Gated Recurrent Models

Handout Notes - Generative and Deep Learning (GDL)

Davide Bacciu - University of Pisa

Notation. A sequence is written as $\mathbf{x} = (x_1, \ldots, x_T)$, where x_t is the input at time or position t. The hidden state is $h_t \in \mathbb{R}^m$. For gated recurrent models we use c_t for the internal cell state of an LSTM, and h_t for its exposed hidden/output state. Gate vectors are written i_t, f_t, o_t, r_t, z_t, with entries in $(0, 1)$ obtained through the logistic sigmoid $\sigma(\cdot)$. Elementwise multiplication is denoted by $\odot$. Recurrent and input weight matrices are written generically as W_h, W_{in}, with gate-specific matrices carrying subscripts. A dataset is denoted by $\mathcal{D}$ with size $N = |\mathcal{D}|$. When discussing randomized recurrent models, the reservoir state is still denoted by h_t, while only the readout weights are trained.

1 Why gated recurrent models are needed

Vanilla recurrent neural networks are elegant but fragile. They process sequential data by iteratively updating a hidden state,

$$h_t = \phi(W_h h_{t-1} + W_{\text{in}} x_t + b),$$

and in principle this state can summarize information from arbitrarily long input histories. In practice, however, the recurrent dynamics often make long-range learning difficult. Backward gradients tend to vanish or explode, and forward information about distant inputs may either dissipate or destabilize the memory.

A naive solution, discussed in the study of information propagation, is to use identity-like recurrence:

$$h_t = h_{t-1} + \widehat{c}(x_t).$$

This has favorable spectral properties because it can support near-constant error propagation. But it creates a new problem: memory is accumulated without control. Irrelevant inputs, redundant repetitions, and transient noise can saturate the state, because the model has no mechanism to decide what should be stored, overwritten, exposed, or forgotten.

Gated recurrent models solve this by introducing *learned multiplicative controls*. These controls are neurons whose outputs lie in $[0, 1]$ and are used to modulate other signals. The idea is simple but profound:

> *Do not only compute a candidate memory update; also compute when to write it, when to erase previous content, and when to expose internal memory to the outside.*

This is the key innovation behind Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs).

2 Gating as adaptive control

The general notion of a gate predates recurrent memory models. In a broad sense, gating appears whenever a network uses one signal to modulate another. A gate usually takes the form

$$g_t = \sigma(a_t),$$

where σ is the logistic sigmoid, so that each component of g_t lies between 0 and 1. This bounded output is then used multiplicatively:

$$u_t = g_t \odot v_t.$$

If a component of g_t is close to 0, the corresponding component of v_t is suppressed. If it is close to 1, it is preserved. Intermediate values allow soft, differentiable control.

The importance of this construction is that it turns memory management into a trainable, gradient-based process. Instead of hard reset or hard overwrite rules, the network learns soft decisions about retention and update.

Worked example — Why sigmoid gates implement soft decisions

Let $g = \sigma(a)$ with

$$\sigma(a) = \frac{1}{1 + e^{-a}}.$$

Then:

- if $a \gg 0$, then $\sigma(a) \approx 1$ and the signal passes almost unchanged,
- if $a \ll 0$, then $\sigma(a) \approx 0$ and the signal is almost blocked,
- if $a \approx 0$, then $\sigma(a) \approx 0.5$ and the signal is partially transmitted.

Thus a gate is a differentiable analogue of an on/off switch. Because the controlling variable a is itself computed by the network, the model can learn context-dependent memory policies.

3 From constant-error propagation to controlled forgetting

To motivate gating, it is useful to start from the constant-error idealized memory:

$$h_t = h_{t-1} + \widehat{c}(x_t).$$

This recurrence preserves gradients well because the derivative with respect to the previous state is close to the identity. But it has no forgetting mechanism.

A first refinement is to introduce a *forget gate*:

$$f_t = \sigma(W_{fh} h_{t-1} + W_{fx} x_t + b_f),$$

and define

$$h_t = f_t \odot h_{t-1} + \widehat{c}(x_t).$$

Now the previous state is no longer copied blindly. It is rescaled componentwise before being carried forward. This helps prevent uncontrolled accumulation and allows the model to learn how much past information should survive.

However, this is still incomplete. To obtain a robust recurrent memory system, we also want to control what new information enters the state and what part of the internal memory is exposed externally. These requirements lead directly to the LSTM architecture.

4 Long Short-Term Memory networks

The LSTM is one of the most influential recurrent architectures. Its central idea is to separate:

- an **internal memory state** c_t, designed for relatively stable storage,
- an **exposed hidden state** h_t, used for communication with the rest of the network.

This separation is crucial. The cell state c_t acts as the long-term memory carrier, while h_t is a filtered view of that memory. Three gates regulate the process:

- the **input gate** decides how much candidate information enters the cell,
- the **forget gate** decides how much of the old cell state is retained,
- the **output gate** decides how much of the cell state is exposed as hidden output.

4.1 LSTM equations

A standard LSTM cell is defined by

$$
\begin{aligned}
i_t &= \sigma(W_{ih}h_{t-1} + W_{ix}x_t + b_i),\\
f_t &= \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f),\\
g_t &= \tanh(W_{gh}h_{t-1} + W_{gx}x_t + b_g),\\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t,\\
o_t &= \sigma(W_{oh}h_{t-1} + W_{ox}x_t + b_o),\\
h_t &= o_t \odot \tanh(c_t).
\end{aligned}
$$

The term g_t is often called the candidate state or input proposal. It is the content that could be written into memory. The actual write happens only after scaling by the input gate i_t.

4.2 Interpretation of the LSTM update

The internal update

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

has a very clear interpretation:

- $f_t \odot c_{t-1}$ is the retained part of the old memory,
- $i_t \odot g_t$ is the newly written content.

Thus the LSTM cell behaves like a learnable memory accumulator with controlled forgetting and controlled writing.

The output equation

$$h_t = o_t \odot \tanh(c_t)$$

then decides how much of the internal memory should be exposed externally. This means that memory storage and memory exposure are not the same thing.

Worked example — Why the LSTM cell helps gradient propagation

Consider the cell update

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t.$$

Differentiate with respect to c_{t-1}:

$$\frac{\partial c_t}{\partial c_{t-1}} = \operatorname{diag}(f_t),$$

ignoring indirect dependencies through the gates for the moment. If the forget gate is close to one, then

$$\frac{\partial c_t}{\partial c_{t-1}} \approx I.$$

Thus the cell state can propagate information and gradients almost unchanged through time. This is the core "constant error carousel" intuition: the memory path can remain close to identity when needed, while still allowing learned forgetting through f_t.

This is the essential advantage of LSTMs over vanilla RNNs: the architecture includes a dedicated near-linear memory path whose behavior can be modulated rather than imposed.

5 Step-by-step view of LSTM design

One helpful way to understand LSTMs is as a sequence of architectural refinements.

5.1 Step 1: introduce a linear memory state

Instead of storing everything directly in the nonlinear hidden state, introduce a separate internal state c_t that is updated additively. This creates a memory path less vulnerable to repeated nonlinear contraction.

5.2 Step 2: control what enters memory

Add an input gate i_t so that not every candidate update is written. This prevents the model from indiscriminately absorbing every incoming signal.

5.3 Step 3: control what remains in memory

Add a forget gate f_t so that the model can discard obsolete or harmful past information. This addresses the saturation problem of naive constant-memory propagation.

5.4 Step 4: control what leaves memory

Add an output gate o_t so that the exposed state h_t is a filtered version of internal memory. This allows the network to store information without necessarily revealing it at every step.

Seen this way, the LSTM is not an arbitrary collection of equations. It is a structured answer to the question:

> *How can a recurrent network preserve gradients and memory traces, yet still manage memory selectively?*

6 Deep LSTM architectures

Just as feedforward and convolutional networks become more expressive when stacked, LSTMs can also be arranged in depth. A deep LSTM uses several recurrent layers:

$$h_t^{(1)} \to h_t^{(2)} \to \cdots \to h_t^{(L)},$$

where each layer processes the sequence of hidden representations produced by the previous one.

This creates two kinds of depth:

- **temporal depth**, due to recurrence across time,
- **spatial depth**, due to stacking multiple recurrent layers.

Different layers can then capture different levels of temporal abstraction. Lower layers may encode local or short-range patterns, while higher layers integrate broader contextual structure.

7 Regularizing gated recurrent networks

LSTMs are powerful but also prone to overfitting, especially in sequence modeling tasks with high-capacity hidden representations. Regularization is therefore essential.

7.1 Dropout in recurrent models

Dropout randomly removes units during training to discourage co-adaptation. In recurrent models, however, dropout must be applied with care. If the dropout mask changes at every time step, the recurrent dynamics become excessively noisy. A common strategy is therefore to use the *same dropout mask for the whole sequence* on the recurrent pathway or on specific connections.

In recurrent settings one also encounters *DropConnect*, where individual weights rather than unit activations are randomly suppressed.

The intuition is the same as in feedforward networks:

- reduce reliance on any single pathway,
- create an implicit committee effect,
- improve generalization.

7.2 Activity regularization

Another idea is to penalize hidden activations directly. This is particularly relevant in gated recurrent networks because saturating nonlinearities such as tanh can still cause small gradients when activations become too large in magnitude.

A simple activity regularization term is

$$\alpha \|M \odot h_t\|_2^2,$$

where M may mask dropped units so that only active units are penalized. This encourages moderate activation magnitudes.

A temporal smoothness regularizer can also be used:

$$\beta \|h_t - h_{t+1}\|_2^2.$$

This encourages temporal consistency in the hidden dynamics.

Worked example — Why activity regularization helps with saturation

Suppose a hidden activation passes through tanh. If u is the pre-activation, then

$$\frac{d}{du} \tanh(u) = 1 - \tanh^2(u).$$

When $|u|$ is large, $\tanh(u)$ saturates near ± 1, and the derivative becomes close to zero. Thus strongly saturated activations reduce gradient flow. Penalizing large hidden activations encourages the network to remain more often in a regime where the derivative is not negligible. This does not eliminate all propagation difficulties, but it makes them less severe.

8 Practical training issues: mini-batches and truncated BPTT

Training recurrent networks on long sequences is computationally expensive because backpropagation must traverse many time steps. In practice, two standard strategies are used.

8.1 Mini-batch training

Instead of computing gradients over the full dataset, one processes batches of sequences and averages their gradients. This is the recurrent analogue of mini-batch stochastic gradient descent in feedforward models.

8.2 Truncated backpropagation through time

For very long sequences, one often truncates gradient propagation to a finite window. That is, the hidden state is carried forward continuously, but gradients are only propagated back over the last K steps. This reduces computation and memory use.

The trade-off is clear:

- larger truncation windows capture longer dependencies but cost more,

- shorter windows are cheaper but may miss important long-range credit assignment.

This is one reason why architectural bias matters so much in sequence modeling: the better the architecture preserves useful memory, the less it relies on extremely long explicit gradient paths.

9 Gated Recurrent Units

The Gated Recurrent Unit (GRU) is a simpler gated architecture introduced after the LSTM. It removes the separate cell state and combines some of the LSTM mechanisms into a more compact design.

A standard GRU uses two gates:

- an **update gate** z_t,
- a **reset gate** r_t.

Its equations are

$$z_t = \sigma(W_{zh}h_{t-1} + W_{zx}x_t + b_z),$$
$$r_t = \sigma(W_{rh}h_{t-1} + W_{rx}x_t + b_r),$$
$$\tilde{h}_t = \tanh\big(W_{hh}(r_t \odot h_{t-1}) + W_{hx}x_t + b_h\big),$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t.$$

Here $\tilde{h}_t$ is the candidate hidden update. The reset gate controls how much of the previous hidden state is used to compute that candidate. The update gate interpolates between keeping the old state and replacing it with the candidate.

9.1 Interpretation of the GRU

The GRU can be read as a simplified LSTM:

- the update gate plays a role analogous to a coupled input–forget mechanism,
- the reset gate controls how strongly the past contributes when computing the candidate,
- there is no separate output gate and no separate cell state.

This makes the GRU computationally lighter and often easier to train, while still preserving the core idea of gated memory control.

Worked example — Why the GRU can preserve old state

Consider the update equation

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t.$$

If $z_t \approx 0$, then

$$h_t \approx h_{t-1},$$

so the state is preserved almost unchanged. If $z_t \approx 1$, then

$$h_t \approx \tilde{h}_t,$$

so the hidden state is mostly overwritten by the new candidate. Thus the update gate directly implements a soft interpolation between memory retention and memory rewrite.

This interpolation viewpoint makes the GRU especially intuitive: the model learns how much to trust the past versus the present at each step.

10 LSTM versus GRU

LSTMs and GRUs solve the same broad problem, but they do so with different levels of structural complexity.

10.1 LSTM advantages

LSTMs provide more explicit memory management:

- separate internal cell memory c_t,
- distinct input, forget, and output gates,
- more precise control over storage versus exposure.

This can be advantageous when tasks require subtle memory operations.

10.2 GRU advantages

GRUs are more compact:

- fewer gates,
- no separate cell state,
- fewer parameters and often lower computational cost.

In many practical tasks they perform comparably to LSTMs and can be easier to optimize.

There is no universal winner. Architecture choice depends on the task, dataset size, computational budget, and desired inductive bias.

11 Randomized recurrent architectures

Not all strategies for handling long-term dependencies rely on learning a sophisticated recurrent core. A different idea is to control memory properties directly through the architecture and train only a simple readout. This is the logic of *reservoir computing*.

The guiding principle is:

> *Use a fixed, randomly initialized recurrent dynamical system to generate rich nonlinear state trajectories, and train only the output layer.*

This shifts the burden from end-to-end recurrent optimization to architectural design plus linear readout learning.

12 Reservoir computing

In reservoir computing, the state update typically has the form

$$h_t = \tanh(W_{\text{in}} x_t + W_h h_{t-1}),$$

but now W_{in} and W_h are *not trained*. They are randomly initialized once and then kept fixed. Only the readout is trained, often linearly:

$$y_t = W_{\text{out}} h_t.$$

This produces a striking contrast with standard RNNs:

- **standard RNNs:** learn the recurrent dynamics,
- **reservoir computing:** fix the recurrent dynamics and learn only how to decode them.

The hope is that a high-dimensional nonlinear reservoir creates a rich basis expansion of the input history, making the target task approximately linearly solvable in the reservoir state space.

13 Echo State Property

The core theoretical condition behind echo state networks is the *Echo State Property* (ESP). Intuitively, ESP means that the effect of initial conditions disappears over time, so that the current state depends only on the input history, not on arbitrary initial transients.

Formally, ESP requires

$$\lim_{k\to\infty} \frac{\partial h_t}{\partial h_{t-k}} = 0.$$

Using Jacobian products,

$$\frac{\partial h_t}{\partial h_{t-k}} = \prod_{i=t-k+1}^{t} J_i.$$

Thus ESP requires this product to contract on average.

This is deeply related to the same propagation analysis used earlier:

- if the recurrent dynamics contract too strongly, memory decays rapidly,
- if they are close to the stability boundary, memory decays more slowly,
- if they are expansive, the dynamics may become unstable or chaotic.

Worked example — Spectral-radius intuition for reservoir memory

Suppose the reservoir operates in a near-linear regime, so that the recurrent Jacobian is approximately W_h. Then

$$\frac{\partial h_t}{\partial h_{t-k}} \approx W_h^k.$$

If $\rho(W_h) \ll 1$, powers of W_h decay quickly, so the reservoir forgets very fast. If $\rho(W_h) \approx 1$, decay is slow and memory lasts longer. If $\rho(W_h) > 1$, powers of W_h tend to amplify perturbations, leading to instability. Thus the spectral radius of the recurrent matrix acts as a practical control knob for the memory depth of the reservoir.

14 Echo State Networks in practice

An Echo State Network (ESN) is the standard model of reservoir computing. Its practical workflow is:

1. initialize a sparse random recurrent matrix W_h,
2. initialize a random input matrix W_{in},
3. rescale W_h to a desired spectral radius ρ_{des},
4. drive the reservoir with the full training sequences to collect states,
5. discard an initial washout transient,
6. fit the readout weights by linear regression or ridge regression.

If the collected state matrix is

$$H = [h_1, \ldots, h_T],$$

and the corresponding target matrix is

$$Y = [y_1, \ldots, y_T],$$

then a ridge-regression readout can be obtained as

$$W_{\text{out}} = YH^\top (HH^\top + \lambda I)^{-1}.$$

This makes ESNs very training-efficient: all difficult recurrent optimization is avoided.

15 Why randomized reservoirs can work

At first sight, it may seem surprising that a randomly initialized recurrent system can be useful. The underlying intuition combines two ideas.

First, the reservoir projects the input into a high-dimensional nonlinear dynamical feature space. This is analogous in spirit to random-feature expansions and Cover's theorem: complex structure may become more linearly separable after a sufficiently rich nonlinear embedding.

Second, when the reservoir is contractive but not too contractive, the state tends to organize recent input history in a suffix-sensitive way. Different recent contexts map to distinguishable regions of state space, even without training the recurrent core.

Thus randomized reservoirs can provide a strong memory basis, while the readout learns the actual task.

16 Deep echo state networks

Reservoir ideas can also be stacked in depth. A deep echo state network uses multiple reservoir layers:

$$h_t^{(1)} \to h_t^{(2)} \to \cdots \to h_t^{(L)},$$

with each layer providing a different timescale or level of abstraction. As in deep LSTMs, depth introduces hierarchical temporal representations. The difference is that the recurrent dynamics remain fixed and only the readout is trained.

17 Autoregressive sequence generation with recurrent models

Gated recurrent models can also be used generatively. In a basic autoregressive setup, the model predicts the next element in a sequence and then feeds its own prediction back as future input.

During training, one often uses *teacher forcing*: the true previous target is provided as the next-step input. During generation, this is no longer possible, so the model must use its own previously generated output. This discrepancy explains why sequence generation can be harder at test time than one-step-ahead prediction during training.

An autoregressive recurrent model is therefore a *stateful differentiable machine*: it maintains a state, emits outputs, and reuses those outputs to continue generation.

18 Conceptual summary

This lecture presents two broad families of solutions to the difficulty of learning long-term dependencies in recurrent models.

Gated approaches

- Gates are neurons whose outputs multiplicatively scale other signals.
- LSTMs introduce a controlled memory cell with input, forget, and output gates.
- GRUs offer a simpler gated alternative with reset and update gates.
- These models improve information propagation by combining near-linear memory paths with adaptive control of writing, retention, and exposure.

Randomized approaches

- Reservoir computing fixes recurrent dynamics and trains only a readout.

- The Echo State Property ensures that initial conditions fade and the state depends on input history.
- Spectral properties of the reservoir determine the memory timescale.
- ESNs trade expressive recurrent optimization for efficient linear readout training.

Taken together, these approaches illustrate a general theme of deep learning:

> *Good recurrent architectures are not only about expressive nonlinear transformations, but also about shaping information propagation and memory dynamics.*