Vanilla Recurrent Neural Network (RNN)

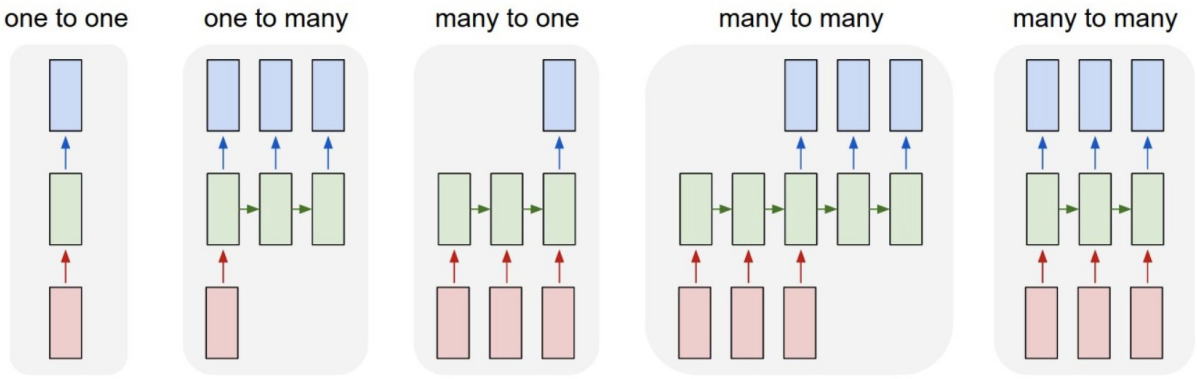
1.0) About

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Derived from feedforward neural networks, RNNs can use their internal state (memory) to process variable length sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition.They’re often used in Natural Language Processing (NLP) tasks because of their effectiveness in handling text

The term “recurrent neural network” is used indiscriminately to refer to two broad classes of networks with a similar general structure, where one is finite impulse and the other is infinite impulse. Both classes of networks exhibit temporal dynamic behavior. A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that can not be unrolled.

Both finite impulse and infinite impulse recurrent networks can have additional stored states, and the storage can be under direct control by the neural network. The storage can also be replaced by another network or graph, if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated state or gated memory, and are part of long short-term memory networks (LSTMs) and gated recurrent units. This is also called Feedback Neural Network.

Here are a few examples of what RNNs can look like:



2.0) “Many to one” vanilla RNN

Inputs - x_0, x_1, ..., x_n.

Output - y.

x_i vector can have arbitrary dimension.RNNs work by iteratively updating a hidden state h, which is a vector that can also have arbitrary dimension. At any given step t.

- The next hidden state h_t is calculated using the previous hidden state h_{t-1} and the next input x_t.
- The output y is calculated using h_n.

Rnn uses the same weights for each step(this makes a RNN recurrent). Vanilla RNN uses only 3 sets of weights to perform its calculations:

- W_{xh} - used for all x_t -> h_t links.
- W_{hh} - used for all h_{t-1} -> h_t links.
- W_{hy} - used for h_n -> y link.

Biases for RNN:

- b_h - added when calculating h_t.
- b_y - added when calculating y.

3.0) The Problem

We'll implement an RNN from scratch to perform a simple Sentiment Analysis task: determining whether a given text string is positive or negative.Each x_i will be a vector representing a word from the text. The output y will be a vector containing two numbers, one representing positive and the other negative. We'll apply Softmax to turn those values into probabilities and ultimately decide between positive / negative.

4.0) The Process

To start, we'll construct a vocabulary of all words that exist in our data(all words that appear in at least one training text).Next, we'll assign an integer index to represent each word in our vocabulary.

We can now represent any given word with its corresponding integer index(this is necessary because RNNs can't understand words).

Finally, recall that each input x_i to our RNN is a vector. We'll use one-hot vectors, which contain all zeros except for a single one. The "one" in each one-hot vector will be at the word's corresponding integer index.

If we have 15 unique words in our vocabulary, each x_i will be a 15-dimensional one-hot vector.

4.1) The Forward Phase Formulas

- $h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$.
- $y = W_{hy}h_n + b_y$.

We use tanh as an activation function for the first equation.

4.2) The Backward Phase Formulas

In order to train our RNN, we first need a loss function. We'll use cross-entropy loss, which is often paired with Softmax. Here's how we calculate it:

$$L = -\ln(p_c)$$

where p_c is our RNN's predicted probability for the correct class (positive or negative). For example, if a positive text is predicted to be 90% positive by our RNN, the loss is:

$$L = -\ln(0.90) = 0.105$$

Definitions:

- y represent the raw outputs from our RNN.
- p represent the final probabilities: p = softmax(y).
- c refer to the true label of a certain text sample, a.k.a. the "correct" class.
- L is the cross-entropy loss: $L = -\ln(p_c)$.

$$L = -\ln(p_c) = -\ln(\text{softmax}(y_c))$$

$$\frac{\partial L}{\partial y_i} = \begin{cases} p_i & \text{if } i \neq c \\ p_i - 1 & \text{if } i = c \end{cases}$$

For example, if we have p = [0.2, 0.2, 0.6] and the correct class is c = 0, then we'd get dL/dy = [-0.8, 0.2, 0.6].

Gradients for W_{hy} and b_y(h_n is the final hidden state):

$$\frac{\partial L}{\partial W_{hy}} = \frac{\partial L}{\partial y} * \frac{\partial y}{\partial W_{hy}}$$

$$y = W_{hy} h_n + b_y$$

$$\frac{\partial y}{\partial W_{hy}} = h_n$$

$$\frac{\partial L}{\partial W_{hy}} = \boxed{\frac{\partial L}{\partial y} h_n}$$

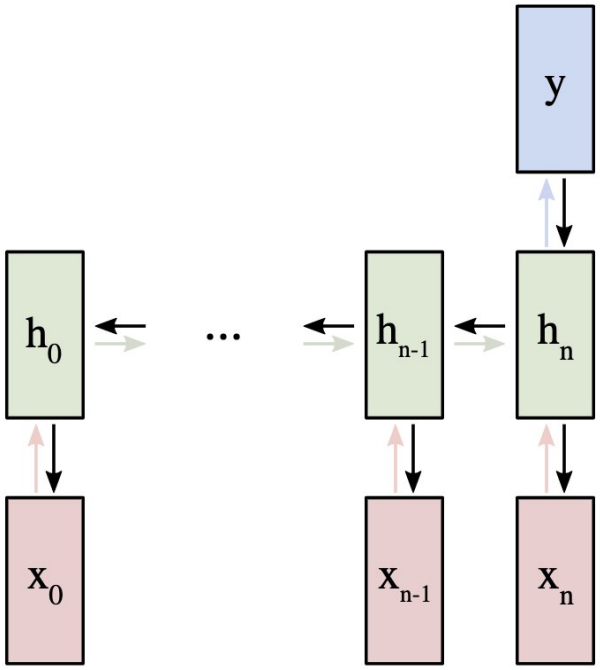
$$\frac{\partial y}{\partial b_y} = 1$$

$$\frac{\partial L}{\partial b_y} = \boxed{\frac{\partial L}{\partial y}}$$

Gradients for W_{hh}, W_{xh} and b_h:

$$\frac{\partial L}{\partial W_{xh}} = \frac{\partial L}{\partial y} \sum_t \frac{\partial y}{\partial h_t} * \frac{\partial h_t}{\partial W_{xh}}$$

Because changing W_{xh} affects every h_t, which all affect y and ultimately L. In order to fully calculate the gradient of W_{xh}, we'll need to backpropagate through all timesteps, which is known as Backpropagation Through Time (BPTT):



W_{xh} is used for all x_t -> h_t forward links, so we have to backpropagate back to each of those links.

Once we arrive at a given step t, we need to calculate dh_t/dw_{xh}:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

The derivative of tanh:

$$\frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

Using Chain Rule we get:

$$\frac{\partial h_t}{\partial W_{xh}} = \boxed{(1 - h_t^2)x_t}$$

Similarly:

$$\frac{\partial h_t}{\partial W_{hh}} = \boxed{(1 - h_t^2)h_{t-1}}$$

$$\frac{\partial h_t}{\partial b_h} = \boxed{(1 - h_t^2)}$$

dy/dh_t(calculating recursively):

$$\frac{\partial y}{\partial h_t} = \frac{\partial y}{\partial h_{t+1}} * \frac{\partial h_{t+1}}{\partial h_t}$$
$$= \frac{\partial y}{\partial h_{t+1}}(1 - h_t^2)W_{hh}$$

We'll implement BPTT starting from the last hidden state and working backwards, so we'll already have dy/dh_{t+1} by the time we want to calculate dy/dh_t.The exception is the last hidden state, h_n:

$$\frac{\partial y}{\partial h_n} = W_{hy}$$

5.0) Results

Epochs: 1000.

Learn rate: 0.02.

```
<----------Vocabulary size(number of words) is 18---------->
Epoch 100
Train loss: 0.688136    Train accuracy: 0.551724
Test  loss: 0.696814    Test  accuracy: 0.5
Epoch 200
Train loss: 0.665736    Train accuracy: 0.62069
Test  loss: 0.746577    Test  accuracy: 0.5
Epoch 300
Train loss: 0.841534    Train accuracy: 0.37931
Test  loss: 0.659748    Test  accuracy: 0.5
Epoch 400
Train loss: 0.51811     Train accuracy: 0.689655
Test  loss: 0.505747    Test  accuracy: 0.7
Epoch 500
Train loss: 0.0109037   Train accuracy: 1
Test  loss: 0.0294207   Test  accuracy: 1
Epoch 600
Train loss: 0.00320424  Train accuracy: 1
Test  loss: 0.00780295  Test  accuracy: 1
Epoch 700
Train loss: 0.00189234  Train accuracy: 1
Test  loss: 0.00453078  Test  accuracy: 1
Epoch 800
Train loss: 0.00135413  Train accuracy: 1
Test  loss: 0.0031085   Test  accuracy: 1
Epoch 900
Train loss: 0.00105386  Train accuracy: 1
Test  loss: 0.00236587  Test  accuracy: 1
Epoch 1000
Train loss: 0.000855745 Train accuracy: 1
Test  loss: 0.00191447  Test  accuracy: 1
```

6.0) Hints

Compile command(Linux terminal): g++ reader.cpp tools.cpp rnn.cpp main.cpp -o a.out