Overview

1. **Recurrent Neural Networks**
   - What are RNNs? (10.1, 10.2)
   - Computing gradients (10.2.2)

2. **Challenge 1: Long-Term Dependencies**
   - Vanishing or exploding gradients (10.7)

3. **Approaches to learn long-term dependencies**
   - Multiple Time Scale Strategies (10.9)
   - ESN and Leaky Units (10.8, 10.9)
   - Simple, but effective way (Le et al., 2015)
   - Gated RNNs: LSTM, GRUs (10.10)
   - Solution for exploding - gradient clipping (10.11)
   - Solution for vanishing - regularization (10.12)

4. **Challenge 2: Memorizing Facts**
   - Explicit memory, NTM (10.13)
Overview

5 Variations of RNNs

- Feedback only from o to h (10.2.1)
- Removing conditional independence (10.2.4)
- Bidirectional RNNs (10.3)
- Single output (10.2)
- Single input (10.2.4)
- Encoder-Decoder Sequence-to-Sequence Arch. (10.4)
- Only output sequence (10.2.3)
- Deep RNNs (10.5)
- Recursive NNs (10.6)
1. Recurrent Neural Networks
What are RNNs?
What are RNNs?

- Handling sequential data
- Memory
  - You can think of the hidden state $h^{(t)}$ as the (lossy) “memory” of the network, which captures information in all the previous time steps.
- Sharing parameters
  - RNN shares the same parameters ($U, V, W$ above) across all steps. This reflects the fact that we are performing the same task at each step, just with different inputs → “recurrent”
- In theory, can be very deep, but in practice, are limited to look back only a few steps
Express the RNN as a unrolled computational graph, and then apply the back-propagation algorithm (BPTT).

The runtime is $O(\tau)$ and cannot be reduced by parallelization because it is inherently sequential; each time step may only be computed after the previous one.

The memory cost is also $O(\tau)$ as states computed in the forward pass must be stored until they are reused during the backward pass.
Example 1.1 (BPTT)

\[ a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)} \]
\[ h^{(t)} = \text{tanh}(a^{(t)}) \]
\[ o^{(t)} = c + Vh^{(t)} \]
\[ \hat{y}^{(t)} = \text{softmax}(o^{(t)}) \]

(1)

\[ L(\{x^{(1)}, ..., x^{(\tau)}\}, \{y^{(1)}, ..., y^{(\tau)}\}) = \sum_{t} L^{(t)} = \sum_{t} - \log \hat{y}_{y^{(t)}} \]

(2)

\[ \rightarrow \nabla_c L, \nabla_b L, \nabla_V L, \nabla_W L, \nabla_U L? \]
Computing Gradients - Example

\[ \frac{\partial L}{\partial L(t)} = 1 \]

\[ (\nabla_{o(t)} L)_i = \frac{\partial L}{\partial o_i(t)} = \frac{\partial L}{\partial L(t)} \frac{\partial L(t)}{\partial o_i(t)} = 1 \star \frac{\partial}{\partial o_i(t)}(-\log \text{softmax}(o^{(t)}_{y(t)})) = \frac{\partial}{\partial o_i(t)} \left( \log \sum_j \exp(o_j^{(t)}) - o_{y(t)}^{(t)} \right) = \hat{y}_i^{(t)} - 1_{i,y(t)} \]

\[ \nabla_{h(\tau)} L = (\nabla_{o(\tau)} L) \frac{\partial o^{(\tau)}}{\partial h^{(\tau)}} = (\nabla_{o(\tau)} L) V \quad (t = \tau) \]

\[ \nabla_{h(t)} L = (\nabla_{h(t+1)} L) \frac{\partial h^{(t+1)}}{\partial h^{(t)}} + (\nabla_{o(t)} L) \frac{\partial o^{(t)}}{\partial h^{(t)}} \]

\[ = (\nabla_{h(t+1)} L) \text{diag}(1 - (h^{(t+1)})^2) W + (\nabla_{o(t)} L) V \quad (t < \tau) \]
Computing Gradients - Example

\[ \nabla_c L = \sum_t (\nabla_{o(t)} L) \frac{\partial o(t)}{\partial c} = \sum_t \nabla_{o(t)} L \]

\[ \nabla_b L = \sum_t (\nabla_{h(t)} L) \frac{\partial h(t)}{\partial b} = \sum_t (\nabla_{h(t)} L) \text{diag}(1 - (h(t))^2) \]

\[ \nabla_V L = \sum_t (\nabla_{o(t)} L) \frac{\partial o(t)}{\partial V} = \sum_t (\nabla_{o(t)} L) h(t)^T \]

\[ \nabla_W L = \sum_t (\nabla_{h(t)} L) \frac{\partial h(t)}{\partial W} = \sum_t (\nabla_{h(t)} L) \text{diag}(1 - (h(t))^2) h(t-1)^T \]

\[ \nabla_U L = \sum_t (\nabla_{h(t)} L) \frac{\partial h(t)}{\partial U} = \sum_t (\nabla_{h(t)} L) \text{diag}(1 - (h(t))^2) x(t)^T \]

(\nabla_{h(t)} L\) indicates the full influence of \(h(t)\) through all path from \(h(t)\) to \(L\), whereas \(\frac{\partial h(t)}{\partial W}\) refers to the effect of \(W\) on \(h(t)\) only via the use of \(W\) at \(t\).)
2. Challenge 1: Long-Term Dependencies
Vanishing or exploding gradients

- Gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization).
- Let’s see a simple example.

\[ h^{(t)} = Wf(h^{(t-1)}) + Ux^{(t)} + b \]
\[ \hat{y}^{(t)} = Vf(h^{(t)}) \]
Vanishing or exploding gradients

- For analysis, apply chain rule rather than back-propagation:

\[
\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial W}
\]

\[
\frac{\partial L^{(t)}}{\partial W} = \sum_{k=1}^{t} \frac{\partial L^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W}
\]

- More chain rule:

\[
\frac{\partial h^{(t)}}{\partial h^{(k)}} = \prod_{j=k+1}^{t} \frac{\partial h^{(j)}}{\partial h^{(j-1)}} = \prod_{j=k+1}^{t} W^T \text{diag}[f'(h^{(j-1)})]
\]
Vanishing or exploding gradients

If we define $\beta$’s as upper bounds of the norms,

\[ \left\| \frac{\partial h^{(j)}}{\partial h^{(j-1)}} \right\| \leq \left\| W^T \right\| \left\| \text{diag}[f'(h^{(j-1)})] \right\| \leq \beta_w \beta_h \]

\[ \Rightarrow \left\| \frac{\partial h^{(t)}}{\partial h^{(k)}} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h^{(j)}}{\partial h^{(j-1)}} \right\| \leq (\beta_w \beta_h)^{t-k} \]

This can become very small or very large quickly!
3. Approaches to learn long-term dependencies
One way to deal with long-term dependencies is to design a model that operates at multiple time scales.

- Some parts of the model operate at fine-grained time scales and can handle small details.
- Other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently.

Adding skip connections through time

- RNN with longer delays
- The gradients of recurrent connections with a time-delay $d$ diminish exponentially as a function of $\frac{\tau}{d}$, while those of single steps connections still a function of $\tau$

Removing connections

- Actively remove length-one connections and replacing them with longer connections
In the previous section, we found that the problem mainly occurs due to the value of the norm of Jacobians \( \| J(t) \| = \| \frac{\partial h^j}{\partial h^{(j-1)}} \| \).

Intuitively, to prevent gradients from vanishing or exploding, we have to make \( \| J(t) \| \) not too small and not too big. \( \approx 1 \).
Echo State Networks and Leaky Units

- **Echo State Networks**
  - Sets the input and recurrent weights properly (by fixing the spectral radius of recurrent parameter $W$) so that the recurrent hidden units capture past information well, and then only learn the output weights.
  - **How to train?**
    - Randomly construct a RNN: # of layers, (sparse) $U$ & $W$
    - Renormalize the spectral radius of $W$: $W \leftarrow \lambda \frac{W}{\lambda_M}$
    - Train only the output weights

- **Leaky (Integration) Units**
  - hidden recurrent units with linear-self connections and a weight near 1 on these connections
  - $h(t) = \alpha h(t) + (1 - \alpha) f(b + Wh(t-1) + Ux(t))$
  - $\alpha$ is near one, the information is remembered for a long time
  - $\alpha$ is near zero, the information is discarded rapidly
  - $\alpha$ can be chosen manually or learned
Simple, but effective way

- Another way is to initialize the recurrent weight matrix $W$ properly, or to select proper activation functions like ReLU
- Combined: “A Simple Way to Initialize Recurrent Networks of Rectified Linear Units (Le et al., 2015)”
  - Initialize $W$ to be $I$ and biases to be zero and use ReLUs
  - Good performance in the MNIST classification experiment, where the sequential inputs are 784 pixels, the output is the category, and the networks read one pixel at a time (784 time steps!)
As of now, the most effective models based on the idea of making the product of gradients is close to one are gated RNNs, which include long short-term memory and gated recurrent units.

**Leaky Units vs. Gated RNNs**

- Leaky units \( h^{(t)} = \alpha h^{(st)} + (1 - \alpha)f(b + Wh^{(t-1)} + Ux^{(t)}) \) have the same \( \alpha \) over time, whether it is chosen manually or learned, whereas gated RNNs allow the connection weights to change at each time step.
- Leaky units only accumulate information, but gated RNNs can forget the old state.
- The scalar value \( \alpha \) can be either near one or near zero, but cannot be both at the same time. Gated RNNs achieve this using gate units passed a sigmoid layer.
The rest slides about LSTMs are based on Chris Olah’s excellent post: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Slightly different notations: in LSTM, $C_t, i_t, o_t = s^{(t)}, g^{(t)}, q^{(t)}$ in the book, and in GRUs, $z_t = 1 - u^{(t)}$ in the book.
LSTM - Standard RNN vs. LSTM

\[ h_{t-1} \rightarrow A \rightarrow h_t \rightarrow A \rightarrow h_{t+1} \]

\[ X_{t-1} \rightarrow A \rightarrow X_t \rightarrow A \rightarrow X_{t+1} \]
LSTM - Core idea

- The key to LSTMs is the cell state units having an internal recurrence (linear self-loop), which work as a conveyor belt of information.
- LSTMs have the ability to remove or add information to the cell state, carefully regulated by structures called gates (= forget/input gates).

Gates output numbers between zero and one, describing how much of each component should be let through.
LSTM - Forget gate

- It looks at $h_{t-1}$ and $x_t$ and outputs a number between 0 and 1 for each number in the cell state $C_{t-1}$.
- $1 = \text{“completely keep”, 0 = “completely get rid of”}$

$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$
LSTM - Input gate

- It decides what new information we will store in the cell state.
- First, a sigmoid layer called the “input gate layer” decides which values we will update.
- Next, a tanh layer (or sigmoid layer like (10.26) in the book) creates a vector of new candidate values $\tilde{C}_t$ that could be added to the state.
- We will combine these two to create an update to the state.

\[
i_t = \sigma (W_i \cdot [h_{t-1}, x_t] + b_i)
\]
\[
\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)
\]
• Update the old cell state $C_{t-1}$ by
  • multiplying $f_t$ to forget the things, and
  • addying $i_t \ast \tilde{C}_t$, the new candidate values

$$C_t = f_t \ast C_{t-1} + i_t \ast \tilde{C}_t$$
LSTM - Output gate

- It is a filtered version of the cell state $C_t$.
- First, a sigmoid layer decides what parts of the cell state to output.
- Then, the cell state goes through tanh, in order to push the values to be between -1 and 1.
- We obtain the final output by multiplying the two.

\[
o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)
\]
\[
h_t = o_t \times \tanh(C_t)
\]
The main difference with LSTM is that a single gating unit called "update gate" \( (=z_t) \) simultaneously controls the forgetting factor and the decision to update the state unit.

The reset gate \( (=r_t) \) controls which parts of the state get used to compute the next target state \( (=\tilde{h}_t) \), introducing an additional nonlinear effect in the relationship between past state and future state.

GRUs also merge the cell state and hidden state.

The mathematical equations are as follows:

\[
\begin{align*}
    z_t &= \sigma (W_z \cdot [h_{t-1}, x_t]) \\
    r_t &= \sigma (W_r \cdot [h_{t-1}, x_t]) \\
    \tilde{h}_t &= \tanh (W \cdot [r_t \ast h_{t-1}, x_t]) \\
    h_t &= (1 - z_t) \ast h_{t-1} + z_t \ast \tilde{h}_t
\end{align*}
\]
The objective function for RNNs often contains sharp nonlinearities in parameter space result from the multiplication of several parameters. When the parameter gradient is very large, GD could throw the parameters very far into a region where the objective function is larger.

Ways to clip the gradient $g$ (per-minibatch)

- Clip $\|g\|$ just before the parameter update if $\|g\| > \nu$ by $g \leftarrow \frac{g\nu}{\|g\|}$
- Clip $g$ element-wise
- take a random step if $\|g\| > \nu$
Solution for vanishing - regularization

- We would like the gradient vector $\nabla_{h(t)} L$ being back-propagated to maintain its magnitude, even if the loss function only penalizes the output at the end of the sequence. (Sec 7.2)
- Formally, we want $(\nabla_{h(t)} L) \frac{\partial h(t)}{\partial h(t-1)}$ to be as large as $\nabla_{h(t)} L$

$$\rightarrow \Omega = \sum_t \left( \frac{\| (\nabla_{h(t)} L) \frac{\partial h(t)}{\partial h(t-1)} \|}{\| \nabla_{h(t)} L \|} - 1 \right)^2$$

(Pascanu et al, 2013a. $\rightarrow$ deals with Sec 10.7, 10.11, and 10.12)
4. Challenge 2: Memorizing Facts
Needs for explicit memory

- Neural networks excel at storing implicit knowledge, but struggle to memorize facts.
- Graves et al. (2014b) hypothesized that this is because neural networks lack the equivalent of the working memory system.
- Weston et al. (2014) introduced memory networks that include a set of memory cells that can be accessed via an addressing mechanism. Memory networks originally required a supervision signal instructing them how to use their memory cells.
- Graves et al. (2014b) introduced the neural Turing machine, which is able to learn to read from and write arbitrary content to memory cells without explicit supervision about which actions to undertake, and allowed end-to-end training without this supervision signal, thanks the use of a content-based soft attention mechanism (see Bahdanau et al. (2014) and Sec. 12.4.5.1)
The task network learns to “control” the memory, deciding specific addresses to read from and to write to within the memory.
It is difficult to optimize functions that produce exact, integer addresses. To alleviate this problem, NTMs actually read to or write from many memory cells simultaneously.

- To read, NTMs take a weighted average of many cells.
- To write, they take a weighted average of many cells. To write, they modify multiple cells by different amounts.

Memory cells are typically augmented to contain a vector.

- One reason is to offset the cost of accessing a memory cell.
- Another reason is that they allow for content-based addressing. Vector-valued cells allow us to retrieve a complete vector-valued memory if we are able to produce a pattern that matches some but not all of its elements. (e.g., “Retrieve the lyrics of the song that has the chorus ‘We all live in a yellow submarine.’”)
5. Variations of RNNs
Feedback only from o to h (10.2.1)
Feedback only from o to h (10.2.1)

- Cons.
  - Less powerful: can express a smaller set of functions
  - Lack important information from the past: only through o

- Pros.
  - Training can be parallelized: all the time steps are decoupled so the gradient for each time step can be computed in isolation via “teacher forcing”

- Teacher forcing
  - A procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output $y^{(t)}$ as input at time $t + 1$
  - Originally motivated to avoid BPTT
Example 5.1 (Teacher forcing)

Let $t = 2$ and we want to maximize conditional log likelihood.

$$\log p(y^{(1)}, y^{(2)}|x^{(1)}, x^{(2)}) = \log p(y^{(1)}|y^{(2)}, x^{(1)}, x^{(2)}) + \log p(y^{(1)}|x^{(1)}, x^{(2)})$$
Still has one restriction: the length of input and output sequence should be the same.
Bidirectional RNNs (10.3)

- Based on the idea that the output at time $t$ may depend on both the previous and the future elements in the sequence.

(Left) Typical Bidirectional RNNs: two RNNs stacked on top of each other. (Right). Deep Bidirectional RNNs: multiple layers per time step.
Summarize a sequence and produce a fixed-size representation, which can be used as a input for further processing.
The single input vector $x$ can be an extra input at each time step (like below), or the initial state $h^{(0)}$, or both

This RNN is appropriate for tasks such as image captioning.
Can be trained to map an input sequence to output sequence which is not necessarily of the same length.

An encoder processes the input sequence and then emits the context $C$, usually as a simple function of its final hidden state.

A decoder is conditioned on the fixed-length vector $C$ to generate the output sequence.
Only output sequence (10.2.3)

- Fully connected graphical model for a sequence $y^{(1)}, y^{(1)}, ..., y^{(t)}, ...$: every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$)
  - $y^{(t)} = g^{(t)}(y^{(t-1)}, y^{(t-2)}, ..., y^{(1)})$

- Introduce hidden states & share parameters (stationary assumption)
  - $y^{(t)} = g(h^{(t-1)}); h^{(t)} = f(h^{(t-1)}, y^{(t)})$
How to determine the length of the sequence?

- Add a special symbol corresponding to the end of a sequence
- Introduce an extra Bernoulli output to the model that represents the decision to either continue or halt generation at each time step
- Add an extra output to the model that predicts the integer $\tau$ itself
Deep RNNs (10.5)

(a) Several hidden recurrent states. (b) Deeper computation in the input-to-hidden, hidden-to-hidden, and hidden-to-output parts. (c) Skip connections to mitigate path-lengthening effect in (b).
Recursive NNs (10.6)

- Richard Socher's lecture
  https://youtu.be/DJHvaGU9SW8
Recursive NNs

How can we represent the meaning of longer phrases such as “the country of my birth”? → By mapping them into the same vector space!
How should we map phrases into a vector space? → Principle of compositionality!

The meaning (vector) of a sentence is determined by

- the meanings of its words, and
- the rules that combine them
Recursive NNs

- Inputs: two candidate children’s representations
- Outputs:
  - The semantic representation if two nodes are merged.
  - Score of how plausible the new node would be.

One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length $\tau$, the depth (measured as the number of compositions of nonlinear operations) can be drastically reduced from $\tau$ to $O(\log \tau)$.