Recurrent Networks: Advanced Topics

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

---

*Optional. This note assumes the basic **RNN** note (the cell, BPTT, gates, the encoder–decoder, BLEU) and uses its notation: time-major tensors, row vectors, PyTorch's GRU convention. Each section stands on its own.*

1 The LSTM Cell From Scratch, and How `nn.LSTM` Stores It

The four LSTM equations of the RNN note (§6) each have an input matrix, a recurrent matrix and a bias. Instead of twelve separate tensors, write them as one product and split the result:

$$Z = X_t W_x + H_{t-1} W_h + b, \qquad W_x \in \mathbb{R}^{d\times 4h},\ W_h \in \mathbb{R}^{h\times 4h},\ b \in \mathbb{R}^{4h},$$

then cut Z $(n \times 4h)$ into four blocks of h columns, in PyTorch's order **(i, f, g, o)**: $I = \sigma(Z_i)$, $F = \sigma(Z_f)$, $\tilde{C} = \tanh(Z_g)$, $O = \sigma(Z_o)$, and finish with $C_t = F \odot C_{t-1} + I \odot \tilde{C}$, $H_t = O \odot \tanh C_t$. **Meaning:** the four gates are four dense layers reading the same input; stacking them is only an efficiency trick (one big matrix product instead of four small ones).

Mapping to `nn.LSTM`. `weight_ih_l0` is $W_x^\top$ $(4h\times d)$, `weight_hh_l0` is $W_h^\top$ $(4h\times h)$, and b is `bias_ih_l0 + bias_hh_l0` (two biases again, as in `nn.RNN`). A from-scratch step with this mapping matches `nn.LSTM` to rounding (optional HW6 P8).

Worked example: one LSTM step, one unit. Suppose that the four pre-activations come out as $(Z_i, Z_f, Z_g, Z_o) = (0, 2, 1, 0)$ and $C_{t-1} = 0.5$. Then $I = \sigma(0) = 0.5$, $F = \sigma(2) = 0.8808$, $\tilde{C} = \tanh(1) = 0.7616$, $O = 0.5$; $C_t = 0.8808 \cdot 0.5 + 0.5 \cdot 0.7616 = 0.8212$ and
$H_t = 0.5 \tanh(0.8212) = 0.3379$. Most of the old memory survived (the forget gate is open), and half of the candidate was added.

The usual mistakes: the wrong gate order (f and i swapped), tanh on the output gate, or forgetting the tanh in $H_t = O \odot \tanh C_t$.

The forget-gate bias. At initialization $Z_f \approx 0$, so $F \approx \sigma(0) = 0.5$, and the cell halves its memory every step: $0.5^{20} \approx 10^{-6}$. A common trick is to initialize the forget-gate bias to 1 or more, so that $F \approx 0.73$ or higher and the network starts out remembering (Gers et al.; Jozefowicz et al.). PyTorch initializes all biases the same way and does not do this by default.

2 GRU Conventions: Where the Reset Gate Goes

There are three GRU formulas in circulation, and they are *not* the same function.

- **PyTorch** (`nn.GRU`, the RNN note): $\tilde{H}_t = \tanh\left(X_t W_{xh} + b_{xh} + R_t \odot (H_{t-1} W_{hh} + b_{hh})\right)$ — the reset gate multiplies the *result* of the recurrent product.
- **d2l** (10.2): $\tilde{H}_t = \tanh\left(X_t W_{xh} + (R_t \odot H_{t-1}) W_{hh} + b_h\right)$ — the reset gate multiplies the state *before* the product. Same as the original paper's first version.

- **Cho et al. 2014 and many tutorials** write $H_t = (1 - Z_t) \odot H_{t-1} + Z_t \odot \tilde{H}_t$: the roles of Z_t and $1 - Z_t$ are swapped. This is only a relabelling of the gate (it learns $1 - Z$ instead), but it flips the meaning of "$Z \approx 1$".

Worked example: same weights, two formulas. A small GRU (`nn.GRU(2,2)`: 2 inputs, 2 units; seed 0), input $(1, -1)$, state $(0.5, -0.25)$. `nn.GRU` and the PyTorch formula by hand both give $H_t = (0.5205, -0.0175)$; the d2l formula with the same weights gives (0.5200, 0.1548). So a from-scratch GRU written from the book will *not* match `nn.GRU` after a weight copy; both are fine models, they are different ones.

Why PyTorch's placement: $H_{t-1}W_{hh}$ for all three gates can be computed as one product before any gate is known, which is faster on a GPU (the cuDNN formulation).

3 Swapping RNNs Inside the Translator

Your RNN in the translator (optional HW6 P9). It is more than a one-line swap, for two reasons.

- **Packed sequences.** The HW6 encoder runs `nn.GRU` on a *packed* batch (`pack_padded_sequence`): each sentence is processed only up to its own length, so the final state belongs to the real last token. A from-scratch loop cannot read a packed object. The simple alternative: run your RNN on the padded batch, keep all states (T, n, h), and pick each sentence's state at its last real position, `H[len-1, i]`. Without that pick, the "final" state of a short sentence has read several `<pad>` tokens and is a worse summary.
- **Two layers.** The translator stacks two recurrent layers: layer 2 reads layer 1's outputs as its inputs. Stack two of your RNNs the same way; the state is then $(2, n, h)$, one row per layer, and the context is the top layer's final state.

An LSTM translator (optional HW6 P10). The LSTM's state is a pair (H, C). The decoder's `init_state` must pass both on (the context is still the top layer's final H), and every place that unpacks the state must expect a tuple. Expect a translator of similar quality to the GRU one at the same budget; neither gate design changes the fixed-context bottleneck.

4 The Gradient Analysis, Properly

The Jacobian product. For $H_t = \tanh(X_t W_{xh} + H_{t-1} W_{hh} + b)$ (one row, one example), one step back multiplies by $J_t = W_{hh} \operatorname{diag}(1 - H_t^2)$ (row-vector convention), so

$$\left\| \frac{\partial H_T}{\partial H_{T-d}} \right\| \le \prod_k \left\| W_{hh} \right\| \left\| \operatorname{diag}(1 - H_k^2) \right\| \le \left(\sigma_{\max}(W_{hh}) \right)^d,$$

since $0 < 1 - H^2 \le 1$. **Meaning:** if the largest singular value of W_{hh} is below 1, the gradient from d steps back is *guaranteed* to shrink at least geometrically — vanishing. If it is above 1, the bound allows growth, and saturation ($1 - H^2 \approx 0$) may or may not stop it — the "can explode" case that clipping handles (Bengio et al. 1994; Pascanu et al. 2013).

The LSTM path. Along the cell, $\partial C_t / \partial C_{t-1} = \operatorname{diag}(F_t)$ *plus* the terms through the gates' dependence on H_{t-1}. Ignoring those, the product over d steps is $\prod_k F_k$, controlled directly by learned gates and free

of W_{hh}. This is the "uninterrupted gradient flow" picture. It is not a guarantee: if the network learns $F \approx 0$ the path closes, and the gate terms add the usual products back in. Gates make long-range learning *possible*, not automatic.

Truncation and state carrying. HW6 trains on random 64-token windows, each from a zero state. The alternative is **sequential partitioning**: consecutive windows, the final state of one becoming the initial state of the next, with `H.detach()` in between so that backpropagation stops at the window border. The state then carries information across windows, but the gradient does not. Random windows are simpler and lose little for short stories; carried states matter for long documents.

5 Exposure Bias: Training on the Truth, Testing on Your Own Guesses

Teacher forcing (RNN note, §8) feeds the decoder the *true* previous word in training, but at test time it feeds back its *own* guesses. The model never practised recovering from its own mistakes: one wrong word puts it in a state it has not seen, and errors can compound along the sentence. This mismatch is called **exposure bias**. Remedies:

- **Scheduled sampling** (Bengio et al. 2015): during training, replace the true previous word by the model's own prediction with a probability that grows over training. It breaks the one-call parallel training (each step needs the previous step's output), which is why it is rarely used with Transformers.
- **Beam search** (§9) reduces the damage at test time by not committing to a single early choice.
- In practice, with enough data, teacher forcing works well, and every model in HW6 and HW7 uses it.

A symptom you can see in HW6: greedy translations of long sentences often start right and drift (repeated words, a wrong ending), while short ones are exact.

6 Comparing Perplexities Across Tokenizers

Perplexity is "per token", so it is only comparable between models that cut the text into the same tokens. A character-level model with perplexity 3 per character is not better than a word model with perplexity 20 per word: a word is about five characters, and $3^5 = 243 > 20$. The fair comparison is the total log-probability of the same text, converted to a common unit:

$$\text{bits per character} = \frac{\sum_t -\log_2 p(x_t \mid x_{<t})}{\text{number of characters}}, \qquad \text{PPL}_{\text{word}} = \exp\Big(\frac{\sum_t -\ln p(x_t \mid x_{<t})}{\text{number of words}}\Big),$$

where the sum runs over the model's own tokens. **Meaning:** measure surprise per unit of *text*, not per unit of the model's own choosing. HW6 and HW7 sidestep the issue by using the same word vocabulary for the RNNs, the LSTM and the GPT; the same rule is why a dev-mode perplexity (smaller vocabulary) is not comparable with a full one (§12).

7 Weight Tying: One Table for Reading and Writing Words

A language model has two big word tables: the embedding $E \in \mathbb{R}^{|\mathcal{V}| \times d}$ (reading) and the output layer $W_{hq} \in \mathbb{R}^{d \times |\mathcal{V}|}$ (writing). In HW6's language models ($|\mathcal{V}| = 5{,}274$, embedding 256, hidden 512) the

embedding has 1.35M parameters and the output layer 2.7M: together the bulk of the 4.4M. **Weight tying** uses one table for both, $W_{hq} = E^\top$ (Press & Wolf 2017; Inan et al. 2017): a word's input vector is also the direction its output score is read from. It halves those parameters and usually helps, because rare words get gradient through both roles. It needs the last hidden size to equal the embedding size. GPT-2 ties its weights, and so does HW7's GPT; HW6's RNNs do not (their hidden size 512 differs from the embedding size 256).

8 Deep and Bidirectional RNNs

Deep (stacked) RNNs. Layer ℓ reads layer $\ell - 1$'s state sequence as its input sequence. Two or three layers are common (HW6's translator uses two); dropout between the layers, not along the recurrence, is the usual regularizer.

Bidirectional RNNs. Run one RNN left to right and another right to left, and concatenate their states at each position: every position then sees the whole sequence. This is fine for an *encoder*, which reads a complete sentence before anything is produced; HW7's reverse task uses a bidirectional encoder for exactly this reason (each output must know its distance from the end). It is *not* allowed for a language model or a decoder: the backward RNN would read the very word to be predicted.

9 Beam Search vs Greedy vs Sampling

Greedy decoding takes the best next word; the best *sentence* may start with a worse first word.

Worked example: a beam of width 2. Step 1: $p(A) = 0.5$, $p(B) = 0.4$, $p(C) = 0.1$. Step 2 after A: C 0.6, `<eos>` 0.2, A 0.1, B 0.1; after B: `<eos>` 0.8, C 0.1, A 0.05, B 0.05. Greedy picks A, then C: probability $0.5 \cdot 0.6 = 0.30$. A beam of width 2 keeps both A and B after step 1 and scores all extensions: $(B, \texttt{<eos>}) = 0.32$ beats $(A, C) = 0.30$.

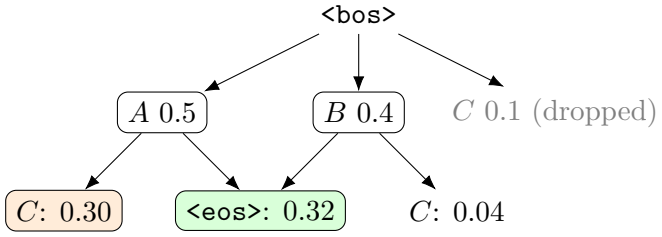


Figure 1: Beam search with width 2. Greedy follows the orange path; the beam finds the green one.

The algorithm. Keep the k best partial sentences; extend each by every word; keep the k best of the $k|\mathcal{V}|$ extensions by total log-probability; move finished sentences aside. Because every word multiplies in a probability below 1, longer sentences always score lower, so the final choice uses a **length-normalized** score, e.g. the log-probability divided by the length (or by length$^\alpha$, $\alpha \approx 0.6$–1). Beam search helps translation, where there is one right answer to search for. Story-writing and chat models *sample* instead (temperature, top-k; the TRANSFORMER note), because the highest-probability text is bland and repetitive.

10 Classical Relatives of "Plan B"

The RNN note's Plan B, a hidden state that summarizes the past, is old. An **HMM** (EM_ADVANCED) has a discrete random state with transition probabilities; the forward algorithm computes the belief $p(z_t \mid x_{1..t})$ recursively, exactly the shape $H_t = f(H_{t-1}, x_t)$. A **Kalman filter** does the same with a Gaussian state and linear dynamics. An RNN replaces the probabilistic state by a deterministic vector and the hand-designed update by a learned network: less interpretable, but able to learn whatever summary the task needs.

11 Characters, Words or Subwords?

Karpathy's character-level RNNs write convincing-looking Shakespeare and C code character by character, with no vocabulary problem at all, but each word costs about five steps and the model must spend capacity on spelling. HW6 uses words because TinyStories' vocabulary is tiny (5,274 words cover 99.6% of it). Real text needs subwords: frequent words whole, rare words in pieces — the tokenizers of all modern language models, and the first topic of Week 13 (TOKENS note, HW7 P1).

12 Debug Small, Run Full

Every HW6 and HW7 notebook has a switch, `MODE`, with two settings that are given, not chosen by you. The design behind them is worth understanding (optional HW6 P11).

HW6 part	**full**	**dev**
P1 Fashion-MNIST	60k / 10k images, 10 epochs	6k / 2k, 2 epochs
P2 CIFAR-10 ResNet	50k / 10k, 10 epochs, widths 32/64/128	2k / 1k, 3 epochs, widths 16/32/64
P2(C) transfer	500 per class, 128 px	50 per class, 64 px
P3 language model	batch 64 × 64 tokens, min. count 10, embedding/hidden 256/512, 3,000 steps	32 × 32, min. count 50, 64/128, 300 steps
P4 translator	100k pairs, 256/512, 2 layers, 2,500 steps	10k pairs, 64/128, 2 layers, 800 steps

What a dev run must keep. Every line of code must run on real inputs with real shapes: the same data modules and tokenization, the same architectures (same number of layers, only narrower), padding, several batches, every length bucket (dev keeps HW6's full 2,000-sentence test set), saving and loading. Bugs then show up as failed checkpoints or errors, in seconds.

What it can give up. Quality: width, steps, data. That is why dev numbers are labelled `[DEV --- not for answers]`. Optional P11 asks you to work out the consequences; three questions to think through, using the table above:

- A checkpoint compares your layer with PyTorch's on fixed inputs; an accuracy measures what a model learned. Which of the two can depend on how much data and how many steps the run had?
- Dev's language model keeps only words seen at least 50 times. What happens to all the rarer words, and how does that change what one perplexity number measures? When are two perplexities comparable?
- "Worse on long sentences" is a statement about a model that is already decent on short ones. What would the length buckets of a translator that is poor everywhere look like?

The overfit-one-batch check. A correct model with enough capacity can memorize a handful of examples: the loss on one fixed batch must fall close to zero in a few hundred steps (HW6's translator: 9.2 to 0.1 on 16 sentence pairs in a few seconds). If it cannot, something is wrong with the wiring, the loss or the masking — whatever the data size. It is the cheapest end-to-end test of a training setup.

When is dev too small? Turn the first paragraph around: list what a run must exercise for a bug to show up (padding, every length bucket, several batches, saving and loading, a loss that moves), and ask which shrinking step would quietly remove one of them.

13 RNNs Today

Transformers (Week 14) replaced RNNs for most language tasks: they train in parallel over all positions and connect any two positions in one step. RNNs keep one real advantage: generating or reading a stream costs a *fixed* amount of memory and compute per token, independent of how long the stream has been (a Transformer's cost per token grows with the context). That property is behind the recent **state-space models** (e.g. Mamba), which are recurrent in the RNN sense but designed so that training can still be parallelized — a sign that the recurrence of this note is not obsolete, only re-engineered.

14 Further Reading

- S. Hochreiter, J. Schmidhuber. Long short-term memory. *Neural Computation* 9(8), 1997.
- F. A. Gers, J. Schmidhuber, F. Cummins. Learning to forget: continual prediction with LSTM. *Neural Computation* 12(10), 2000.
- Y. Bengio, P. Simard, P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks* 5(2), 1994.
- R. Pascanu, T. Mikolov, Y. Bengio. On the difficulty of training recurrent neural networks. *ICML* 2013 (arXiv:1211.5063).
- K. Cho et al. Learning phrase representations using RNN encoder–decoder for statistical machine translation. *EMNLP* 2014 (arXiv:1406.1078).
- I. Sutskever, O. Vinyals, Q. V. Le. Sequence to sequence learning with neural networks. *NeurIPS* 2014 (arXiv:1409.3215).
- R. Jozefowicz, W. Zaremba, I. Sutskever. An empirical exploration of recurrent network architectures. *ICML* 2015.
- K. Papineni, S. Roukos, T. Ward, W.-J. Zhu. BLEU: a method for automatic evaluation of machine translation. *ACL* 2002.
- S. Bengio, O. Vinyals, N. Jaitly, N. Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. *NeurIPS* 2015 (arXiv:1506.03099).
- O. Press, L. Wolf. Using the output embedding to improve language models. *EACL* 2017, arXiv:1608.05859.
- H. Inan, K. Khosravi, R. Socher. Tying word vectors and word classifiers: a loss framework for language modeling. *ICLR* 2017 (arXiv:1611.01462).

- A. Graves. Generating sequences with recurrent neural networks. arXiv:1308.0850, 2013.
- A. Gu, T. Dao. Mamba: linear-time sequence modeling with selective state spaces. arXiv:2312.00752, 2023.
- d2l chapters 9–10; C. Olah, "Understanding LSTM Networks"; A. Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks"; Stanford CS231n Lecture 10 (RNNs).