Introduction to Machine Learning

Lecture 14 - Recurrent Neural Networks (RNNs)

Guang Bing Yang, PhD

Recurrent Neural Network (RNN)

- Introduction to Recurrent Neural Networks
- RNN based Language Model
- GRU and LSTM

What is Recurrent Neural Network

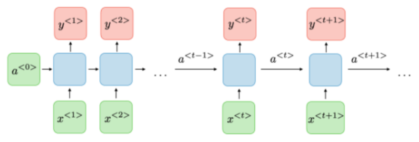
- RNNs are deep neural networks for building models for natural language, audio, and other sequence data
- Applications implementing the concepts of sequence models include speech recognition, music synthesis, chatbots, machine translation, natural language understanding, etc. RNN models perform well on temporal data
- There are several variants including LSTMs (Long Short-Term Memories), GRUs (Gate Recurrent Units), and Bidirectional RNNs
- Sequence Models -- They are a type of models that deal with temporal data--data include sequential information

RNNs

- RNNs have been widely applied in:
  - Speech recognition (sequence to sequence),
  - Music generation (one to sequence),
  - Sentiment classification (sequence to one),
  - DNA sequence analysis (sequence to sequence),
  - Machine translation (sequence to sequence),
  - Video activity recognition (sequence to one),
  - Named entity recognition (sequence to sequence).

RNNs: Architecture

- Overview:
  - Regular neural networks have two problems when dealing with sequence data:
    - Inputs, outputs can be different lengths in different examples
    - Doesn't share features learned across different positions of text/sequence
  - RNN has no such problems



Standard RNN Architecture. Cited from Stanford Deep Learning

RNNs: Architecture

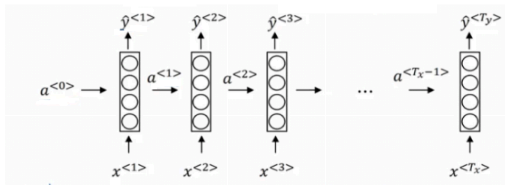
- RNNs are a set of neural networks that use previous outputs as inputs of next neural networks
- $a^{<0>}$ is usually initialized with zeros, but some others may initialize it randomly in some cases.
- For each time step t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:
  - $a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$, and
  - $y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$
  - where, $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are parameters matrices, and g_1, g_2 are activation functions.
    - W_{ax} : (NoOfHiddenNeurons,n_x)
    - W_{aa} : (NoOfHiddenNeurons,NoOfHiddenNeurons) — the memory the RNN is trying to maintain from the previous layers
    - W_{ya} : (n_y, NoOfHiddenNeurons)

RNNs: Example

- The activation function of a is usually tanh or ReLU and for y depends on your task choosing some activation functions like sigmoid and softmax.
- For example, a sentence: 'She said, "Alan Turing was a great scientist"'. In this example Alan is a person's name but we know that from the word **scientist** that came after Alan not from **She** and **said** that was before it.
- So classical RNNs can not learn from elements later in the sequence.
- Bidirectional RNN (BRNN) can address this problem.

RNNs: Architecture

- Forward propagation:
  - $a^{<0>} = 0$, $a^{<1>} = g_1(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a)$, $g_1 \rightarrow$ tanh
  - $\hat{y}^{<1>} = g_2(W_{ya}a^{<1>} + b_y)$, $g_2 \rightarrow$ sigmoid
  - $a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$, $\hat{y}^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$

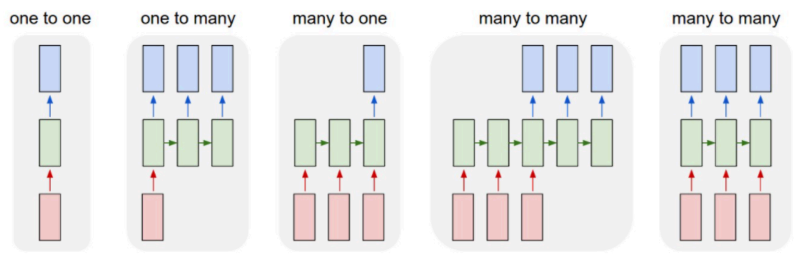


RNNs: Architecture

- The loss function L of all time steps is defined based on the loss at every time step as follows:
- $\mathscr{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathscr{L}(\hat{y}^{<t>}, y^{<t>})$
- Backpropagation through time
  - Usually deep learning frameworks do backpropagation automatically for you.
  - It is done at each point in time. At time step T, the derivative of the loss $\mathscr{L}$ with respect to weight matrix W is expressed as follows:
- $\frac{\partial \mathscr{L}^{(T)}}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathscr{L}^{(T)}}{\partial W}\bigg|_{(t)}$

RNNs: Types

- Different types of RNNs:



Types of RNNs: Cited from Andrej Karpathy blog

RNNs: Types

- Many-to-Many RNNs are used in Named Entity Recognition, etc.
- A One-to-Many architecture application would be music generation.
- A Many-to-One architecture application would be sentiment analysis
- An alternative Many-to-Many architecture that fits the translation would be as follows: There are an encoder and a decoder parts in this architecture. The encoder encodes the input sequence into one matrix and feed it to the decoder to generate the outputs. Encoder and decoder have different weight matrices.

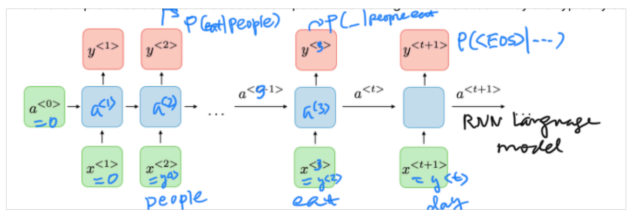
RNNs: Language model and sequence generation

- RNNs do very well in language model problems
- What is a language model?
  - It is a probability distribution for sentences and the application decides the best based on this probability.
  - For example, a speech recognition problem and someone says a sentence that can be interpreted into to two sentences:
    - The apple and **pair** salad
    - The apple and **pear** salad
  - **Pair** and **pear** sounds exactly the same, so how would a speech recognition application choose from the two.
  - To find the correct term, where the language model comes in. It gives a probability for the two sentences and the application decides the best based on this probability.

RNNs: Language model and sequence generation

- How to build language models with RNNs?
- The first thing is to get a training set: a large corpus of target language text.
- Then tokenize this training set by getting the vocabulary and then one-hot each word.
- Put an end of sentence token <EOS> with the vocabulary and include it with each converted sentence. Also, use the token <UNK> for the unknown words.
- For example, Given the sentence “People eat 3 meals per day. <EOS>”

RNNs: Language model and sequence generation



RNN Language Model.

- The loss function: $\mathcal{L}(\hat{y}, y) = \sum_{i=1}^{T_y} y_i^{<t>} \log \hat{y}_i^{<t>}$, $\mathcal{L} = \sum \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$

RNNs: Language model and sequence generation

- To use this model:
  - For predicting the chance of the next word, we feed the sentence to the RNN and then get the final $y^{<t>}$ hot vector and sort it by maximum probability.
  - For taking the probability of a sentence, compute this:
    - $p(y^{<1>}, y^{<2>}, y^{<3>}) = p(y^{<1>}) * p(y^{<2>} \mid y^{<1>}) * p(y^{<3>} \mid y^{<1>}, y^{<2>})$,
    - thesis simply feeding the sentence into the RNN and multiplying the probabilities.

RNNs: Sampling new sequences

- To use this model to sample new sequences:
  1. first pass $a^{<0>} = 0$ vector, and $a^{<1>} = 0$ vector
  2. randomly choose a prediction from distribution obtained by $\hat{y}^{<1>}$. For example it could be "The"
  3. pass the last predicted word with the calculated $a^{<1>}$
  4. keep doing 3 & 4 steps for a fixed length or until get the <EOS> token.
  5. reject any <UNK> token

RNNs: Sampling new sequences

- Just discussed how to build a word-level language model.
- It's also possible to implement a character-level language model.
- In the character-level language model, the vocabulary will contain [a-zA-Z0-9], punctuation, special characters and possibly token.
- Character-level language model has some pros and cons compared to the word-level language model
- Pros:
  - There will be no <UNK> token - it can create any word.
- Cons:
  - The main disadvantage is that you end up with much longer sequences.
- Character-level language models are not as good as word-level language models at capturing long range dependencies between how the the earlier parts of the sentence also affect the later part of the sentence.
- Also more computationally expensive and harder to train.

RNNs: Vanishing gradients

- One of the problems with naive RNNs that they run into vanishing gradient problem.
- An RNN that process a sequence data with the size of 10,000 time steps, has 10,000 deep layers which is very hard to optimize.
- This is the reason why they happen because it is difficult to capture long term dependencies due to multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers.

RNNs: Vanishing gradients

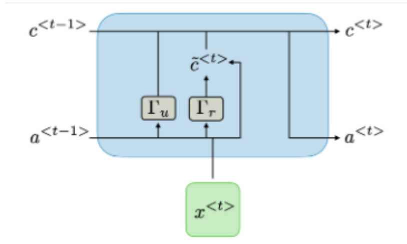
- For example, there is a language modeling problem and there are two sequences that model tries to learn:
  - "The cat, which already ate …, was full"
  - "The cats, which already ate …, were full"
  - Dots represent many words in between.
- What need to learn here that "was" came with "cat" and that "were" came with "cats". The naive RNN is not very good at capturing very long-term dependencies like this.
- For computing the word "was", the model needs to compute the gradient for everything behind. Multiplying fractions tends to vanish the gradient, while multiplication of large number tends to explode it.
- In the problem it is hard for the network to memorize "was" word all over back to "cat". So in this case, the network won't identify the singular/plural words so that it gives it the right grammar form of verb was/were.
- The conclusion is that RNNs aren't good in long-term dependencies.

RNNs: Solutions to Vanishing gradients

- Solution for the Vanishing gradient problem:
  - Weight initialization.
  - Echo state networks.
  - Use LSTM/GRU networks.

RNNs: Gated Recurrent Unit (GRU)

- GRU is an RNN type that can help solve the vanishing gradient problem and can remember the long-term dependencies.
- The basic RNN unit can be visualized to be like this:

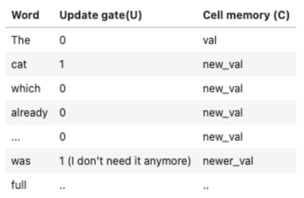


GRU Architecture. Cited from Stanford Deep Learning

- Each layer in GRUs has a new variable C which is the memory cell. It can tell to whether memorize something or not.
- In GRUs, $a^{<t>} = C^{<t>}$
- Equations of the GRUs:
  - $\tilde{C}^{<t>} = \tanh(W_c[\Gamma_r * a^{<t-1>}, x^{<t>}] + b_c)$,
  - $C^{<t>} = \Gamma_u * \tilde{C}^{<t>} + (1 - \Gamma_u) * C^{<t-1>}$
  - $a^{<t>} = C^{<t>}$

RNNs: Gated Recurrent Unit (GRU)

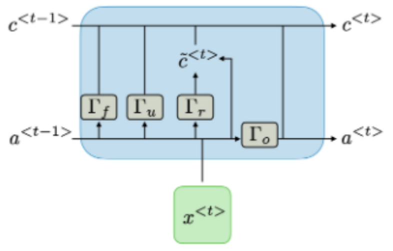
- The update gate is between 0 and 1
- To understand GRUs imagine that the update gate is either 0 or 1 most of the time.
- So update the memory cell based on the update cell and the previous cell.
- For the previous example,
  - suppose that U is 0 or 1 and is a bit that tells us if a singular word needs to be memorized.
  - Splitting the words and get values of C and U at each place:
- Because the update gate U is usually a small number like 0.00001, GRUs doesn't suffer the vanishing gradient problem.



Word	Update gate(U)	Cell memory (C)
The	0	val
cat	1	new_val
which	0	new_val
already	0	new_val
...	0	new_val
was	1 (I don't need it anymore)	newer_val
full	..	..

RNNs: Long Short Term Memory (LSTM)

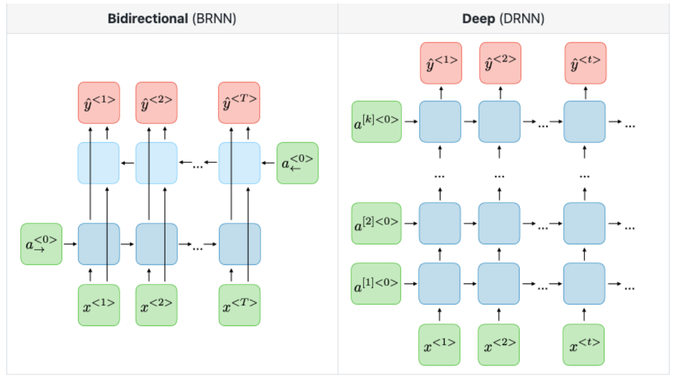
- LSTM - the other type of RNN that can enable you to account for long-term dependencies. It's more powerful and general than GRU.
- In LSTM , $a^{<t>} \neq C^{<t>}$
- Here are the equations of an LSTM unit:



LSTM Architecture. Cited from Stanford Deep Learning

- In LSTM we have an **update gate U** (sometimes it's called **input gate I**, a **forget gate F,** an **output gate O**, and a candidate cell variables $\tilde{C}^{<t>}$
- Equations of the LSTM:
  - $\tilde{C}^{<t>} = \tanh(W_c[\Gamma_r * a^{<t-1>}, x^{<t>}] + b_c)$,
  - $C^{<t>} = \Gamma_u * \tilde{C}^{<t>} + \Gamma_f * C^{<t-1>}$
  - $a^{<t>} = \Gamma_o * C^{<t>}$

RNNs: Variants of RNNs



Variant RNN Architecture. Cited from Stanford Deep Learning

Recap

- RNNs are deep neural networks for building models for natural language, audio, and other sequence data
- RNNs are for sequence modelling problem. RNN models perform well on temporal data
- There are several variants including LSTMs (Long Short-Term Memories), GRUs (Gate Recurrent Units), and Bidirectional RNNs
- RNNs have been widely applied in Speech recognition, Sentiment classification, DNA sequence analysis, Machine translation, etc.
- There are many different types of RNNs: one-to-one, one-to-many, many-to-many, many-to-one.
- RNN based language model is widely used in NLP and NLU related problems
- Standard RNNs have gradient vanishing problem
- GRUs and LSTMs solve this problem

Questions?