each step it reads one word, updates its hidden state (blue arrows: the memory), and predicts the next word. The targets (red) are the inputs shifted by one.

Almost everything here is reused. Table 1 lists the parts of this note and where you have seen each one.

Part	**Where you have seen it**	**What changes**
Next-word prediction = softmax over the vocabulary, cross-entropy loss	HW5's 10-class classifier	5,274 classes (the words), one prediction per position
Embedding lookup (§2)	a linear layer on a one-hot input (NN_BASICS)	a table lookup instead of a product
The cell $H_t = \tanh(X_t W_{xh} + H_{t-1} W_{hh} + b)$ (§4)	a dense layer with tanh	it also reads its own previous output
Weight sharing (§4)	CNN: the same kernel at every pixel	the same matrices at every *time step*
Vanishing / exploding gradients (§5)	NN_BASICS: a product of per-layer factors	the factors come from time steps, often 50+ of them
Hidden-state model (§1)	HMMs (EM_ADVANCED)	the state is a deterministic vector, not a random one
Gradient clipping (§5)	**NEW** (one line)	rescale the gradient when its norm is too large
Gates: GRU, LSTM (§6)	**NEW**	learned switches that decide what the memory keeps
Encoder–decoder, teacher forcing, decoding, BLEU (§7–§9)	**NEW**	a second network, generated one word at a time

Table 1: The parts of this note. Six are reused from earlier in the course.

2 From Text to Numbers: Tokens, IDs, Embeddings, Targets

A network multiplies numbers, so text has to become numbers first. Four steps, shown on one sentence of the TinyStories data that HW6 uses (short children's stories written with a small vocabulary).

1. **Tokenize**: cut the text into pieces. Here the pieces are lower-cased words and punctuation marks: `"Lily's dog ran."` becomes `lily's | dog | ran | .`, and every story ends with an `<eos>` (end-of-sequence) token.

2. **Vocabulary and IDs**: give each distinct token an integer. Words that occur fewer than 10 times in the training stories are replaced by `<unk>` (unknown). That leaves a vocabulary $\mathcal{V}$ of $|\mathcal{V}| = 5{,}274$ tokens, and a story becomes a list of IDs.

3. **Embedding**: look up a learned vector for each ID. An embedding table $E \in \mathbb{R}^{|\mathcal{V}| \times d}$ has one row per token; ID i becomes row i. This is exactly a linear layer applied to a one-hot vector (a row with a single 1 at position i), $\text{onehot}(i)\, E = E_{i,:}$, without the wasteful multiplication by zeros. The rows are trained like any other weights.

4. **Targets = inputs shifted by one.** Training windows are cut from the long stream of IDs. For a window $x_1, \ldots, x_T$ the input at position t is x_t and the target is x_{t+1}: *every* position is a training example (Figure 1).

In HW6 the windows are 64 tokens long and start at random places in the training stream; each window starts from a fresh zero state. (The alternative, consecutive windows that carry the state over, is in RNN_ADVANCED.)

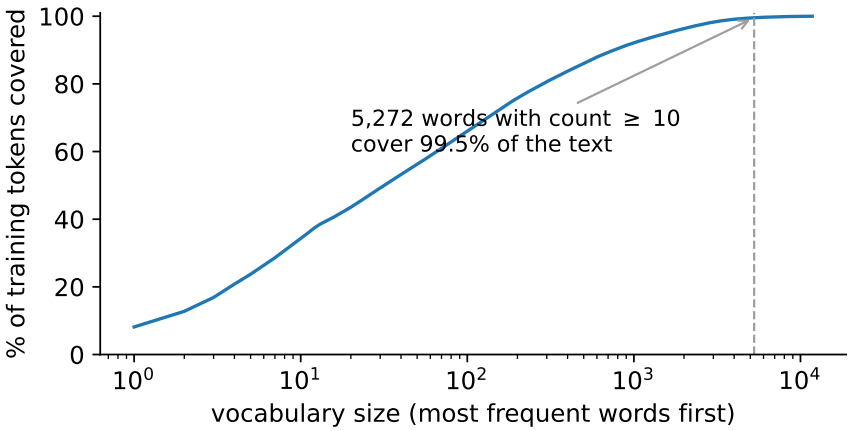


Figure 2: TinyStories was built with a small vocabulary: the 5,272 words that occur at least 10 times cover 99.6% of the training text, and only 0.6% of the validation tokens become `<unk>`.

Key point: Whole words are fine here — real language models use subwords. TinyStories is unusual: a few thousand words cover almost everything (Figure 2). On real text a word vocabulary either explodes (every new spelling or word form needs its own entry) or turns many tokens into `<unk>`, and the model can neither read nor write a word it has never seen (a new product name, a hashtag, a chemical compound). Cutting into characters avoids `<unk>` but makes sequences several times longer and forces the network to learn spelling. Modern language models sit in between, with **subwords**: frequent words stay whole, rare ones are built from frequent pieces, and the pieces are learned from data (*byte-pair encoding*, the first topic of Week 13, TOKENS note).

3 Language Models and Perplexity

A **language model** assigns a probability to the next token given the ones before it, i.e. it estimates each factor of Equation (1). It is a classifier with $|\mathcal{V}|$ classes, applied at every position, and it is trained with the usual cross-entropy, averaged over the T positions:

$$\mathcal{L} = -\frac{1}{T}\sum_{t=1}^{T} \log p(x_t \mid x_{<t}).$$

The loss in nats is hard to picture, so language models report its exponential, the **perplexity**:

$$\mathrm{PPL} = \exp(\mathcal{L}) = \exp\Big(-\frac{1}{T}\sum_{t} \log p(x_t \mid x_{<t})\Big). \tag{2}$$

Meaning: perplexity is the effective number of words the model is choosing between at each step. A model that spreads its probability uniformly over k words has perplexity exactly k. Guessing uniformly over the whole vocabulary gives PPL $= |\mathcal{V}| = 5{,}274$; a perfect model gives 1. Because it is an *average* per token ("exp of the entropy", your board), perplexities are comparable across texts of different lengths — but only with the same vocabulary and tokenization.

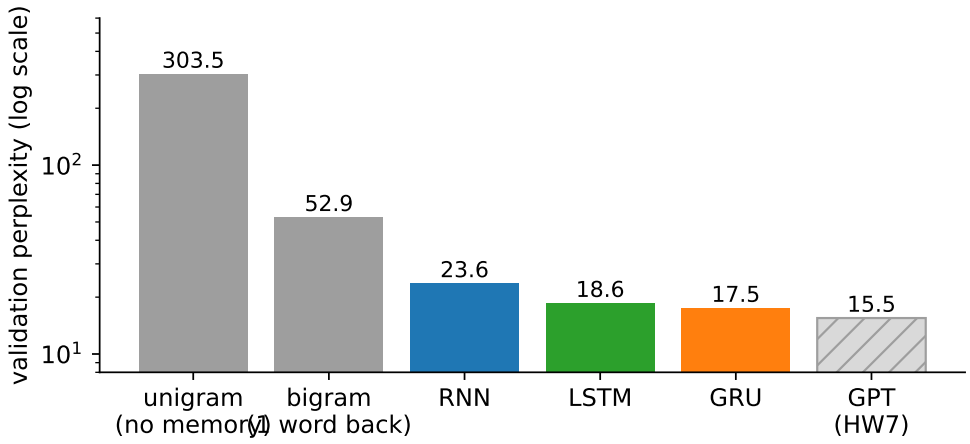


Figure 3: The perplexity ladder on HW6's validation stories (reference run). Each rung uses more of the past. The GPT bar is HW7's decoder-only Transformer on the same data, for later.

Worked example: the ladder, rung by rung. **Unigram** (no memory: predict every word by its frequency): perplexity about 303.5, loss $\ln 303.5 = 5.715$ nats. **Bigram** (one word back, mixed with the unigram so unseen pairs keep some probability): about 53. An **RNN** that reads the whole window: about 23.8 (loss 3.17). The gated GRU and LSTM of §6: about 17.5 and 18.6. Each rung uses more of the past, and the perplexity drops by a factor each time. (This ladder is the sense in which a "model can actually do something": from choosing among 300 words to choosing among 18.)

4 The Recurrent Cell

Shapes. The *model* takes token IDs as a batch-first (n, T) tensor and returns batch-first logits $(n, T, |\mathcal{V}|)$; only the *recurrent core* in between is **time-major**, as in the d2l book: a batch of n windows of length T

is T matrices $X_1, \ldots, X_T$, each $n \times d$ (one embedded token per row, as in the course's "rows = examples" convention). The hidden state H_t is $n \times h$.

The cell. One step of a (plain, "vanilla") RNN is a dense layer that reads two things, the current input and its own previous state:

$$\boxed{H_t = \tanh\big(X_t W_{xh} + H_{t-1} W_{hh} + b_h\big), \qquad O_t = H_t W_{hq} + b_q} \tag{3}$$

with shapes $X_t\,(n \times d)$, $W_{xh}\,(d \times h)$, $H_{t-1}\,(n \times h)$, $W_{hh}\,(h \times h)$, $H_t\,(n \times h)$, $W_{hq}\,(h \times |\mathcal{V}|)$ and logits $O_t\,(n \times |\mathcal{V}|)$. **Meaning:** the new state mixes what just came in ($X_t W_{xh}$) with what was remembered ($H_{t-1} W_{hh}$), squashed into $(-1, 1)$ by tanh; the output layer turns the state into scores for the next token, and the softmax of the scores is the prediction. $H_0 = 0$.

Three weight matrices, independent of t (your board). W_{xh}, W_{hh}, W_{hq} are the same at every step: the network is *shared over time* exactly as a CNN's kernel is shared over space. This is why one RNN handles any length. It is also "not deep": one layer, applied T times.

Worked example: one step by hand (the HW checkpoint). $d = h = 2$, $X_t = (1, 0)$, $H_{t-1} = (0.5, -0.5)$, $W_{xh} = \left(\begin{smallmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{smallmatrix}\right)$, $W_{hh} = 0.5\,I$, $b_h = 0$. Then $X_t W_{xh} = (0.1,\ 0.2)$ (the first row of W_{xh}), $H_{t-1} W_{hh} = (0.25,\ -0.25)$, the sum is $(0.35,\ -0.05)$, and $H_t = \tanh(0.35, -0.05) = (0.3364,\ -0.0500)$.

Algorithm: RNN language model, forward pass

1. $H_0 \leftarrow 0$ (shape $n \times h$).
2. For $t = 1, \ldots, T$: $H_t \leftarrow \tanh(X_t W_{xh} + H_{t-1} W_{hh} + b_h)$; keep H_t.
3. Stack the states, (T, n, h); logits $O = H W_{hq} + b_q$, $(T, n, |\mathcal{V}|)$.
4. Loss = cross-entropy of the logits against the shifted targets, averaged over all $T \cdot n$ positions.

The library version. `nn.RNN` computes Equation (3) with two bookkeeping differences: it stores the weights transposed (`weight_ih_l0` is $W_{xh}^\top$, of shape $h \times d$) and it has two bias vectors, `b_ih + b_hh`, whose sum plays the role of b_h. Copy the weights across with that mapping and the outputs agree to rounding. Trained from *independent* random starts, your RNN and `nn.RNN` should reach similar, not identical, perplexities: the same recurrence, different random starting points.

Generating text. After training, the model writes a story by sampling: feed a prompt, take the logits o at the last position, draw the next word from $\mathrm{softmax}(o/\tau)$, append it, repeat. The **temperature** τ reshapes the distribution: $\tau < 1$ sharpens it (safer, more repetitive), $\tau > 1$ flattens it (more varied, more mistakes), and $\tau \to 0$ is always taking the top word. For logits $o = (2, 1, 0)$ over three words: $\tau = 1$ gives $(0.665, 0.245, 0.090)$, $\tau = 0.5$ gives $(0.867, 0.117, 0.016)$, $\tau = 2$ gives $(0.506, 0.307, 0.186)$. The temperature divides the *logits*, before the softmax.

Remark. Bridge to the lecture slides. The STAT157 slides write column vectors, $h_t = \phi(W_{hh} h_{t-1} + W_{hx} x_{t-1} + b_h)$, so their W_{hx} is our $W_{xh}^\top$ and their output matrix W_{ho} is our W_{hq}; they also index the input one step earlier. Same model, different bookkeeping.

5 Training Through Time: Vanishing and Exploding Gradients, Clipping

Backpropagation through time (BPTT) is ordinary backpropagation on the unrolled graph of Figure 1: T copies of the cell, chained by $H_{t-1} \to H_t$. Because the weights are shared, each copy contributes to the same gradient, and BPTT adds up the contributions of all steps.

Why long-range gradients die. How much does the loss at the last step depend on the input d steps back? The chain rule passes through every state in between:

$$\frac{\partial H_T}{\partial H_{T-d}} = \prod_{k=T-d+1}^{T} \frac{\partial H_k}{\partial H_{k-1}}, \qquad \frac{\partial H_k}{\partial H_{k-1}} = \operatorname{diag}\big(\tanh'(\cdot)\big)\, W_{hh}^{\top}.$$

It is the NN_BASICS argument again, with time steps in place of layers: a **product of d per-step factors**. If the factors shrink a vector, the product shrinks exponentially; if they stretch it, it explodes. With a factor of 0.9 per step: $0.9^{20} = 0.12$, $0.9^{50} = 0.005$. With 1.1: $1.1^{20} = 6.7$, $1.1^{50} = 117$. And since $\tanh' \le 1$, saturated units push the factors down further. Your board: an RNN does not "deal well with long-term dependencies".

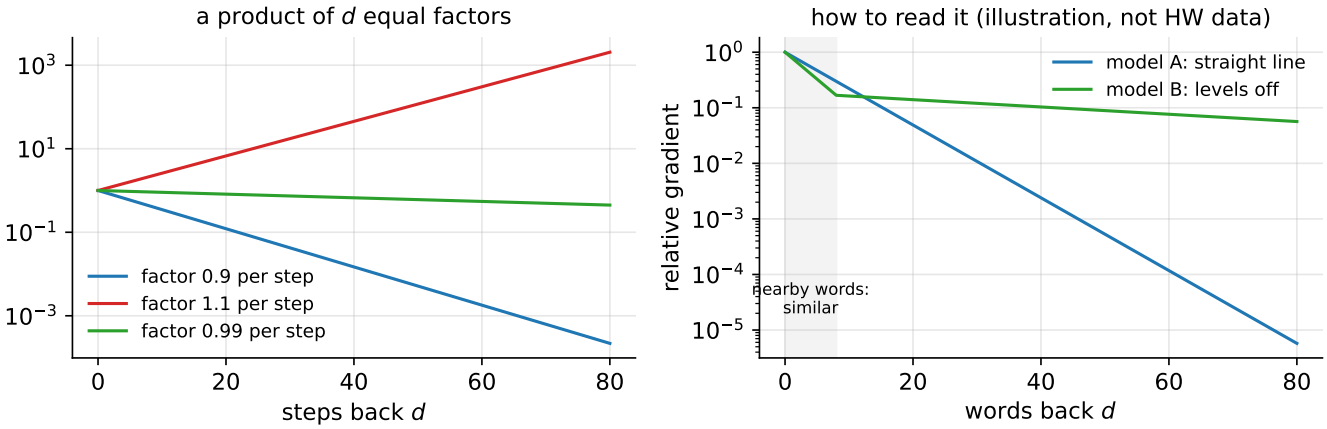


Figure 4: Left: a product of d equal factors is a straight line on a log scale — down for 0.9, up for 1.1, nearly flat for 0.99. Right: an *illustration* of the two shapes to look for in a measured curve (not HW data): a model whose gradient keeps falling on a straight line, and one that drops at first and then levels off.

What HW6 measures. The provided experiment asks: how much does the *last* prediction of a window depend on each earlier word? It makes every input embedding in the window a tensor that gradients flow into, computes the loss at the last position only, and backpropagates. The gradient that reaches the embedding d words before the end measures that word's influence. Its size (the norm over the embedding coordinates) is averaged over many windows and divided by the value at $d = 0$, so every model starts at 1 and the curves are comparable.

How to read the curve (Figure 4). On a log scale, a product of per-step factors is a *straight line*, and its slope is the log of the factor. So: (1) where do the models still agree? Nearby words matter most for every model; (2) which curve keeps falling on a straight line, as a plain product would, and which ones bend and level off, so that a nearly-constant path carries the signal further? (3) at what distance do

they become clearly different, and by how many powers of ten? A word whose gradient is 10^{-5} of the last word's gives training essentially nothing to learn from. Read these three things off *your* plot.

Exploding gradients: clipping. When the factors are larger than 1, one bad step can produce a gradient so large that a single update wrecks the weights. The fix is to cap the gradient's *length* and keep its direction:

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}, \tag{4}$$

where $\mathbf{g}$ is the gradient of *all* parameters stacked into one vector and θ is a threshold (1 in HW6). **Meaning:** if the gradient is shorter than θ nothing happens; otherwise it is shrunk to length exactly θ, pointing the same way. Example: $\mathbf{g} = (3, 4)$ has norm 5; with $\theta = 1$ it becomes $(0.6, 0.8)$; with $\theta = 10$ it is unchanged. Clipping goes between `backward()` (which computes $\mathbf{g}$) and the optimizer step (which uses it).

Key point: Clipping fixes exploding gradients, not vanishing ones. An exploding gradient has a useful direction and a bad size, and clipping fixes the size. A vanishing gradient has (almost) no information from far back to begin with; rescaling it would mostly amplify the nearby words, which dominate the sum. Vanishing needs a change of architecture: the gates of the next section.

Truncated BPTT. Backpropagating through a whole book is infeasible, so in practice the stream is cut into windows (64 tokens in HW6) and gradients flow only inside a window. Your board's verdict: a fixed truncation "works!" — and it is what every RNN in HW6 does.

6 Gates: the GRU and the LSTM

The idea. Not every step of a sequence is equally relevant. "The cat, which had been chasing the mouse all morning and was now very tired, *was* asleep": the verb must agree with *cat*, ten words back, and the words in between should not overwrite that memory. A plain RNN rewrites its whole state at every step. A **gated** RNN computes, at every step, numbers between 0 and 1 (sigmoids) that decide how much of the old state to keep and how much new information to write. Gates are learned switches; they are ordinary dense layers followed by σ, so backprop trains them like everything else.

The GRU (gated recurrent unit), in PyTorch's convention (the one `nn.GRU` uses):

$R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r)$	reset gate: how much of H_{t-1} to use in the proposal
$Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z)$	update gate: how much of H_{t-1} to keep
$\tilde{H}_t = \tanh\left(X_t W_{xh} + R_t \odot (H_{t-1} W_{hh} + b_{hh}) + b_{xh}\right)$	candidate: a proposed new state
$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \tilde{H}_t$	keep the old, or take the new

$\odot$ is the element-wise (Hadamard) product. **Meaning:** the candidate $\tilde{H}_t$ is what a plain RNN would write; the update gate mixes it with the old state, unit by unit. Where $Z_t \approx 1$ the unit simply copies its old value forward. The candidate is a proposal (a tanh vector), not a probability distribution.

The LSTM (long short-term memory) keeps two vectors, a **memory cell** C_t and the output state H_t, and three gates:

$I_t,\ F_t,\ O_t = \sigma(X_t W_{x\cdot} + H_{t-1} W_{h\cdot} + b_\cdot)$	input, forget, output gates (each its own weights)
$\tilde{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c)$	candidate memory
$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$	forget some old memory, add some new
$H_t = O_t \odot \tanh(C_t)$	expose part of the memory

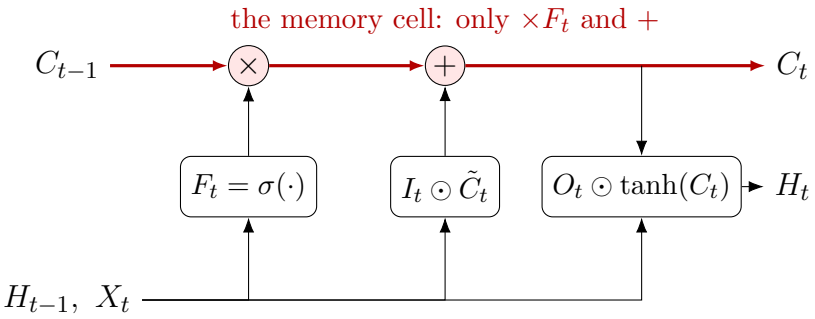


Figure 5: The LSTM cell. Along the memory cell (red) the old memory is only multiplied by the forget gate and added to: no weight matrix and no squashing on the way.

Key point: Why gates let gradients reach further back. Along the memory cell, one step back multiplies the gradient by F_t — a number between 0 and 1 that the network *learns*, with no W_{hh} and no $\tanh'$ in between, and the update is *additive*. When the network needs to remember, it keeps F_t near 1: $0.99^{50} = 0.61$ and $0.95^{50} = 0.077$, against $0.9^{50} = 0.005$ for a shrinking plain-RNN factor. That is the flatter LSTM and GRU curves of Figure 4. (The GRU's Z_t plays the same role: where $Z_t \approx 1$, $H_t \approx H_{t-1}$.)

Cost. A GRU layer has 3 copies of the RNN's weights (two gates + the candidate), an LSTM layer 4 (three gates + the candidate). With HW6's sizes (embedding 256, 512 hidden units) the recurrent layer grows from about 0.39M parameters (RNN) to 1.18M (GRU) and 1.58M (LSTM); the embedding and output tables (about 4M) are the same for all three.

What to expect in HW6 (the same data, steps and evaluation for all three): both gated models should end clearly below the plain RNN in perplexity. Between the GRU and the LSTM, do not expect a fixed order: at a short budget either can be ahead, and nothing in the theory says which of them wins on a given task. Generated stories are worth reading for what carries over from sentence to sentence (a character, a scene) and where a story contradicts itself, but one sample per model is noisy; the perplexity table is the firm evidence.

7 The Encoder–Decoder

The task. Translate a Spanish sentence into English: *¿dónde está la estación?* $\rightarrow$ *where is the station?* Input and output are both sequences, of different lengths, and words do not line up one-to-one. One RNN reading and writing at the same time cannot handle that. Use two.

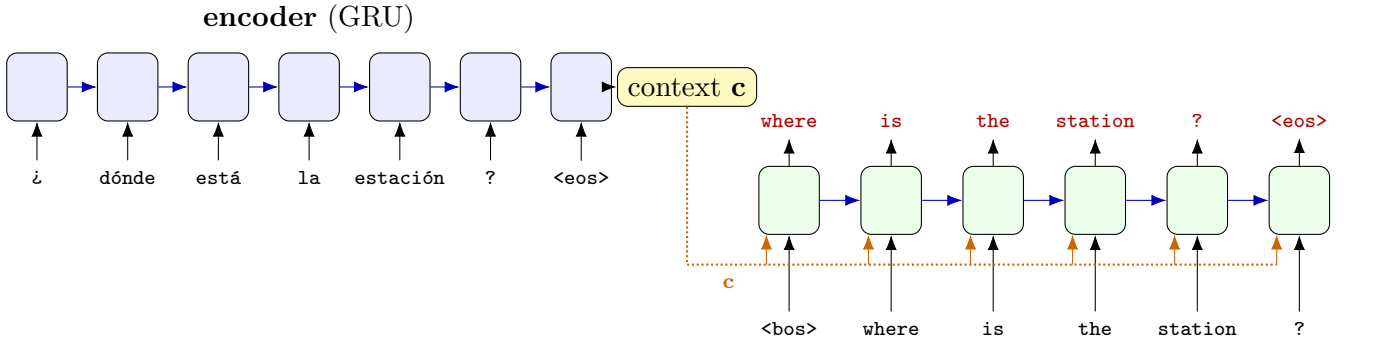


Figure 6: HW6's translator. The encoder reads the Spanish sentence and its final state becomes the **context c**. The decoder starts from that state, reads **c** again at every step (dotted), and predicts the English sentence one token at a time.

- **Encoder**: embedding + a GRU over the Spanish tokens. "Variable-length sequence ⇒ a fixed-size hidden state" (your board). Its final top-layer state is the **context c**, a summary of the whole source sentence. (HW6's starter runs the GRU only over each sentence's real tokens, so the final state is taken right after that sentence's own `<eos>`, not after padding.)
- **Decoder**: a second, *different* GRU with its own weights. Its state starts as the encoder's final state, and at every step its input is the previous English word's embedding *concatenated with* **c**. A linear layer turns its output into scores over the English vocabulary.

Key point:

Shapes through the translator (n sentences, source length T, target length T', embedding E, hidden size H, $L = 2$ GRU layers):

$$\underbrace{(n, T)}_{\text{IDs}} \xrightarrow{\text{embed, transpose}} \underbrace{(T, n, E)}_{\text{GRU input}} \xrightarrow{\text{GRU}} \underbrace{(T, n, H)}_{\text{top-layer outputs}} \quad , \quad \underbrace{(L, n, H)}_{\text{final state, one per layer}} \quad \rightarrow \mathbf{c} : \underbrace{(n, H)}_{\text{top layer's}}$$

The decoder's GRU starts from the (L, n, H) state, reads $(T', n, E + H)$ (each target embedding next to **c**), and its outputs become logits $(n, T', |\mathcal{V}_{en}|)$. With two layers, layer 2 reads layer 1's outputs at every step; each layer keeps its own state.

Key point: The decoder is the language model of §4, conditioned on a source. It predicts the next English word from the English words so far — exactly the TinyStories task — with the context **c** as an extra input at every step. Everything you know about language models (shifted targets, softmax over the vocabulary, cross-entropy, sampling) carries over.

Data (given): Tatoeba's Spanish–English pairs, lower-cased, punctuation split off, at most 12 tokens per side; 100,000 training pairs and 2,000 test sentences never seen in training, each scored against all of its English translations. Special tokens: `<bos>` (begin), `<eos>` (end), `<pad>` (padding to make a batch rectangular). The interface follows the d2l book: an `Encoder`, a `Decoder` with `init_state` (turn the encoder's output into the decoder's starting state), and an `EncoderDecoder` that chains them.

8 Training: Teacher Forcing and Padding Masks

Training and generation are different computations. This is the part of the topic that confuses most people, so look at Figure 7 before the details.

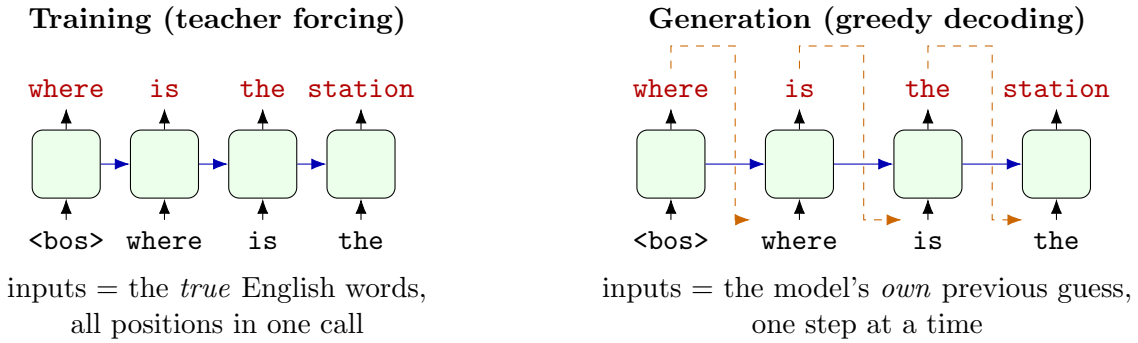


Figure 7: The same decoder, used two ways. Left: in training the true previous word is fed in, so all inputs are known in advance and go into one library call (inside it, the recurrence still runs step by step). Right: when translating a new sentence there is no true English, so each predicted word (dashed) becomes the next input.

Teacher forcing. In training we know the whole English sentence, so the decoder's input is `<bos>` followed by the true English words, and the target is the same words followed by `<eos>` — the input shifted by one, exactly as in the language model. The batch format: `tgt_input` = `<bos> where is the station ?`, `label` = `where is the station ? <eos>`. Because every input is known in advance, the whole target goes into one GRU call: training is as fast as the language model's. (The GRU still steps through time inside that call; what teacher forcing removes is the wait for the model's own guesses.)

Padding and the masked loss. Sentences in a batch have different lengths, so shorter ones are filled with `<pad>` up to the longest. Positions whose *label* is `<pad>` are not real predictions and must not count in the loss or its average.

Worked example: a padded batch of 2. Labels (4 positions): row 1 is `i see . <eos>`, row 2 is `hi <eos> <pad> <pad>`. Suppose the per-position cross-entropies are $(0.5, 1.0, 0.25, 0.25)$ and $(2.0, 0.5, 3.0, 3.0)$. The mask is $(1, 1, 1, 1)$ and $(1, 1, 0, 0)$. The correct loss is the average over the 6 real positions: $(0.5 + 1 + 0.25 + 0.25 + 2 + 0.5)/6 = 0.75$. The plain average over all 8 positions would be 1.31, dominated by the meaningless padding scores. In PyTorch the whole mask is one argument of the loss, `ignore_index` set to the padding ID.

A consistency check that catches real bugs. Run the decoder twice on the same target: once with teacher forcing (the whole target in one call) and once one token at a time, passing the state along. Give both runs the same starting state, the same supplied target tokens (not the model's guesses), and evaluation mode (dropout off: two runs with different dropout masks differ for no fault of yours). Then the scores must be identical, because at step t the decoder may only use steps before t, and both runs give it exactly that. If they differ, the decoder is reading something it should not, or losing its state between steps.

Remark. A real example: in an early version of this assignment's pilot, the decoder took its context from its own current state instead of keeping the encoder's context fixed. Training (teacher forcing) still looked fine, but generation drifted into repetition ("london london london") and the BLEU score was

wrong. The whole-versus-step-by-step check exposed it immediately; HW6 runs that check, and a second one that the scores really change when only the context changes.

9 Decoding and Evaluation

Greedy decoding. To translate a new sentence, encode it, start the decoder with `<bos>`, and repeatedly feed back the highest-scoring word (Figure 7, right):

Algorithm: greedy translation of a batch

1. Encode the source sentences; state ← `init_state`(encoder output); $y \leftarrow$ `<bos>` for every sentence.
2. Repeat up to a length cap (e.g. 2 + the longest allowed sentence):
   (a) scores, state ← decoder(y, state) (one step);
   (b) $y \leftarrow$ arg max of the scores; append y to each sentence's output;
   (c) mark a sentence *finished* once it has produced `<eos>`; stop when all are finished.
3. Cut every output at its first `<eos>`; map IDs back to words.

Greedy decoding takes the best word at each step, which is not always the best sentence; beam search keeps several candidates (RNN_ADVANCED). Sampling with a temperature (§4) is for creative text, not for translation.

BLEU: how close is the translation to the references? BLEU compares the words and word sequences (n-grams) of the output with those of one or more human translations:

1. **Clipped n-gram precision** p_n, for $n = 1, \ldots, 4$: the fraction of the output's n-grams that appear in a reference, where each n-gram is credited at most as many times as it appears in a *single* reference (its maximum count over the references, not the sum). Otherwise `the the the the` would score perfectly against "the cat is on the mat".
2. **Brevity penalty** $\mathrm{BP} = \min\left(1, e^{1-r/c}\right)$, with c the output length and r the reference length: short outputs would otherwise get high precision by saying little.
3. $\mathrm{BLEU} = \mathrm{BP} \cdot \exp\left(\frac{1}{4}\sum_{n=1}^{4} \log p_n\right)$, usually reported $\times 100$.

Corpus BLEU (the standard, used in HW6 and HW7) adds up the matches and lengths over the whole test set before dividing, rather than averaging per-sentence scores. Several BLEU variants exist (the d2l book's `d2l.bleu` weights the n-grams differently and gives a different number); we use the standard one throughout, so all BLEU numbers in the course are comparable.

Worked example: BLEU by hand. Output `the cat sat on mat`, reference `the cat sat on the mat`. Unigrams: 5 of 5 match, $p_1 = 1$. Bigrams: *the cat, cat sat, sat on* match, *on mat* does not, $p_2 = 3/4$. Trigrams $p_3 = 2/3$, 4-grams $p_4 = 1/2$. $c = 5$, $r = 6$, $\mathrm{BP} = e^{1-6/5} = 0.819$. $\mathrm{BLEU} = 0.819 \cdot (1 \cdot \frac{3}{4} \cdot \frac{2}{3} \cdot \frac{1}{2})^{1/4} = 0.579$. **Clipping with two references:** output `the the the cat` against "the cat is the best" and "a cat the dog the". *the* appears at most twice in any one reference, so it is credited twice; with *cat*, 3 unigrams match (summing over references would wrongly give 4).

What to expect from HW6's translator. Short everyday sentences often come out exactly right; **longer sentences tend to score worse**. Read your BLEU-by-length table with two things in mind: a bucket of very short sentences has few 3- and 4-grams, so a single wrong word costs a lot there; and the d2l book's `Seq2Seq`, trained the same way, differs from HW6's model mainly in its encoder, which runs on through the padding (its final state has also read the `<pad>` tokens after a short sentence), while HW6's packed encoder stops at each sentence's own end. Compare the two in your own run.

Key point: A suspect for the long-sentence drop: the bottleneck. The whole source sentence has to fit into the encoder's fixed-size final states (one 512-number vector per GRU layer), which start the decoder; the top one is also fed in again at every step as $\mathbf{c}$. The decoder must recover every detail from these. The longer the sentence, the more is squeezed into the same fixed number of values. This is a hypothesis, not a proven cause; HW7 tests it by letting the decoder look back at *every* encoder state (attention).

10 HW6 Map, Common Bugs, and What Comes Next

HW6 part	Sections	The one effect to see
P3 opening walkthrough: tokens, IDs, targets	§2	every position is a training example
P3(A) RNN step, sequence, `nn.RNN`, clipping, unigram baseline	§3, §4, §5	303.5 → about 24: memory pays
P3(B) GRU/LSTM, temperatures, subword side question	§2 key point, §4, §6	gated models far below the RNN
P3(C) gradient vs distance	§5, Figure 4	the RNN's straight falling line
P4(A) encoder	§7	shapes; final state ignores padding
P4(B) decoder, masked loss, consistency check	§7, §8	whole target = one word at a time
P4(C) greedy decoding	§9	short sentences come out right
P4(D) BLEU overall and by length; save	§9	longer sentences tend to score worse
P5 (theory) why the LSTM reaches further back; clipping	§5, §6 key points	product of factors vs the cell path
P8–P11 (optional)	RNN_ADVANCED	

Common bugs.

- Targets not shifted by one (the model learns to copy its input: a suspiciously low loss).
- Time-major vs batch-first mixed up: HW6's RNN tensors are $(T, n, \cdot)$; a transposed batch still runs, and learns nonsense.
- Copying weights into `nn.RNN` without the transpose, or with only one of its two biases.
- Clipping before `backward()` (no gradient yet) or after `step()` (too late).
- Perplexity from a summed instead of an averaged loss, or with padding counted.
- Temperature applied to probabilities instead of logits.

- The masked loss averaging over padding (§8 example: 1.31 instead of 0.75).
- The decoder reading its context from its own state instead of the fixed encoder context.
- Decoding that never stops at `<eos>`, or BLEU computed on IDs with `<eos>`/`<pad>` left in.

Key point: HW7 needs your own HW6 full-run results. HW7 compares attention and Transformers with *your* HW6 language models and translator. Problem 4's last cell writes `results_hw6.json` (including Problem 3's perplexities) and `hw6_seq2seq.pt`. Produce them in **full** mode (dev-mode files are labelled and HW7 refuses them) and keep both.

What comes next.

- **Words → subwords** (TOKENS note, HW7 P1): byte-pair encoding, the tokenizer of every modern language model.
- **The bottleneck → attention** (ATTENTION note, HW7 P3): instead of one context $\mathbf{c}$, the decoder looks at all encoder states at every step.
- **Recurrence → self-attention** (TRANSFORMER note, HW7 P4–P5): drop the recurrence altogether, so all positions are processed in parallel, and the same TinyStories task goes from about 18 to about 15 perplexity with a GPT.

	RNN (this note)	**Transformer (Week 14)**
How a position sees the past	through the chain of states	directly, by attention
Path length from word $t-d$ to t	d steps (the product of factors)	1 step
Training over T positions	T sequential steps	one parallel pass
Memory while generating	a fixed-size state	grows with the text

For more depth (the LSTM cell from scratch, GRU conventions, beam search, bidirectional and deep RNNs, and how a "dev" run is chosen), see **RNN_ADVANCED** (optional).

Further reading. d2l chapters 9 (RNNs) and 10 (modern RNNs, encoder–decoder, seq2seq); C. Olah, "Understanding LSTM Networks" (the standard LSTM picture); A. Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks" (character-level samples).