Introduction to Recurrent Neural Networks

Ryan Miller

Introduction

- Convolutional neural networks are designed to exploit the *spatial* structures of images (or similarly formatted data)
- Recurrent neural networks are designed exploit the *sequential* structures of certain data types
  - For example, documents are a sequence of words with meaningful relative positions
  - Time-series, such as financial data, or recorded speech or music are other examples

Recurrence

In general terms, a recurrence relationship takes the form:

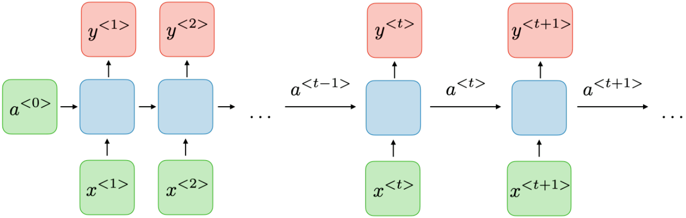
$$h_t = f_W(h_{t-1}, x_t)$$

- h_t is a "hidden state" at sequence position t
- f_W is a function involving weight parameters
- x_t is a input at position t

Weight parameters are shared across sequence positions (times).

Basic Architecture

The diagram below shows the basic architecture of a simple recurrent neural network:



- At each sequence position, indexed by t, there is a hidden state and an output, $y^{<t>}$
- Hidden states are a function of the previous state and the input $x^{<t>}$

Details

The following linear equation determines the hidden state:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

And the following equation determines the output:

$$y^{<t>} = g_2(\mathbf{W}_{ya}a^{<t>} + b_y)$$

- The weight matrices, W_{aa}, W_{ax}, and W_{ya}, and biases, b_a and b_y, are shared at every position
- g_1 and g_2 are activation functions

Simple Example

- Consider data consisting of a sequence of characters, and a model that aims to predict the next character in the sequence
  - For simplicity, we'll assume the only characters in this model's vocabulary are "h", "e", "l", and "o"
- Each input is a one-hot vector representing that letter
  - For example, "h" $= [1, 0, 0, 0]$, e $= [0, 1, 0, 0]$, etc.

Simple Example

Consider the input sequence: "hello"

- The first input is the vector $x^{<1>} = [1, 0, 0, 0]$
- We'll define the initial hidden state as $a^{<0>} = [0, 0, 0, 0]$

Thus, the input *h* produces the hidden state:

$$a^{<1>} = g_1(W_{aa} * [0, 0, 0, 0] + W_{ax} * [1, 0, 0, 0] + b_a)$$

Then this hidden state leads to the output:

$$y^{<1>} = g_2(W_{ya} a^{<1>} + b_y)$$

Simple Example (with numbers)

Suppose:

$$W_{aa} = \begin{pmatrix} 1 & 1 & -1 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0.5 \end{pmatrix}$$

$$W_{ax} = \begin{pmatrix} 0 & 0 & 1 & -1 \\ 1 & 1 & -1 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0.5 \end{pmatrix}$$

$$b_a = [0, 0, 0, 0]$$

and g_1 is the sigmoid function

- What happens when our first observed character, “h”, is input?

Simple Example (with numbers)

$$a^{<1>} = g_1\left(\begin{pmatrix} 0 & 0 & 1 & -1 \\ 1 & 1 & -1 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0.5 \end{pmatrix} * [1, 0, 0, 0] \right)$$

or

$$a^{<1>} = g_1([0, 1, -1, 0]) = [0.5, 0.73, 0.27, 0.5]$$

What is the role of $a^{<1>}$ in our network?

Simple Example (with numbers)

One place where $a^{<1>}$ is used is the generation of the predicted output at position t. Let's suppose:

$$W_{ay} = \begin{pmatrix} 0 & 1 & 0 & -1.5 \\ 1 & 0 & 0 & 0 \\ 0 & -0.5 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix}$$

and

$$b_y = [0, 0, 0, 0]$$

How do we find this output?

Simple Example (with numbers)

We have:

$$y^{<1>} = g_2\left(\begin{pmatrix} 0 & 1 & 0 & -1.5 \\ 1 & 0 & 0 & 0 \\ 0 & -0.5 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix} * [0.5, 0.73, 0.27, 0.5] \right)$$

or

$$y^{<1>} = g_2([-0.02, 0.5, -0.095, 0.23])$$

Simple Example (with numbers)

- If $g_2()$ is the softmax function, the predicted output is "e" (which happens to be correct)
- The associated probability is given by $\frac{exp(0.5)}{exp(-0.02)+exp(0.5)+exp(-0.095)+exp(0.23)} = 0.34$

Model Training

Similar to previous neural network architectures we've discussed, training a recurrent neural network consists of two important steps:

- Forward-propagation of examples to calculate the cost and other intermediate quantities
- Back-propagation to find the gradient and update the network's weights and biases

Model Training

The cost, at time-point t, is a function (such as cross-entropy loss) of $y^{<t>}$:

$$y^{<t>} = g_2(W_{ya} * [g_1([W_{aa}, W_{ax}] * [x^{<t>}, a^{<t-1>}] + b_a)] + b_y)$$

- Here, we've contactenated (stacked) matrices W_{aa} and W_{ax} and the vectors $x^{<t>}$ and $a^{<t-1>}$ to simplify the form of the model (since we can say $W_c = [W_{aa}, W_{ax}]$)
- We should note that $a^{<t-1>}$ is a function of $W = [W_{aa}, W_{ax}]$, so the chain rule in back-propagation will lead us to work backwards through time

We will not cover the details of gradient calculations for these models.

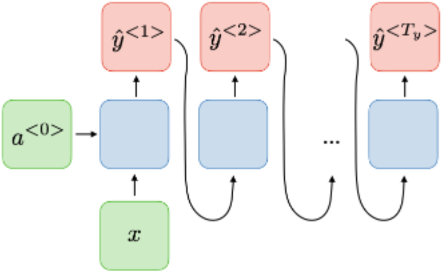
Applications

With minor modifications, the basic model architecture we've covered can be adapted to a wide range of applications, including:

1. *Generative Models* - a single input predicts a sequence of output (one-to-many)
2. *Sequence Classification Models* - a sequence input predicts a single output (many-to-one)
3. *Named Entity Recognition Models* - a sequence input predicts sequence output (many-to-many)

Generative Models

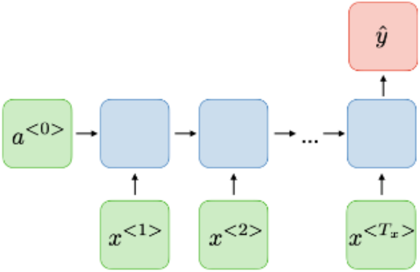
Suppose we've trained an RNN model and we provide a single input (perhaps the first letter or word in sequence). We can use the prediction $\hat{y}^{<1>}$ as the next sequential input



Notice how this architecture generates a sequential response from a single input.

Sentiment Classification

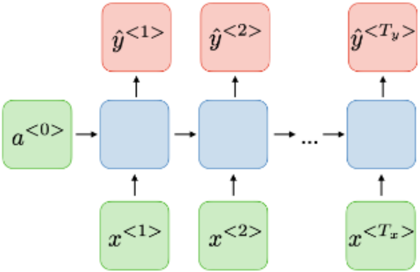
Suppose we're only interested in a single output corresponding to the entire input sequence:



- This model might be used to classify the sentiment of a text
- The architecture is similar to our example, except the weights in W_{ya} will be learned differently during training

Named Entity Recognition Models

Suppose we'd like to make use of every predicted output in our original model:



We used this architecture in our example, it can also be used to classify words as nouns, verbs, or adjectives while considering their position in a sentence.