CS728 PA2: Training Dynamics of RNNs and GRUs

Due: 10th March, 2026 (subject to change)

Learning goals

This assignment is inspired by Pascanu et al's paper on the learning dynamics of RNNs. Here, you will:

1. Implement a vanilla RNN and a GRU *from the equations* (no `torch.nn.RNN/GRU`).
2. Train them on synthetic long-range dependency tasks.
3. Log and interpret diagnostics of the gradients, and relate them to activation/gate saturation, and possible gradient vanishing and exploding.
4. Compare the behavior of RNN vs GRU under different regimes (with/without clipping; tanh vs sigmoid).

1 Starter code and how to run

You are given a PyTorch training script:

- `trainingRNNs_torch/train.py` (entry point: `python -m trainingRNNs_torch.train`)
- `trainingRNNs_torch/model.py` (you will implement RNN and GRU here)
- `trainingRNNs_torch/tasks.py` (task generators; do not modify unless asked)

Important: "iterations" mean SGD updates (batches), not epochs. Each iteration samples a fresh mini-batch (and usually a fresh sequence length). There is no dataset epoch.

2 What you must implement

2.1 Part 1: Vanilla RNN cell

Implement the recurrence

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)\,, \qquad o_t = W_{ho}h_t + b_o,$$

where ϕ is either tanh or σ depending on `--init`. Your code must:

- Implement `recurrent_weight_for_rho()` to expose the hidden-to-hidden weight (used for logging `rho_Whh`);
- Implement the forward pass and return both logits (or regression output) and the full hidden sequence $h_{1:T}$ with shape (T, B, H) for diagnostics. The shape of the logits will depend on the type of task.

2.2 Part 2: GRU cell

Implement a GRU from equations (use the same symbol naming convention as the starter code):

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)\,, \quad r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)\,,$$

$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)\,, \quad h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t.$$

Your GRU must:

- Implement the forward pass and return logits and hidden sequence $h_{1:T}$. The shape of the logits will depend on the type of task;
- When `--diagGates` is enabled, also return gate traces $\{z_t\}$ and $\{r_t\}$ for diagnostics;
- Implement `recurrent_weight_for_rho()` to return the *candidate* recurrent matrix (the one multiplying h_{t-1} inside $\tilde{h}_t$; typically W_{hh} in the equation above).

2.3 Part 3 (logging): diagnostics you must produce

The starter script already contains diagnostic hooks. You must ensure your model exposes what the script needs. At each check point (`--checkFreq`), the script stores and/or prints:

1. **Global gradient norm** $\|\nabla_\theta \mathcal{L}\|_2$ (printed as `avg grad norm`).
2. **Post-clipping gradient norm** (printed as `avg grad norm (post clip)` when clipping is on).
3. **Gradient-through-time signal**:

   $$g_t = \text{mean over the batch of } \left\| \tfrac{\partial \mathcal{L}}{\partial h_t} \right\|_2, \quad t = 1, \ldots, T,$$

   stored in `grad_time` and summarized as a histogram of $\log_{10}(g_t)$ (use a tiny ε to avoid $\log 0$). If most mass sits at very negative values (e.g., -8 to -12), the gradient has effectively vanished at many timesteps.
4. **Hidden distance-to-saturation**: For tanh hidden states ($h_t \in [-1, 1]$ elementwise), a simple "distance from saturation" for each unit is

   $$d(h) = 1 - |h| \in [0, 1],$$

   and we average over batch and hidden units to get a per-timestep summary

   $$s_t = \mathbb{E}\big[\, 1 - |h_t| \,\big].$$

   This is stored in `sat_time` and histogrammed. Small s_t means many units are close to ± 1 (saturated). For sigmoid-valued quantities $v \in [0, 1]$ (e.g., GRU gates), we use $d(v) = \min(v, 1 - v)$.
5. **Gate saturation distances (GRU only, with `--diagGates`).** A GRU has two sigmoid gates: the update gate z_t and the reset gate r_t. Each gate value lies in $[0, 1]$. A gate is *saturated* when it is very close to 0 or 1.

   We measure "distance to saturation" for a gate value $v \in [0, 1]$ by

   $$d(v) \;=\; \min(v,\; 1 - v).$$

   So:

- $d(v) \approx 0$ means the gate is near 0 or 1 (saturated),
- $d(v) \approx 0.5$ means the gate is near 0.5 (not saturated).

With `--diagGates`, the code logs the per-time-step average of $d(\cdot)$ over all batch examples and hidden dimensions:

$$\texttt{z\_sat\_t} = \mathbb{E}[\, d(z_t)\,], \qquad \texttt{r\_sat\_t} = \mathbb{E}[\, d(r_t)\,].$$

It also prints histogram summaries (mean / percentiles) of these values.

6. **Spectral radius proxy** `rho_Whh`: $\rho(W)$ computed from eigenvalues of the returned recurrent weight matrix W.

You will plot: (i) histogram of $\log_{10}(g_t)$, (ii) histogram of saturation distance(s), and (iii) validation curves.

3 Regimes to run (commands)

Use the commands below *exactly* (copy/paste). You may change only the `--name` prefix to keep files organized.

3.1 Common flags

All runs below use:

```
--nhid 50 --lr 0.01 --bs 20 --min_length 50 --max_length 200
--maxiters 50000 --ebs 10000 --cbs 1000 --checkFreq 20
--seed 52 --valid_seed 12345 --collectDiags --diagBins 60 --satThresh 0.05
```

3.2 Task 1: Memorization (classification)

In most other tasks the model usually makes predictions on the hidden state of the final time step. In this case, the model must predict/classify *every step* of the sequence. This corresponds to `classifType = softmax` in the code.

RNN (tanh), no clipping

```
python -m trainingRNNs_torch.train --task mem --model rnn --alpha 0.0 \
  --clipstyle nothing \
  --nhid 50 --lr 0.01 --bs 20 --min_length 50 --max_length 200 \
  --maxiters 50000 --ebs 10000 --cbs 1000 --checkFreq 20 \
  --seed 52 --valid_seed 12345 --collectDiags --diagBins 60 --satThresh 0.05 \
  --name A1_mem_rnn_tanh_noclip
```

RNN (tanh), with clipping (moderate)

```
python -m trainingRNNs_torch.train --task mem --model rnn --alpha 0.0 \
  --clipstyle rescale --cutoff 0.05 \
  --nhid 50 --lr 0.01 --bs 20 --min_length 50 --max_length 200 \
  --maxiters 50000 --ebs 10000 --cbs 1000 --checkFreq 20 \
  --seed 52 --valid_seed 12345 --collectDiags --diagBins 60 --satThresh 0.05 \
  --name A2_mem_rnn_tanh_clip005
```

RNN (tanh), with clipping (aggressive)

```
python -m trainingRNNs_torch.train --task mem --model rnn --alpha 0.0 \
  --clipstyle rescale --cutoff 0.01 \
  --nhid 50 --lr 0.01 --bs 20 --min_length 50 --max_length 200 \
  --maxiters 50000 --ebs 10000 --cbs 1000 --checkFreq 20 \
  --seed 52 --valid_seed 12345 --collectDiags --diagBins 60 --satThresh 0.05 \
  --name A3_mem_rnn_tanh_clip001
```

GRU, no clipping (with gate diagnostics)

```
python -m trainingRNNs_torch.train --task mem --model gru --alpha 0.0 \
  --clipstyle nothing --diagGates \
  --nhid 50 --lr 0.01 --bs 20 --min_length 50 --max_length 200 \
  --maxiters 50000 --ebs 10000 --cbs 1000 --checkFreq 20 \
  --seed 52 --valid_seed 12345 --collectDiags --diagBins 60 --satThresh 0.05 \
  --name A4_mem_gru_noclip
```

GRU, with clipping (moderate, with gate diagnostics)

```
python -m trainingRNNs_torch.train --task mem --model gru --alpha 0.0 \
  --clipstyle rescale --cutoff 0.05 --diagGates \
  --nhid 50 --lr 0.01 --bs 20 --min_length 50 --max_length 200 \
  --maxiters 50000 --ebs 10000 --cbs 1000 --checkFreq 20 \
  --seed 52 --valid_seed 12345 --collectDiags --diagBins 60 --satThresh 0.05 \
  --name A5_mem_gru_clip005
```

3.3 Task 2: Multiplication (regression)

In this task, the prediction is made using the hidden state at the final time step. Two positions in the sequence have been marked and the ground truth value is their product; the model must predict this. For this task, `classifType=lastLinear`.

RNN (tanh), no clipping

```
python -m trainingRNNs_torch.train --task mul --model rnn --alpha 0.0 \
  --clipstyle nothing \
  --nhid 50 --lr 0.01 --bs 20 --min_length 50 --max_length 200 \
  --maxiters 50000 --ebs 10000 --cbs 1000 --checkFreq 20 \
  --seed 52 --valid_seed 12345 --collectDiags --diagBins 60 --satThresh 0.05 \
  --name B1_mul_rnn_tanh_noclip
```

GRU, no clipping (with gate diagnostics)

```
python -m trainingRNNs_torch.train --task mul --model gru --alpha 0.0 \
  --clipstyle nothing --diagGates \
  --nhid 50 --lr 0.01 --bs 20 --min_length 50 --max_length 200 \
  --maxiters 50000 --ebs 10000 --cbs 1000 --checkFreq 20 \
  --seed 52 --valid_seed 12345 --collectDiags --diagBins 60 --satThresh 0.05 \
  --name B2_mul_gru_noclip
```

3.4 Extra credit

Pascanu's paper reports a favourable performance on the **temporal order** task, using the smart_tanh initialization, gradient clipping and Ω regularization (see Figure 7 in the paper, and read it to find out more about the init and the regularizer). With a cutoff of 0.05 and a regularization alpha of 2.0 or 4.0, the training nll continues to remain stuck at 1.386 for several updates. Instead of this configuration, find a better configuration where training for temporal order actually succeeds, similar to Pascanu's original setup.

Note on Omega regularizer. The starter code contains a Pascanu-style Ω regularizer and a metric called `steps in the past`. **This assignment does not require it unless you are attempting the extra credit.** Run all commands with `--alpha 0.0` if not doing extra credit.

4 How to plot from saved `.npz`

Each run writes {`name`}`_final_state.npz`. Use the following snippet (you may adapt it):

```
import numpy as np
import matplotlib.pyplot as plt

z = np.load("A1_mem_rnn_tanh_noclip_final_state.npz")
grad_time = z["grad_time"] # (num_checkpoints, Tstore), NaN padded
sat_time = z["sat_time"] # (num_checkpoints, Tstore)
valid_err = z["valid_error"] # (num_checkpoints,)
rho = z["rho_Whh"] # (num_checkpoints,)

# choose a checkpoint (e.g., last non-empty)
g = grad_time[-1]; s = sat_time[-1]
g = g[np.isfinite(g)]; s = s[np.isfinite(s)]

plt.figure(); plt.hist(np.log10(g + 1e-12), bins=60); plt.title("log10␣||dL/dh_t||")
plt.figure(); plt.hist(s, bins=60, range=(0,1)); plt.title("hidden␣saturation␣
    distance")
plt.figure(); plt.plot(valid_err); plt.title("validation␣error␣(%)")
plt.show()
```

If `--diagGates` is enabled, also plot: `gate_z_sat_time` and `gate_r_sat_time` histograms.

5 What you must submit

1. **Code:** your modified `model.py` and `train.py` (and any helper code you changed).
2. **Report (PDF):** include the plots and answer the questions below.
3. **Logs and the npz file**: The logs of your training runs, and the saved npz file. We need these to judge your training.
4. **Command log:** a text file listing the exact commands you ran (copy/paste).
5. **AI tool chats**: You will be allowed to access AI tools, but you cannot use them directly to generate code. You may only use them to assist you or for a better understanding of the code. As always, you need to share the links of all AI tool chats. Not doing so constitutes a violation of the course's honor code.

6 Questions to answer in the report

For each required run (A1–A5, B1–B2):

1. Show the histogram of $\log_{10} \|\partial \mathcal{L}/\partial h_t\|$ (at the final checkpoint). Is it concentrated near very negative values (vanishing), very positive values (exploding), or spread?
2. Show the hidden saturation-distance histogram. Is the network saturating? Is saturation *always* related to the vanishing/exploding gradient dynamics?

3. Compare no clipping vs clipping: how do (i) gradient norm, (ii) $\log_{10}(g_t)$ histogram, and (iii) saturation change?

4. Compare RNN vs GRU: do gate saturation histograms show gates near 0/1 (saturated) or near 0.5 (unsaturated)? How does this relate to gradient-through-time behavior? Does the GRU have its own failure modes?

5. Report `rho_Whh` over training. How does it correlate with gradient behavior?

Notes

- Some tasks report a strict `valid error` (e.g., sequence-level error). Always interpret it alongside `valid nll` (and for regression tasks, the MAE/error@thresholds printed by the script).
- Ignore "steps in the past" and Omega unless doing extra credit.
- You don't need GPU at all for this assignment.