Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of **neural networks** that **process sequences** of **inputs** by **maintaining** a **hidden state** that **depends** on the **previous inputs** and **outputs**.

- Much of the **data** we want to **process** is **sequential** (e.g. **time series**, **sentences**, **videos**, **audio**):

$$h_t = g(Vx_t + Uh_{t-1} + c)\hat{y}_t = Wh_t + b$$

- x_t is the **input** at **time step** t;
- h_t is the **hidden state** at **time step** t - it **encodes** the **history** of the **sequence** up to **time step** t;
- y_t is the **output** at **time step** t;
- V, U, W are **weight matrices**;
- c, b are **bias vectors**.

But how do we **train** such a **network**?

- **Backpropagation** can be used to **compute the gradients** of the **loss function** with respect to the **parameters - chain rule**;
- Parameters are **shared** across **time steps - backpropagation through time** - BPTT:

$$\frac{\partial L}{\partial U} = \sum_{t=1}^{T} \frac{\partial L}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial U}$$

Standard Applications of RNNs

- **Sequence Generation**: generate a sequence of outputs given a sequence of inputs; e.g. **language modeling**;
- **Sequence Tagging**: generate a sequence of outputs given a sequence of inputs; e.g. **part-of-speech tagging**;
- **Pooled Classification**: generate a single output given a sequence of inputs by pooling the hidden states; e.g. **text classification**.

Sequence Generation

- The **full history model**: $p(y_1, \ldots, y_L) = \prod_{t=1}^{L+1} p(y_t|y_{t-1}, \ldots, y_0)$; - generates each word depending on **all previous words** - thats huge expressive power, but there are **too many parameters to train**;
- **Markov models** avoid this problem by using **limited memory**: $p(y_t|y_{t-1}, \ldots, y_0) \approx p(y_t|y_{t-1}, \ldots, y_{t-m})$; - an **order-$m$ Markov model**;
- Another alternative is to consider **all** the history, but **compress** it into a **fixed-length vector - recurrent neural networks**.

Auto-Regressive Models

- **Auto-regressive models** are a class of **models** that **predict** a **sequence** of **outputs** by **conditioning** on **previous outputs**;
- **Key idea**: feed the **previous output** as **input** to the **next step**: $x_t =$ embedding of y_{t-1};
- Maintain a **state vector** h_t that **encodes** the **history** of the **sequence** up to **time step** t **- compresses all the history**: $h_t = f(x_t, h_{t-1})$;
- To compute the next probability distribution: $p(y_t = i|y_{t-1}, \ldots, y_0) = softmax(Wh_t + b)$;
  - To generate text, y_t is a word in a **large vocabulary - softmax** is **expensive**;

Algorithms are needed for:

- **Sampling sequences** from the probability distribution - **easy**;
  - Compute h_t from h_{t-1} and x_t;
  - Sample y_t from $softmax(Wh_t + b)$;
  - Repeat until the end-of-sequence token is generated;
- Obtaining the **most probable sequence - hard**;
  - Find the sequence $y_1, \ldots, y_L$ that **maximizes** $(softmax(Wh_1+b))_{y_1} \times \cdots \times (softmax(Wh_L + b))_{y_L}$;
  - Picking the best y_t **greedily** does not work;
  - This is important fro **conditional language modeling**.
- **Training** the RNN (i.e. **learning** the **parameters** W, U, V, b, c);
  - Usually **maximum likelihood estimation** is used;
  - In other words, **minimize** the **negative log-likelihood** of the **training data - cross-entropy loss**:
    * $L(\theta; y_1, \ldots, y_L) = -\frac{1}{L+1} \sum_{t=1}^{L+1} logp(y_t|y_{t-1}, \ldots, y_0)$, where $\theta = \{W, U, V, b, c\}$;
  - This is equivalent to minimizing **perplexity**: $expo(L(\theta; y_1, \ldots, y_L))$.

Sequence Tagging

- Input: a sequence of words $x_1, \ldots, x_L$;
- Goal: assign a **tag** to each element of the sequence, yielding and output sequence $y_1, \ldots, y_L$;
- **Examples**: POS tagging, named entity recognition, etc.
- Different from **sequence generation** because the input and the output are distinct (no need for auto regression), and the length of the output is known;
- **POS tagging** maps a sequence of words to a sequence of POS (part-of-speech) tags; example:
  - Input: "I am a student";
  - Output: "PRON VERB DET NOUN";
  - Can be implemented using RNNs, and improved by using **bidirectional RNNs - two RNNs** are used, one **forward** and one **backward**.

Pooled Classification

- Predict a **single label** for a sequence of inputs;
- **Pool** the RNN hidden states into a **fixed-length vector** and use a single softmax to output the final label;
- There are some **pooling strategies**:
  - **Last hidden state**: h_L; **- simple**, but **losses** the **history**;
  - Average pooling;
  - Use a bidirectional RNN and combine both last states;
  - ...

GRUs and LSTMs - The Vanishing Gradient Problem

- As we go **back in time**, the **gradient decreases** exponentially;

$$\prod_t \frac{\partial h_t}{\partial h_{t-1}} = \prod_t \frac{\partial h_t}{\partial z_t} \frac{\partial z_t}{\partial h_{t-1}} = \prod_t Diag(g'(z_t))U$$

Three cases:

- **Eigenvalues** of U are **all greater than 1**: **exploding gradients**;
  - Dealt with by **gradient clipping -** truncate the gradient if it is too large;
- **Eigenvalues** of U are **all smaller than 1**: **vanishing gradients**;
  - **Long-term dependencies** are **hard to learn**;
  - **Solutions**:
    * Better optimizers;
    * Normalization to keep the gradient norms stable across time;
    * Clever initialization to start with good spectra;
    * **Alternative parameterizations**: **GRUs** and **LSTMs** - instead of multiplying across time, we want the error to be **approximately constant across time**.
- **Eigenvalues** of U are exactly 1: **gradient propagation stable**.

Gradient Clipping

- **Gradient clipping** is a technique used to **prevent exploding gradients**;
- **Idea**: if the **gradient norm** is **larger** than a **threshold**, **rescale** it to the **threshold**:

$$\tilde{\nabla} \leftarrow \begin{cases} \frac{\nabla}{||\nabla||} \times \text{threshold} & \text{if } ||\nabla|| > \text{threshold} \\ \nabla & \text{otherwise} \end{cases}$$

- **Element-wise clipping** is also possible: $\tilde{\nabla}_i \leftarrow \min threshold, |\nabla_i| \times sign(\nabla_i)$.

GRUs - Gated Recurrent Units

- The error must backpropagate through all the intermediate states;
- **Key idea**: create some **shortcuts** to **skip** some of the **intermediate states - adaptive shortcuts** controlled by **gates**;
- $h_t = u_t \odot \tilde{h_t} + (1 - u_t) \odot h_{t-1}$, where:
  - u_t is the **update gate -** controls how much of the previous state is kept: $u_t = \sigma(V_u x_t + U_u h_{t-1} + b_u)$;
  - $\tilde{h_t}$ is the **candidate state** - the new state; $\tilde{h_t} = g(V x_t + U(r_t \odot h_{t-1}) + b)$;

        * r_t is the **reset gate** - controls how much of the previous state is forgotten: $r_t = \sigma(V_r x_t + U_r h_{t-1} + b_r)$;
    - h_{t-1} is the **previous state**;
    - $\odot$ is the **element-wise product**.

LSTMs - Long Short-Term Memory

- **Key idea**: use **memory cells** c_t to **store** information for **long periods of time**;
- To avoid the multiplicative effect, we use **addition** instead of **multiplication**;
- Control the flow with special gates: **input**, **forget** and **output** gates;
- $c_t = f_t \odot c_{t-1} + i_t \odot g(Vx_T + Uh_{t-1} + b)$, where:
    - f_t is the **forget gate** - controls how much of the previous state is forgotten: $f_t = \sigma(V_f x_t + U_f h_{t-1} + b_f)$;
    - i_t is the **input gate** - controls how much of the candidate state is added to the memory cell: $i_t = \sigma(V_i x_t + U_i h_{t-1} + b_i)$;
    - g is the **candidate state** - the new state;
    - h_{t-1} is the **previous state**: $h_t = o_t \odot g(c_t)$;
    - $\odot$ is the **element-wise product**.

Beyond Sequences

There are extensions of RNNs for non-sequential structures (e.g. trees and images): **recursive neural networks** and **PixelRNNs**.

Recursive Neural Networks

- **Recursive neural networks** are a class of **neural networks** that **process trees** of **inputs** by **maintaining** a **hidden state** that **depends** on the **previous inputs** and **outputs**;
- Assume a **binary tree** structure;
- Propagate states bottom-up in the tree, computing the parent state p from the children states c_1 and c_2: $p = tanh(W[c_1; c_2] + b)$;
- Use the same parameters at all nodes;
- **Tree-LSTM** is a variant of **LSTM** that uses **tree structures** instead of **sequences**.

Pixel RNNs

- **Pixel RNNs** are a class of **neural networks** that **process images** by **maintaining** a **hidden state** that **depends** on the **previous inputs** and **outputs**;
- They can be used as auto-regressive models for **image generation**;

More Tricks of the Trade

- **Depth** in recurrent layers helps in practice (2-8 layers seem to be standard);
  - Input connections may or may not be used;
- Apply **dropout** between layers, but not on recurrent connections;
- For **speed**:
  - Use diagonal matrices instead of full matrices;
  - Concatenate parameter matrices for all gates and do a single matrix-vector multiplication;
  - Use optimized implementations;
  - Use GRUs or reduced-gate LSTMs;
- For **learning speed and performance**:
  - Initialize so that the bias on the forget gate is large;
  - Use random orthogonal matrices to initialize the square matrices.