Some applications of RNNs

CEng 783 – Deep Learning
Fall 2019

Emre Akbaş
Reminders

• Project progress demos on Dec 10 (two weeks from now)
  – Demo/presentation in class
  – 5 minutes demo/presentation. Then, a few minutes of feedback.
Today

- RNN + LSTM recap
- Applications of RNN
  - Character-level language modeling
    - Word embeddings
  - Image captioning
  - Machine translation (+ attention mechanism)
    - A hands-on example on seq2seq translation.
  - Image generation
  - External memory models (Neural Turing Machines)
- (If there is time) Deep generative models
RNN recap
\[ h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \]

\( h \): hidden state \hspace{0.5cm} \( t \): time

\( \theta \): (shared) parameters \hspace{0.5cm} x: input

The network maps the whole input \( x^{(1)}, x^{(2)}, \ldots, x^{(t)} \) to \( h^{(t)} \). \hspace{0.5cm} e.g.,

\[ h^{(3)} = f(f(h^{(0)}, x^{(1)}; \theta), x^{(2)}; \theta), x^{(3)}; \theta) \]
Folded representation and unfolding

\[ h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \]

Two different ways of drawing above equation:

This means a single time-step

[Fig. 10.2 from Goodfellow et al. (2016)]
A variety of architectures are possible
A recurrent network that maps input sequence $x$ to output sequence $o$, using a loss function $L$ and label sequence $y$.
A recurrent network that maps input sequence $x$ to output sequence $o$, using a loss function $L$ and label sequence $y$.

Update equations:

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

Note: $\tanh(\cdot)$, $\text{softmax}(\cdot)$ are just example choices.
Bi-directional RNNs

Output at t depends on both the past and the future

Hidden states
Ways of adding depth
The challenge of long-term dependencies

- More depth → more “vanishing or exploding gradient” problem
- Why?
- Consider repeated matrix multiplication:

\[ h^{(t)} = W^\top h^{(t-1)} \]
\[ h^{(t)} = (W^t)^\top h^{(0)} \]

\[ W = Q\Lambda Q^\top \rightarrow h^{(t)} = Q^\top \Lambda^t Q h^{(0)} \]

Values here will either vanish or explode!
Solution to exploding gradients: gradient clipping

Clip the magnitude.

Algorithm 1 Pseudo-code for norm clipping

\[
\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \mathbf{\theta}}
\]

if \( \|\hat{\mathbf{g}}\| \geq \text{threshold} \) then

\[
\hat{\mathbf{g}} \leftarrow \frac{\text{threshold}}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}
\]

end if

Error surface for a single hidden unit RNN. Solid lines depict trajectories of the regular gradient, dashed lines clipped gradient.

[From Figure 6 in Pascanu et al. (2013)]
Solution to vanishing gradients: regularize the gradient

\[ \Omega = \sum_k \Omega_k = \sum_k \left( \left\| \frac{\partial \mathcal{E}}{\partial x_{k+1}} \frac{\partial x_{k+1}}{\partial x_k} \right\| - 1 \right)^2 \]

The regularizer prefers solutions for which the error preserves norm as it travels back in time.

\( x \) here refers to the state of the RNN.

[Pascanu et al. (2013), “On the difficulty of training Recurrent Neural Networks”]
Another solution to vanishing gradients is LSTM.

[Hochreiter & Schmidhuber (1997)]

LSTM (Long short-term memory) recap
Long short-term memory (LSTM)

A repeating module in a LSTM:

[Figures from C. Olah’s blog post.]

[Neural Network Layer] [Pointwise Operation] [Vector Transfer] [Concatenate] [Copy]
Long short-term memory (LSTM)

A repeating module in a LSTM:

\[ f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f) \]
\[ i_t = \sigma (W_i \cdot [h_{t-1}, x_t] + b_i) \]
\[ \tilde{C}_t = \tanh (W_C \cdot [h_{t-1}, x_t] + b_C) \]
\[ C_t = f_t \ast C_{t-1} + i_t \ast \tilde{C}_t \]
\[ o_t = \sigma (W_o \cdot [h_{t-1}, x_t] + b_o) \]
\[ h_t = o_t \ast \tanh (C_t) \]

[Explanation of these equations on board]
Long short-term memory (LSTM)

Summary

The key idea behind LSTM: cells can implement the identity transform. i.e. $C_t = C_{t-1}$ is possible with appropriate gate values.

\[
f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)
\]
\[
i_t = \sigma (W_i \cdot [h_{t-1}, x_t] + b_i)
\]
\[
\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)
\]
\[
C_t = f_t * C_{t-1} + i_t * \tilde{C}_t
\]
\[
o_t = \sigma (W_o \ [h_{t-1}, x_t] + b_o)
\]
\[
h_t = o_t * \tanh (C_t)
\]
GRU: Another “gated” RNN variant

Gated Recurrent Unit (GRU)  Cho et al. (2014)

\[ z_t = \sigma (W_z \cdot [h_{t-1}, x_t]) \]
\[ r_t = \sigma (W_r \cdot [h_{t-1}, x_t]) \]
\[ \tilde{h}_t = \tanh (W \cdot [r_t \ast h_{t-1}, x_t]) \]
\[ h_t = (1 - z_t) \ast h_{t-1} + z_t \ast \tilde{h}_t \]

Combines the forget and input gates into a single “update gate.” Merges the cell state and hidden state (no \( C_t \)), and makes some other changes. The resulting model is simpler than LSTM.
Applications of RNNs
Character-level language modeling

The problem: Given a sequence (a partial sentence or word), predict the next character. e.g.:

The sky is b?
The sky is bl?
The sky is blu?

A very educative blog post on this problem by A. Karpathy
Go deeper:

```python
rnn = RNN()
y = rnn.step(x)  # x is an input vector, y is the RNN's output vector

class RNN:
    # ...
    def step(self, x):
        # update the hidden state
        self.h = np.tanh(np.dot(self.W hh, self.h) + np.dot(self.W xh, x))
        # compute the output vector
        y = np.dot(self.W hy, self.h)
        return y

y1 = rnn1.step(x)
y = rnn2.step(y1)
```

[Material from Karpathy's blog post.]
- One-hot encoding of characters.
- Importance of state: hel? vs hell?
- Learnable parameters of the network.

[Material from Karpathy’s blog post.]
• Training
• Testing

Go to the post to see examples.

[Material from Karpathy’s blog post.]
(Not an RNN application but) Closely-related, important topic: word embeddings.
- e.g. Mikolov et al.’s word2vec (2013).

Given a word, output a fixed length vector (i.e. embed the word in a finite-dimensional space).

Tends to map semantically similar words to nearby points in the vector space.
Figure: The Skip-gram model architecture. The training objective is to learn word vector representations that are good at predicting the nearby words. [Mikolov et al. 2013]
word2vec

Basically, form (<context>,<word>) pairs and train a NN on these pairs.

• One method is called skip-gram: predict the context word from the target word.

e.g. for the sentence “the quick brown fox jumped over the lazy dog”, context (N=1) word pairs are:

([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...
They used a shallow network:

Figure 1: A simple CBOW model with only one word in the context

[Figure from Rong, 2014]

[CBOB: Continuous Bag-of-words]
Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.
Figure 2: Left panel shows vector offsets for three word pairs illustrating the gender relation. Right panel shows a different projection, and the singular/plural relation for two words. In high-dimensional space, multiple relations can be embedded for a single word.

[Figure from Mikolov, 2013b]
Word2vec results

Surprisingly, simple algebraic operations work on these vectors:
- “biggest – big + small” → “smallest”
- “france – paris + germany” → “berlin”

Table 8: Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality).

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>France - Paris</td>
<td>Italy: Rome</td>
<td>Japan: Tokyo</td>
<td>Florida: Tallahassee</td>
</tr>
<tr>
<td>big - bigger</td>
<td>small: larger</td>
<td>cold: colder</td>
<td>quick: quicker</td>
</tr>
<tr>
<td>Miami - Florida</td>
<td>Baltimore: Maryland</td>
<td>Dallas: Texas</td>
<td>Kona: Hawaii</td>
</tr>
<tr>
<td>Einstein - scientist</td>
<td>Messi: midfielder</td>
<td>Mozart: violinist</td>
<td>Picasso: painter</td>
</tr>
<tr>
<td>Sarkozy - France</td>
<td>Berlusconi: Italy</td>
<td>Merkel: Germany</td>
<td>Koizumi: Japan</td>
</tr>
<tr>
<td>copper - Cu</td>
<td>zinc: Zn</td>
<td>gold: Au</td>
<td>uranium: plutonium</td>
</tr>
<tr>
<td>Berlusconi - Silvio</td>
<td>Sarkozy: Nicolas</td>
<td>Putin: Medvedev</td>
<td>Obama: Barack</td>
</tr>
<tr>
<td>Microsoft - Windows</td>
<td>Google: Android</td>
<td>IBM: Medvedev</td>
<td>Apple: iPhone</td>
</tr>
<tr>
<td>Microsoft - Ballmer</td>
<td>Google: Yahoo</td>
<td>IBM: Linux</td>
<td>Apple: Jobs</td>
</tr>
<tr>
<td>Japan - sushi</td>
<td>Germany: bratwurst</td>
<td>IBM: McNealy</td>
<td>USA: pizza</td>
</tr>
</tbody>
</table>
Image captioning

Images from NeuralTalk Demo

Demo video

[Karpathy and Fei-Fei (2015)]
[Vinyals et al. (2015)]

- a street sign on a pole in front of a building
- a plate with a sandwich and a salad
- an elephant standing in a grassy field with trees in the background
- a man is throwing a frisbee in a park
Image captioning

Recurrent Neural Network

Convolutional Neural Network

Fei-Fei Li & Andrej Karpathy & Justin Johnson

Lecture 10 - 52 8 Feb 2016

Image captioning

before:
\[ h = \tanh(W_{xh} \ast x + W_{hh} \ast h) \]

now:
\[ h = \tanh(W_{xh} \ast x + W_{hh} \ast h + W_{ih} \ast v) \]

Image captioning

Image captioning

Machine Translation

Composed of two RNNs

Can map an arbitrary length input sequence to an arbitrary length output sequence (notice $n_x$ and $n_y$). e.g. machine translation, speech recognition.

[Cho et al. (2014)]
[Sutskever et al. (2014)]

[Fig. 10.12 from Goodfellow et al. (2016)]
Machine Translation

Steps
1) Encoder or reader RNN reads processes the input sequence.
2) Encoder emits the learned context $C$ (a simple function of its learned hidden states).
3) Decoder or writer RNN which is conditioned on $C$, produces the output sequence.

Input sentence

Output sentence

[Fig. 10.12 from Goodfellow et al. (2016)]
The two RNNs are trained jointly to maximize the average
\[ P(y^{(1)}, \ldots, y^{(n_y)} | x^{(1)}, \ldots, x^{(n_x)}) \]
over all (x,y) pairs in the training set.

Typically, \[ C = h_{n_x} \]
Google’s neural machine translation

Basic encoder-decoder architecture:

- Encoder
- Embed
- Decoder

$He \ loved \ to \ eat$.

$Er \ liebte \ zu \ essen$.

NULL $Er \ liebte \ zu \ essen$

[Figure by S. Merity]

[Wu et al. 2016]
Google’s neural machine translation

Basic encoder-decoder architecture:

S is a lossy summary of the input sentence → performance goes down with longer sentences

26.11.2019
Google’s neural machine translation

[Wu et al. 2016]

Add **attention mechanism**.

![Diagram of neural machine translation model with example input: He loved to eat.](figure)

[Figure by S. Merity]
Google’s neural machine translation

Add **attention mechanism**.

Encoder output for \( t \)th input word

DecoderRNN output for \( (i-1) \)th output word

EncoderRNN output for \( t \)th input word

AttentionFunction is a one hidden-layer feedforward NN.

\[
\begin{align*}
    s_t &= \text{AttentionFunction}(y_{i-1}, x_t) \quad \forall t, \quad 1 \leq t \leq M \\
    p_t &= \frac{\exp(s_t)}{\sum_{t=1}^{M} \exp(s_t)} \quad \forall t, \quad 1 \leq t \leq M \\
    a_i &= \sum_{t=1}^{M} p_t \cdot x_t
\end{align*}
\]
Google’s neural machine translation

[Wu et al. 2016]

Add **attention mechanism.**

EncoderRNN output for \( t \)th input word

DecoderRNN output for \((i-1)^{th}\) output word

AttentionFunction is a one hidden-layer feedforward NN.

**Problem:** \( x_t \) encodes only causal information.

\[
\begin{align*}
s_t &= \text{AttentionFunction}(y_{i-1}, x_t) & \forall t, \; 1 \leq t \leq M \\
p_t &= \frac{\exp(s_t)}{\sum_{t=1}^{M} \exp(s_t)} & \forall t, \; 1 \leq t \leq M \\
a_i &= \sum_{t=1}^{M} p_t \cdot x_t
\end{align*}
\]
Google’s neural machine translation

[ Wu et al. 2016 ]

Add another RNN to make the encoder bi-directional.

[Figure by S. Merity]
Google’s neural machine translation

The whole system:

![Figure by S. Merity](Wu et al. 2016)
Use of “attention” in image captioning:

1. Input Image
2. Convolutional Feature Extraction
3. RNN with attention over the image
4. Word by word generation

Another seq-to-seq example

Let us try to solve simple arithmetic operations using seq-to-seq translation model.

e.g., an example input would be “3+5” and the output will be “=8.” Both of them are strings.

Jupyter notebook (Keras code)
PixelRNN [Oord et al. 2016]

Models image generation as a sequential prediction process.

It can complete partially occluded images:

Figure 1. Image completions sampled from a PixelRNN.
PixelRNN  [Oord et al. 2016]

It can also generate images from scratch.

*Figure 7.* Samples from models trained on CIFAR-10 (left) and ImageNet 32x32 (right) images. In general we can see that the models capture local spatial dependencies relatively well. The ImageNet model seems to be better at capturing more global structures than the CIFAR-10 model. The ImageNet model was larger and trained on much more data, which explains the qualitative difference in samples.
PixelRNN [Oord et al. 2016]

Probability of an image $x$:

$$p(x) = \prod_{i=1}^{n^2} p(x_i | x_1, \ldots, x_{i-1})$$

- Pixel(i,j) is generated by Hidden(i,j).
- Hidden(i,j) depends on Hidden(i,j-1), Hidden(i-1,j)
- However, training takes a lot of time (b/c it has to be sequential)
- The authors propose a few parallelization strategies (on the right)

Figure 4. Visualization of the input-to-state and state-to-state mappings for the three proposed architectures.
Neural Turing Machines

The RNNs we have seen so far had limited “memory” (What limits this?)

If we plugin an external memory to an RNN, we obtain a neural Turing machine.
Neural Turing Machines

[Graves et al. 2014]

Crucially, every component of the architecture is differentiable, making it straightforward to train with gradient descent.
Neural Turing Machines [Graves et al. 2014]

- “Blurry” read and write operations

3.1 Reading

Let $M_t$ be the contents of the $N \times M$ memory matrix at time $t$, where $N$ is the number of memory locations, and $M$ is the vector size at each location. Let $w_t$ be a vector of weightings over the $N$ locations emitted by a read head at time $t$. Since all weightings are normalised, the $N$ elements $w_t(i)$ of $w_t$ obey the following constraints:

$$\sum_i w_t(i) = 1, \quad 0 \leq w_t(i) \leq 1, \forall i.$$ (1)

The length $M$ read vector $r_t$ returned by the head is defined as a convex combination of the row-vectors $M_t(i)$ in memory:

$$r_t \leftarrow \sum_i w_t(i)M_t(i),$$ (2)

which is clearly differentiable with respect to both the memory and the weighting.
Neural Turing Machines [Graves et al. 2014]

3.2 Writing

Taking inspiration from the input and forget gates in LSTM, we decompose each write into two parts: an erase followed by an add.

Given a weighting $w_t$ emitted by a write head at time $t$, along with an erase vector $e_t$ whose $M$ elements all lie in the range $(0, 1)$, the memory vectors $M_{t-1}(i)$ from the previous time-step are modified as follows:

$$\tilde{M}_t(i) \leftarrow M_{t-1}(i) \left[ 1 - w_t(i)e_t \right],$$

where $1$ is a row-vector of all 1-s, and the multiplication against the memory location acts point-wise. Therefore, the elements of a memory location are reset to zero only if both the weighting at the location and the erase element are one; if either the weighting or the erase is zero, the memory is left unchanged. When multiple write heads are present, the erasures can be performed in any order, as multiplication is commutative.

Each write head also produces a length $M$ add vector $a_t$, which is added to the memory after the erase step has been performed:

$$M_t(i) \leftarrow \tilde{M}_t(i) + w_t(i)a_t.$$
Content addressing

For content-addressing, each head (whether employed for reading or writing) first produces a length $M$ key vector $k_t$ that is compared to each vector $M_t(i)$ by a similarity measure $K[\cdot,\cdot]$. The content-based system produces a normalised weighting $w_t^c$ based on the similarity and a positive key strength, $\beta_t$, which can amplify or attenuate the precision of the focus:

$$w_t^c(i) \leftarrow \frac{\exp \left( \beta_t K[k_t, M_t(i)] \right)}{\sum_j \exp \left( \beta_t K[k_t, M_t(j)] \right)}.$$  \hspace{1cm} (5)

In our current implementation, the similarity measure is cosine similarity:

$$K[u, v] = \frac{u \cdot v}{\|u\| \cdot \|v\|}.$$  \hspace{1cm} (6)
Figure 2: Flow Diagram of the Addressing Mechanism. The key vector, $k_t$, and key strength, $\beta_t$, are used to perform content-based addressing of the memory matrix, $M_t$. The resulting content-based weighting is interpolated with the weighting from the previous time step based on the value of the interpolation gate, $g_t$. The shift weighting, $s_t$, determines whether and by how much the weighting is rotated. Finally, depending on $\gamma_t$, the weighting is sharpened and used for memory access.
Neural Turing Machines [Graves et al. 2014]

- They showed that NTM is performing much better than LSTMs in different tasks:
  - Copy,
  - Repeated copy,
  - Associative recall,
  - Sorting.
References

References