Neural Networks
Lecture Notes
Recurrent neural networks

Learning Goals

After studying the lecture material you should:

1. be able to explain how an RNN differs from a feedforward neural network
2. be able to write down the update equation for the hidden unit states in an RNN
3. be able to explain in your own words how BPTT works (no need to remember all the equations)
4. understand the vanishing gradient issue in RNNs
5. understand how word embeddings work
6. be able to explain in your own words why LSTMs are useful
7. understand the gist of what neural Turing machines are and why they are important

Notes

Feedforward versus recurrent neural networks

Recurrent neural networks (RNNs) can be contrasted with feedforward neural networks.

In feedforward neural networks activation is fed forward from input to output through hidden layers. Mathematically, they implement static input-output mappings (functions). A basic theoretical result [Cyb89] is that MLPs can approximate arbitrary functions with arbitrary precision. MLPs are trained using backpropagation and have proven useful in many practical applications as approximators of nonlinear functions and as pattern classifiers

In contrast, recurrent neural networks are implementations of dynamical systems. They come in two variants:

- *Autonomous RNNs* with converging dynamics and fixed inputs (Hopfield networks; Boltzmann machines)
- *Non-autonomous* RNNs that have time-varying inputs

A basic theoretical result ([SS91]) is that RNNs can approximate arbitrary dynamical systems with arbitrary precision. They are more difficult to train than MLPs and different approaches have been developed:

- Real-time recurrent learning
- Neuroevolution

- Backpropagation through time
- Reservoir computing

We will only address the latter two cases.

Recurrent neural networks can be used for various tasks:

- *Sequence output* (e.g. image captioning takes an image and outputs a sentence of words)
- *Sequence input* (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment)
- *Sequence input and sequence output* (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French)
- *Synced sequence input and output* (e.g. video classification where we wish to label each frame of the video)

Language modeling

RNNs can also be used for language modeling. That is, to generate meaningful text after training the RNN on a lot of texts. The question becomes what the best input representation is. We have various options here.

The first option is to use one-hot input vectors $(0, 0, 0, 1, 0, 0, \ldots)$ such that the word with the assigned number i has a 1 at the ith position. This requires N vector elements (input neurons) for a language consisting of N possible words. This can be prohibitive to use with RNNs that are already difficult to train in the first place.

A second option is to use character-level language models such that we have as many vector elements as characters in a language. This option is used often but it requires the RNN to learn to generate proper words as well as proper sentences.

A third option is to use a word embedding. This relies on the notion of distributional semantics which states that one can know the meaning of a word by the company that it keeps. Each word is represented as a continuous vector representation using a word embedding. Here, we focus on the skip-gram model ([MCCD13], [MSC$^+$13]). Given a sequence of words $w_1, w_2, \ldots, w_T$, the *skip-gram model* maximizes the following objective function:

$$J = \frac{1}{T} \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} \mid w_t) \,.$$

i.e. the aim is to predict the context (surrounding) words given a target word. This probability can be modeled using a neural network with one hidden layer. The input-to-hidden weights are given by $\mathbf{U}$ and the hidden-to-output weights are given by $\mathbf{V}$. The probability of a context word w_s given a target word w_t is then expressed as

$$p(w_s \mid w_t) = \frac{\exp\left(\mathbf{v}_{w_s}^T \mathbf{u}_{w_t}\right)}{\sum_{w=1}^{W} \exp\left(\mathbf{v}_w^T \mathbf{u}_{w_t}\right)}$$

where $\mathbf{u}_w$ and $\mathbf{v}_w$ are the input and output vectors associated with word w and W is the number of words in the vocabulary. The corresponding neural network uses a linear activation function for the hidden units and a softmax activation function for the output units. Let $\mathbf{e}(w)$ be the one-hot encoding of a word (e.g. $[0, 0, 0, 1, 0, 0, \ldots, 0, 0]$). Then, the word embedding of w is given by $\text{vec}(w) = \mathbf{e}(w)^T \mathbf{U} = \mathbf{u}_w = \mathbf{h}$.

Backpropagation through time

Consider a (non-autonomous) recurrent neural network with inputs $\mathbf{x}(n) = (x_1(n), \ldots, x_I(n))^T$, hidden units $\mathbf{h}(n) = (h_1(n), \ldots, h_H(n))^T$ and outputs $\mathbf{y}(n) = (y_1(n), \ldots, y_K(n))^T$, where n denotes the nth time point (layer). Let $\mathbf{W}^i$ denote the input to hidden weights, $\mathbf{W}^h$ the hidden to hidden weights and $\mathbf{W}^o$ the hidden to output weights. We use $\mathbf{f}$ and $\mathbf{g}$ to denote the activation functions. In an RNN, updating of the hidden layers is given by

$$\mathbf{h}(n) = \mathbf{f}(\mathbf{W}^i\mathbf{x}(n) + \mathbf{W}^h\mathbf{h}(n-1))$$

and updating of the output units is given by

$$\mathbf{y}(n) = \mathbf{g}(\mathbf{W}^o\mathbf{h}(n))$$

A popular learning algorithm for recurrent neural networks is backpropagation through time (BPTT). It generalizes backprop for feedforward networks to the recurrent case. This is done by unrolling the network so all cycles between units are removed and where the weights at each time point are identical. The teacher data consists of a single input and output time series. At time point (layer!) n we have target values

$$\mathbf{t}(n) \quad = \quad (t_1(n), \ldots, t_K(n))^T$$

The error to minimise is given by:

$$E = \sum_{n=1}^{T} ||\mathbf{t}(n) - \mathbf{y}(n)||^2 = \sum_{n=1}^{T} E^n$$

This is the same as in standard backpropagation but n now iterates over time rather than training examples.

BPTT uses the following forward pass:

- Take a training example and push it through the unrolled network
- At each n-th layer:
  - read the input $\mathbf{x}(n)$
  - compute $\mathbf{h}(n)$ from $\mathbf{x}(n)$ and $\mathbf{h}(n-1)$
  - compute $\mathbf{y}(n)$

After the forward pass, computation of error terms proceeds as follows. The error terms for the output units at time n are given by:

$$\delta_j(n) = \frac{\partial E}{\partial a_j(n)}$$

The error terms for the hidden units at time n are given by:

$$\delta_h(n) = f'(a_h(n)) \left(\sum_{k=1}^{K} \delta_k(n) w^o_{kh} + \sum_{l=1}^{H} \delta_l(n+1) w^h_{lh} \right)$$

The deltas are computed by starting at $n = T$ and recursively applying this equation until $n = 1$ (the deltas at $n = T + 1$ are defined to be 0). The weight updates for the RNN are then given by

$$\begin{aligned}
\Delta w^i_{ij} &\propto \sum_{n=1}^{T} \delta_i(n) x_j(n) \\
\Delta w^h_{ij} &\propto \sum_{n=1}^{T} \delta_i(n) h_j(n-1) \;\; \text{where } h_j(n-1) = 0 \text{ for } n = 1 \\
\Delta w^o_{ij} &\propto \sum_{n=1}^{T} \delta_i(n) h_j(n)
\end{aligned}$$

Note that unlike feedforward backpropagation, BPTT is not guaranteed to converge to a local error minimum. This difficulty cannot arise with feedforward networks, because they realize functions, not dynamical systems.

Long short-term memory

RNNs can be seen as infinitely deep neural networks with the difference that each layer gets its own input and output. This means that the *vanishing gradient problem* can have a major influence. That is, it is hard for RNNs to model long-range dependencies that are too far in the past.

Long Short-term Memory (LSTM, [HS97]) is an RNN architecture designed to have a better memory. It uses linear memory cells surrounded by multiplicative gate units to store read, write and reset information. These gates, instead of sending their activities as inputs to other neurons, set the weights on edges connecting the rest of the neural net to the memory cell. LSTMs can be trained using backpropagation using somewhat more involved gradients.

An LSTM consists of memory cells that contain a *cell state*. The LSTM has the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through", while a value of one means "let everything through!". A memory cell has the following gates (we use subscript t to denote the time index):

- *Forget gate* which decides what information to forget and whose output is given by

$$f_t = \sigma(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + b_f)$$

- *Input gate* which decides what information to store and whose output is given by

$$\begin{aligned} i_t &= \sigma(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + b_i) \\ \tilde{C}_t &= \tanh(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + b_C) \end{aligned}$$

  such that the cell state is updated to

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

- *Output gate* which decides what information to pass on and whose output is given by

$$\begin{aligned} o_t &= \sigma(W_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + b_o) \\ h_t &= o_t \cdot \tanh(C_t) \end{aligned}$$

Note that gated recurrent units (GRUs) are similar to LSTMs but have fewer free parameters.

Neural Turing Machines

RNNs are Turing-Complete ([**?**]), and therefore have the capacity to simulate arbitrary procedures, if properly wired. A Neural Turing Machine ([**?**], NTM) is a differentiable computer that can be trained by gradient descent, yielding a practical mechanism for learning programs. An NTM consists of two basic components:

- a neural network controller (typically an LSTM RNN)
- a memory bank

An NTM is a fully differentiable model by defining blurry read-write operations. NTMs allow *variable binding* and dealing with *variable-length structures*. The absence of these properties was previously used as a counterargument for neural networks as models of human cognition ([**?**]).

The read and write operations are specified as follows: Let $\mathbf{M}_t$ be the contents of the $N \times M$ memory matrix at time t with N the number of memory locations and M the vector size at each location. Let $\mathbf{w}_t$ be a vector of weightings over the N locations emitted by a read head at time t. The length M read vector $\mathbf{r}_t$ returned by the read head is defined as a convex combination of the row-vectors $\mathbf{M}_t(i)$ in memory:

$$\mathbf{r}_t \leftarrow \sum_i w_t(i)\mathbf{M}_t(i)$$

Writing is defined as an erase followed by an add. Given a weighting $\mathbf{w}_t$ emitted by a write head at time t, along with an erase vector $\mathbf{e}_t$ whose M elements all lie in the range (0, 1), the memory vectors $\mathbf{M}_{t-1}(i)$ from the previous time-step are modified as follows:

$$\tilde{\mathbf{M}}_t(i) \leftarrow \mathbf{M}_{t-1}(i)\left[\mathbf{1} - w_t(i)\mathbf{e}_t\right]$$

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

$$\mathbf{M}_t(i) \leftarrow \tilde{\mathbf{M}}_t(i) + w_t(i)\mathbf{a}_t$$

Reading and writing depends on weightings $\mathbf{w}_t$. Weightings are determined by the previous weightings, the state of the memory bank, and controller outputs. This implements a content-based addressing system which is the mechanism for learning programs.

Reading material

For an overview of recurrent neural networks, please check:
http://minds.jacobs-university.de/sites/default/files/uploads/papers/ESNTutorialRev.pdf

For a good intro on BPTT and LSTM, please check Alex Graves' PhD thesis:
Graves, A. (2008). Supervised Sequence Labelling with Recurrent Neural Networks. Image Rochester NY. doi:10.1007/978-3-642-24797-2

Another useful blog post is: http://r2rt.com/recurrent-neural-networks-in-tensorflow-i.html

References

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control. Signals, Syst.*, 2:303–314, 1989.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv Prepr. arXiv1301.3781*, 2013.

[MSC$^+$13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Adv. Neural Inf. Process. Syst.*, pages 3111–3119, 2013.

[SA09] David Sussillo and L F Abbott. Generating coherent patterns of activity from chaotic neural networks. *Neuron*, 63(4):544–557, 2009.

[SS91] Hava T Siegelmann and Eduardo D Sontag. Turing computability with neural nets. *Appl. Math. Lett.*, 4(6):77–80, 1991.