Artificial Neural Networks - Deep Learning

Recurrent Neural Networks

Unat Teksen

1 Why We Need RNNs (Static vs. Sequential Data)

1.1 Static Datasets (Feedforward Mindset)

- One input vector x per example.
- Output computed as $y = g(x; w)$ (where w represents weights).
- Examples treated as strictly independent.
- Data order does not matter.
- Visualized as a single pass through layers.

1.2 Dynamic / Sequential Datasets

- Input is a sequence: $X_0, X_1, X_2, \ldots, X_t$.
- Each X_t is a feature vector $(x_1, \ldots, x_I)$.
- Meaning heavily depends on time/order (e.g., speech, text, sensor signals).
- Inputs are indexed by time.
- Requires the model to use history.
- **Main Takeaway: Sequence modeling** relies on ordered inputs, not i.i.d. points. **Feedforward nets** are for independent examples. Sequences require models handling **dependencies across time**. Notation: X_t = observation at time t; components x_i.

2 Two Families for Sequence Data

2.1 1. Memoryless (No Internal State)

- Do not store history internally.
- Past information must be explicitly provided.
- **Autoregressive approach:** Predict next value from a fixed past window.
- **Feedforward NNs:** Use nonlinear layers on a fixed window.
- **Core limitation:** Fixed history length. Misses important info older than the window.

2.2 2. Models with Memory (Internal State)

- Maintain a hidden state evolving over time.
- State "summarizes" the past.
- **Classical families:** Linear Dynamical Systems, HMMs.
- **Neural families:** RNNs.
- **Main Takeaway: Memoryless** = explicit past window. **Memory-based** = learns a hidden state carrying info forward. RNNs are needed for **variable-length memory** without manually choosing window sizes.

3 State-Space Model Intuition

3.1 Linear Dynamical Systems (LDS)

- Assumes an unobserved hidden state.
- Hidden state evolves over time (state transition).
- Observations generated from hidden state.
- Inputs act as external "driving" signals.
- State is **continuous-valued** (real numbers).
- Uncertainty/noise is **Gaussian**.
- Transitions and outputs are **linear**.
- **Inference tool:** Kalman filter.

Mathematical Structure:

$$\text{State update:} \quad s_t = As_{t-1} + Bx_t + \epsilon_t \tag{1}$$

$$\text{Output:} \quad y_t = Cs_t + \delta_t \tag{2}$$

- **Main Takeaway:** Know the LDS structure. **Kalman filter** applies to **linear + Gaussian** assumptions. This sets up the **"hidden state"** idea reused by RNNs (with learned nonlinear updates).

3.2 Hidden Markov Models (HMM)

- Same template: evolving hidden state → outputs.
- Hidden state is **discrete** (finite set of states).
- Transitions are **probabilistic** via transition matrix: $P(z_t \mid z_{t-1})$.
- Outputs are **stochastic** ("emissions"): $P(y_t \mid z_t)$.
- **Inference tool:** Viterbi algorithm (most likely hidden-state sequence).
- **Main Takeaway:** Know **transition probabilities** and **emissions**. **Viterbi** = "best path" decoding. RNNs are state models where the state is **continuous, distributed, and learned**.

4 What is an RNN?

4.1 State as Learned, Distributed Memory

- Memory achieved by feeding previous internal representation back.
- Hidden state is **distributed** across many units (efficient representation).
- Update is **nonlinear** (enables complex dynamics).
- Highly expressive given enough computational resources.

4.2 Core Equations

- Computes hidden representation using current input + previous state.
- Computes output from current hidden state.

Standard Simplified Form:

$$h_t = \phi(W_x x_t + W_h h_{t-1} + b) \tag{3}$$
$$y_t = \psi(W_y h_t + c) \tag{4}$$

- **Main Takeaway: State recurrence** is the defining feature. **Weight sharing** across time is implied: same W_x, W_h, W_y for all t. RNNs are neural state-space models with **learned, nonlinear updates**.

5 Backpropagation Through Time (BPTT)

5.1 Unrolling the Network

- Converts RNN to a computation graph (like deep feedforward nets).
- Creates one copy per time step.
- Connections link time $t-1$ to t.
- **Critical point:** Uses the **same parameters** (shared weights) at each step.

5.2 Truncated BPTT

- Unroll for a fixed number of steps U (not the whole sequence).
- Compute gradients at each unrolled step.
- Combine (average) gradient contributions.
- Update shared parameters.

Mathematical Form:

$$\nabla W = \frac{1}{U} \sum_{u=0}^{U-1} \nabla W_{(t-u)} \tag{5}$$
$$W \leftarrow W - \eta \nabla W \tag{6}$$

- **Main Takeaway: BPTT** = backprop on the unrolled graph. **Truncated** = backprop only U steps into the past. Training relies on **shared weights** across time.

6 Long-Term Dependencies & Vanishing Gradients

6.1 The Problem

- Output at time t often depends on information from far earlier.
- Early context heavily affects later prediction.

6.2 Why Gradients Die (Math)

- Backprop from time t to earlier time k includes a product of Jacobians:

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \tag{7}$$

- If each factor's norm < 1, the product shrinks exponentially with distance $(t-k)$:

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| \leq (\gamma)^{t-k} \tag{8}$$

- If $\gamma < 1$, gradient $\to 0$ (**vanishing gradient**).
- **Activations:** Sigmoid/tanh derivatives have magnitude < 1 (especially when saturated), causing $\gamma < 1$.

6.3 Attempted Fixes

- **Quick fix (ReLU):** Derivative is 0 or 1. Keeps gradient flowing when active. Doesn't fully solve memory control.
- **Architecture fix:** Modules with self-connection weight $= 1$ (linear). Preserves info without shrinkage. Too crude without smart gating.
- **Main Takeaway:** Be able to derive the **product-of-derivatives**. **Vanishing gradients** make learning long-term dependencies hard. **Gating-based architectures** (LSTM/GRU) were invented to solve this via gated memory paths.

7 Long Short-Term Memory (LSTM)

7.1 Big Picture

- Introduces a **memory cell**.
- Can write, keep/forget, and read information.
- Path through cell state behaves close to linear (controlled scaling).
- Makes BPTT feasible over long sequences.

7.2 Components and Equations

- **Cell state (C_t):** Long-term memory.
- **Hidden state (h_t):** Exposed output.
- **Gates:** Use sigmoid (σ) outputs in $[0, 1]$.

$$\text{Input gate (write new info):} \quad i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{9}$$
$$\text{Candidate content (proposed memory):} \quad \tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C) \tag{10}$$
$$\text{Forget gate (keep old memory):} \quad f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{11}$$
$$\text{Cell update (keep + write):} \quad C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{12}$$
$$\text{Output gate (expose memory):} \quad o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{13}$$
$$\text{Hidden output:} \quad h_t = o_t \odot \tanh(C_t) \tag{14}$$

- **Main Takeaway:** Memorize the C_t **update equation** (the centerpiece). C_t = long-term memory highway, h_t = output features. Gradients flow if $f_t \approx 1$ (preserves info). Solves long-term dependencies via **controlled gating**.

8 Gated Recurrent Unit (GRU)

- Combines "input" and "forget" into one **update gate**.
- Merges cell state and hidden state (no separate C_t and h_t).
- Fewer parameters $\rightarrow$ faster training.
- Often comparable performance to LSTM.
- **Main Takeaway:** GRU has **fewer gates** and **no separate cell state**. Preferred for smaller models and faster training. A streamlined model addressing vanishing gradients via gating.

9 RNNs as Computation Graphs

- Repeated computational block across time.
- Same transformation applied each time step.
- Overall network is a large differentiable graph.
- BPTT is standard backprop on this unrolled graph.
- **Main Takeaway:** Draw/describe the unrolled chain using **shared parameters each step**. Gradients flow through the **time-unrolled computation graph**.

10 Stacked and Bidirectional RNNs

10.1 Stacked (Deep) RNN/LSTM

- Recurrent layers are **stacked**.

- Output of layer 1 at time t is input to layer 2 at time t.
- Nonlinearity (e.g., ReLU) applied between layers.
- **Lower layers:** Learn "local"/surface patterns (e.g., n-grams).
- **Higher layers:** Learn abstract patterns (e.g., phrase/sentence structure).
- **Main Takeaway:** Stacking increases **representational capacity** and learns **hierarchical temporal features**. Time recurrence within layers + depth-wise feed-forward between layers.

10.2 Bidirectional RNNs (BiRNN)

- Runs two separate RNNs over the input.
- One processes left $\rightarrow$ right (past to future).
- One processes right $\rightarrow$ left (future to past).
- Combines (usually **concatenates**) hidden states:

$$h_t^{bi} = [\, h_t^{\rightarrow} \; ; \; h_t^{\leftarrow} \,] \tag{15}$$

- **Valid for:** Offline tasks (entire sequence available). Tagging, NER, text classification.
- **Invalid for:** Causal/online settings (future unavailable). Real-time prediction.
- **Main Takeaway: Concatenate** is most common. Helps when task benefits from **future context**. Cannot be used for **causal real-time prediction**. BiRNN + Stacked = two chains across time + multiple layers.

11 Initial Hidden State Choices

- RNNs need an initial state (h_0, and C_0 for LSTM).
- **Fixed constant:** Usually zeros (simplest).
- **Learned parameters:** Treat h_0, C_0 as trainable vectors.
- **Random initialization:** Start random, optimize via backprop.
- **Backprop to initial state:** Unrolled state influences all steps. Gradients flow back to h_0, learning a good default starting memory.
- **Main Takeaway:** Learning h_0 encodes a **useful prior context** and improves training stability/performance.

12 Sequence Problems (Karpathy Taxonomy)

- **A) Fixed input $\rightarrow$ fixed output:** Image classification.
- **B) Fixed input $\rightarrow$ sequence output:** Image captioning (single image $\rightarrow$ variable text).

- **C) Sequence input → fixed output:** Sentiment analysis (variable text → one label).
- **D) Sequence input → sequence output (unaligned):** Machine translation (input length $m \neq$ output length n).
- **E) Synced sequence input/output (aligned):** Frame-level video labeling (aligned per time step).
- **Main Takeaway:** Name the **5 settings** and give examples. Note the distinction between **aligned vs. unaligned** seq2seq.

13 Seq2Seq Conditional Language Modeling

13.1 Core Goal in Seq2Seq

- Maximize conditional probability of output y given input x:

$$y^* = \arg\max_y P(y \mid x) \tag{16}$$

- Parameterized learning:

$$y' = \arg\max_y P(y \mid x; \theta) \tag{17}$$

- Model learns a distribution over whole sequences, not independent tokens.

13.2 Language Model vs. Conditional LM

- **Language Model (LM):** Assigns probability to a sentence.

$$P(y_1, \ldots, y_n) = \prod_{t=1}^{n} p(y_t \mid y_{<t}) \tag{18}$$

- **Conditional LM:** Conditions on source x.

$$P(y_1, \ldots, y_n \mid x) = \prod_{t=1}^{n} p(y_t \mid y_{<t}, x) \tag{19}$$

- **Main Takeaway:** Write both **factorization formulas**. $y_{<t}$ = all previous generated tokens. Seq2seq decoding is **conditional next-token prediction**.

14 Encoder-Decoder Framework

- **Encoder:** Reads source sequence $x_1, \ldots, x_m$. Produces a representation (final hidden state or sequence of states).
- **Decoder:** Generates output tokens one by one. Uses previous generated tokens (history) + encoded source.
- **Concept:** Compresses source into a "context" conditioning the decoding.
- **Main Takeaway: Encoder** maps variable-length input to representation; **Decoder** maps representation + history to variable-length output. Common because tasks often map inputs to a **latent representation** then decode.

15 Training a Seq2Seq Model

15.1 Probabilities and Cross-Entropy Loss

- Predicts probability distribution p_t over vocabulary V at time t. Vector length is $|V|$.
- **Per-step loss (Cross-entropy):** True token y_t is one-hot.

$$\text{loss}_t = -\sum_{i=1}^{|V|} y_t^{(i)} \log p_t^{(i)} = -y_t^\top \log(p_t) \tag{20}$$

- Simplifies to minus log probability of the correct token.
- **Sequence loss:** Total sum over time.

$$L = -\sum_{t=1}^{n} y_t^\top \log(p_t) \tag{21}$$

15.2 Teacher Forcing vs. Inference

- **Training (Teacher Forcing):** Decoder is fed the **ground-truth** previous token. Stabilizes and speeds up learning.
- **Inference:** Ground truth unknown. Decoder is fed its **own predicted token**.
- **Exposure Bias:** Model trains on perfect histories but tests on its own imperfect histories.
- **Main Takeaway:** Explain **teacher forcing** and the train/test mismatch (**exposure bias**). Cross-entropy with one-hot equals $-\log p(\textbf{correct})$.

16 Special Tokens Batching Pipeline

16.1 Special Tokens

- **PAD:** Makes sequences equal length for batching. Must be masked in loss.
- **EOS:** Marks end of sequence (when decoder should stop).
- **UNK:** Out-of-vocabulary words.
- **SOS / GO:** Start token fed to decoder at first step.

16.2 Batch Preparation Steps

- Sample (source, target) pairs.
- Append EOS to source (encoder knows where it ends).
- **Decoder input:** Prepend SOS.
- **Decoder label:** Append EOS.

- **Shifted Setup:** Input = `SOS y1 ... y(n-1)`. Label = `y1 ... y(n) EOS`.
- Pad within batch to max lengths using PAD. Convert to IDs. Mask PAD in loss.
- **Main Takeaway: EOS** indicates termination and enables variable lengths. **PAD** makes lengths equal (must **mask PAD** in loss). Be ready to explain the **"shifted" decoder input/output trick**.

17 Decoding Strategies (Greedy vs. Beam Search)

- Searching all sequences is computationally exponential.

17.1 Greedy Decoding

- Pick the single most probable next token at each step:

$$y_t = \arg\max_{\text{token}} P(\text{token} \mid y_{<t}, x, \theta) \tag{22}$$

- **Problem:** Local best choices can lead to a globally worse sentence. No backtracking (cannot undo early mistakes).

17.2 Beam Search

- Keep top B partial sequences (**beam width**).
- Expand each by possible next tokens.
- Keep the best B by score (sum of log-probs).
- Continue until EOS.
- **Log probability use:** Product of probs becomes sum of log probs. Numerically stable.

$$\arg\max \prod_t p_t \quad \equiv \quad \arg\max \sum_t \log p_t \tag{23}$$

- **Main Takeaway: Greedy** picks single best token; **Beam** keeps top B paths. Greedy is **not optimal** (no backtracking). Beam width tradeoff: **larger beam → better search but slower**.