**Lecture Notes: RNN Limitations and Improvements**

**1. Recap of Vanilla RNN & BPTT**

* Architecture of vanilla RNN was discussed previously.
* Training is done using Backpropagation Through Time (BPTT).
* Problem: Recurrent gradient multiplications can cause:
  + **Vanishing Gradient** (small gradient values)
  + **Exploding Gradient** (large gradient values)

**2. Practical Limitations of RNN**

* Though RNNs theoretically capture long-range dependencies, they forget distant context in practice due to gradient issues.
* **Exploding Gradient** can be mitigated using **gradient clipping**.
* **Vanishing Gradient** requires more complex solutions, to be discussed.

**3. RNN Use Case: POS Tagging**

* RNN is used for **sequence labeling** tasks like POS tagging.
* Given a sequence of tokens, RNN predicts tags using hidden states:
  + Each hidden state is passed through a linear layer + softmax.
  + Output is a probability distribution over POS tags.
  + Tag is **sampled** from the distribution (not picked as max).

**4. Limitation: Lack of Future Context**

* In sequence labeling, we may need access to both **past and future** context.
* Vanilla RNN only uses past (previous hidden states).
* Since the **entire input is known** (not a generation task), future context can also be used.

**5. Solution: Bidirectional RNN (BiRNN)**

* Two RNNs:
  + One runs **left to right** (forward).
  + One runs **right to left** (backward).
* Each time step has two hidden states (forward + backward).
* These are combined (e.g., concatenated) to get a single hidden state.
* Output prediction is done using this combined hidden state.

**6. RNN for Sentence Classification (e.g., Sentiment Analysis)**

* Use RNN hidden states to classify whole sentences:
  + Option 1: Merge all hidden states (e.g., average, max-pool).
  + Option 2: Use only the **last hidden state**, assuming it summarizes all previous inputs.

**7. Gradient Computation in RNNs**

* Loss at final step is J(θ)J(\theta).
* To compute gradient w.r.t. earlier hidden state (e.g., h1h\_1):
  + Chain rule: Compute gradients through all intermediate states.
  + Each gradient is a **matrix** (derivative of vector w.r.t. vector).
  + Repeated multiplication causes vanishing or exploding gradient.

**8. Example of Long Dependency Problem**

* Sentence: "When she tried to print her tickets, she found... finally printed her \_\_\_"
* Word "tickets" is required for prediction, but is **too far back**.
* Due to vanishing gradient, RNN forgets such distant context.

**9. Fix: Gating Mechanisms**

* Introduce **memory state** (called **C-state**) alongside hidden state.
* Use gated architectures like **LSTM** and **GRU** to control memory flow.
* Helps retain long-term dependencies and mitigate vanishing gradients.

**10. Other Techniques to Improve RNNs**

* **Attention mechanisms**
* **Skip connections / residual connections**

**LSTM (Long Short-Term Memory) - Detailed Lecture Notes**

**Motivation for LSTM**

* Vanilla RNN maintains only a hidden state h\_t at each time step.
* It suffers from vanishing gradient problems and struggles with long-term dependencies.
* **LSTM** introduces an additional **cell state** c\_t, which acts as memory to preserve long-term information.

**Core Idea**

* At each time step t, LSTM maintains:
  + **Hidden state** h\_t: Output of the LSTM at time t.
  + **Cell state** c\_t: Memory of the LSTM used to retain long-term information.
* The content written to c\_t is controlled using a **gating mechanism**:
  + Gates are vectors with values ∈ [0, 1] and control the flow of information.
  + If a gate value = 1 ➔ full information passed; 0 ➔ blocked; [0,1] ➔ partial info passed.

**Types of Gates in LSTM**

**1. Forget Gate f\_t**

* Decides what to **forget** from the previous cell state c\_{t-1}.
* Equation:
* f\_t = σ(W\_f x\_t + U\_f h\_{t-1} + b\_f)

**2. Input Gate i\_t**

* Decides what new information to **store** in the cell.
* Equation:
* i\_t = σ(W\_i x\_t + U\_i h\_{t-1} + b\_i)

**3. Output Gate o\_t**

* Decides what part of the cell state will be output to the hidden state h\_t.
* Equation:
* o\_t = σ(W\_o x\_t + U\_o h\_{t-1} + b\_o)

**Cell State Update**

1. **Generate Cell Candidate**:
2. \tilde{c}\_t = tanh(W\_c x\_t + U\_c h\_{t-1} + b\_c)
3. **Update Cell State**:
4. c\_t = f\_t \* c\_{t-1} + i\_t \* \tilde{c}\_t
   * f\_t \* c\_{t-1} ➔ controls how much of the old memory to retain.
   * i\_t \* \tilde{c}\_t ➔ how much of the new candidate to write.
5. **Update Hidden State**:
6. h\_t = o\_t \* tanh(c\_t)

**Intuition Summary**

* **Forget gate** ➔ How much of old memory to retain.
* **Input gate** ➔ How much new info to add.
* **Output gate** ➔ What part of memory to use as output.

**Training Considerations**

* Gates are parameterized by weight matrices (W, U) and biases (b).
* LSTM has more parameters than vanilla RNN ➔ needs more data to avoid overfitting.

**Avoiding Vanishing Gradients**

* If f\_t = 1 and i\_t = 0, cell state c\_t simply copies c\_{t-1} ➔ past info preserved.
* If f\_t = 0 and i\_t = 1, it overwrites past info completely.

**Design Decisions**

* Use of sigmoid for gates ensures values between 0 and 1.
* tanh is used to generate the cell candidate and to squash the cell state before passing to h\_t.
* These decisions are design-based and not strictly theoretically optimal.

**Comparison with Vanilla RNN**

| **Component** | **Vanilla RNN** | **LSTM** |
| --- | --- | --- |
| Memory | No long-term memory | Cell state (c\_t) |
| Gates | None | Forget, Input, Output |
| Parameters | Few | Many (more W, U, b) |
| Gradient Stability | Poor | Better (avoids vanishing gradient) |

**Diagram Overview (Verbal Description)**

* Input x\_t and hidden state h\_{t-1} feed into all gates.
* Gates generate decisions using learned weights.
* The previous cell state is updated based on the forget and input gates.
* The output hidden state is derived from the updated cell state passed through tanh and scaled by the output gate.

**Conclusion**

* LSTM is well-suited for long sequences due to memory retention.
* Requires careful parameter tuning and sufficient data.
* Crucial for applications like language modeling, machine translation, and time-series prediction.

**Vanishing Gradient Problem in Deep Networks**

**🔹 Issue**

* Not limited to RNNs or LSTMs.
* Occurs in:
  + Deep Feedforward Networks
  + CNNs
  + Simple RNNs
* **Problem**: Bottom layers contribute less to final loss; their gradients diminish over time.

**📌 Solutions to Vanishing Gradient**

**🔹 1. Residual Connections (Skip Connections)**

* Each layer output: f(x)
* Instead of passing just f(x) to next layer, pass:  
  f(x) + x
* **Effect**: Helps model remember input across layers; mitigates forgetting.

**🔹 2. Dense Connections (DenseNet-style)**

* Each layer connects to **all subsequent layers** (forward only).
* Ensures information flow across layers.
* **Example**:  
  Layer 1 connects to Layers 2, 3, 4, ...

**🔹 3. Highway Networks**

* Combination of:
  + **Dense connections**
  + **Gating mechanism**
* Gates decide how much to retain from earlier layers vs. current computation.

**📌 GRU (Gated Recurrent Unit)**

**🔹 Background**

* Proposed in 2014.
* Alternative to LSTM with fewer parameters.
* No **separate cell state** (unlike LSTM).
* Only maintains **hidden state**.

**🔹 Components**

* **Update Gate (z or u)**:
  + Controls how much of previous hidden state to preserve vs. update.
* **Reset Gate (r)**:
  + Controls how much of previous hidden state to use for computing the new content.

**🔹 Computation**

1. Compute update gate u\_t and reset gate r\_t.
2. Compute **new hidden content** (h~\_t) using:
   * Current input x\_t
   * Previous hidden state h\_{t-1} (modulated by r\_t)
3. Final hidden state h\_t is computed as:

cpp

CopyEdit

h\_t = (1 - u\_t) \* h\_{t-1} + u\_t \* h~\_t

**🔹 Parameters**

* 2 parameter sets each for:
  + Update gate (u)
  + Reset gate (r)
  + Hidden content generation (h~)

**📌 LSTM vs GRU**

| **Aspect** | **LSTM** | **GRU** |
| --- | --- | --- |
| States | Cell state + Hidden state | Only Hidden state |
| Gates | Forget, Input, Output | Update, Reset |
| Complexity | More parameters | Fewer parameters |
| Efficiency | Slightly slower | Faster |
| Use Case | Start with LSTM, switch to GRU if data is limited or speed needed |  |
| Performance | No clear winner; varies by task |  |

**📌 Bidirectional RNN (BiRNN)**

**🔹 Architecture**

* Two RNNs:
  + Forward: Left to right
  + Backward: Right to left
* At time t, final hidden state is:  
  concat(h\_forward\_t, h\_backward\_t)

**🔹 Use Case**

* Used when the full sequence is available (e.g., in translation, tagging).

**📌 Multi-layer (Stacked) RNN**

**🔹 Architecture**

* RNNs stacked vertically.
* Output of one RNN layer becomes input to the next.
* Difference from BiRNN:
  + BiRNN processes input at the same timestep from both directions.
  + Multi-layer RNN passes sequence **through layers sequentially**.

**📌 Modern Trends and LSTM Decline**

* LSTM usage declined post-2017 due to **Transformers**.
* Yet, LSTM/RNNs are making a comeback in:
  + **State-space models (e.g., Mamba)**
  + Sequence tasks requiring **temporal continuity**

**📌 RNN Limitation**

* **Sequential Processing**:
  + Cannot parallelize sequence processing.
  + Must compute hidden state at t after t-1 is processed.