Recurrent Neural Networks (RNNs)

Implementing an RNN from scratch in Python.

Jason Brownlee Jul 5, 2018 · 14 min read

The main objective of this post is to implement an RNN from scratch and provide an easy explanation as well to make it useful for the readers. Implementing any neural network from scratch at least once is a valuable exercise. It helps you understand the mechanics and assumptions of the algorithm. Once you are implementing an RNN which has its own complexity and this provides us with a good opportunity to hone our skills.

There are various tutorials that provide a very detailed information of the mathematics behind RNN. I have mentioned some of the important ones at end of this post. I could understand the workings of an RNN rather quickly but what troubled me most was going through the BPTT calculations and its implementation. I had to spent some time to understand and finally put it all together. Without wasting any more time, let us quickly go through the basics of an RNN first.

What is an RNN?

A recurrent neural network is a neural network that is specialized for processing a sequence of data $x(t), x(1), \ldots, x(\tau)$ with the time step index t ranging from 1 to τ. For tasks that involve sequential inputs, such as speech and language, it is almost better to use RNNs. As a MLP predicts, if it has been trained on a set of images to predict a certain number, it recognizes a certain number. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way of looking at it is that RNNs have a "memory" which captures information about what has been calculated so far.

Architecture : Let us briefly go through a basic RNN network.

Unfolded RNN

The left side of the above diagram shows a notation of an RNN and on the right side an RNN being unrolled (or unfolded into a full network. By unrolling we mean that we write out the network for the complete sequence. For example, if the sequence we are talking about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word.

Input: $x(t)$ is taken as the input to the network at time step t. For example, x_1 could be a one-hot vector corresponding to a word of a sentence.

Hidden state: $h(t)$ represents a hidden state at time t and acts as "memory" of the network. $h(t)$ is calculated based on the current input and the previous time step's hidden state: $h(t) = f(Ux(t) + Wh(t-1))$. The function f is taken to be a non-linear transformation such as tanh, ReLU.

Weights: The RNN has input to hidden connections parameterized by a weight matrix U, hidden to hidden recurrent connections parameterized by a weight matrix W, and hidden to output connections parameterized by a weight matrix V and all these weights (U, V, W) are shared across time.

Output: $o(t)$ illustrates the output of the network. In the figure I just put an arrow after $o(t)$ which is also often subjected to non-linearity, especially when the network contains further layers downstream.

Forward Pass

The figure does not specify the choice of activation function for the hidden units. Before we proceed we make few assumptions: 1) we assume the hyperbolic tangent activation function for hidden layers. 2) We assume that the output is discrete, as if the RNN is used to predict words or characters. A natural way to represent such outputs is softmax distribution. 3) We are giving the un-normalized log probabilities of each possible value of the discrete variable. We can then apply the softmax operation as a pre-processing step to obtain a vector $\hat{y}$ of normalized probabilities over the output.

The RNN forward pass can thus be represented by below set of equations.

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

This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. If we add the total loss for sequence of x values paired with a sequence of y values would then be just the sum of the losses over all the time steps. We assume that the output $o(t)$ are used as the argument to the softmax function to obtain the vector $\hat{y}$ of probabilities over the output. We also assume that the loss L is the negative log-likelihood of the true target $y(t)$ given the input so far.

Backward Pass

The gradient computation involves performing a forward propagation pass moving left to right through the graph above followed by a backward propagation pass moving right to left through the graph. The runtime is $O(\tau)$ and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may be computed only after the previous one. States computed in the forward pass must be stored until they are used in the backward pass, so the memory cost is also $O(\tau)$. The back-propagation algorithm applied to the unrolled graph with $O(\tau)$ cost is called back-propagation through time (BPTT). Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps.

Computing Gradients

Given our loss function L, we need to calculate the gradients for our three weight matrices U, W and $V$and biases b and c and update them with a learning rate α. Similar to normal back-propagation, the gradient gives us a sense of how the loss is changing with respect to each weight parameter. We update the weights W to minimize loss with the following equation:

$$W = W - \alpha \frac{\partial L}{\partial W}$$

The same is to be done for the other weights U, V, b, c as well.

Let us now compute the gradients by BPTT for the RNN equations above. The nodes of our computational graph include the parameters U, V, W, b and c as well as the sequence of nodes indexed by t for $x(t), h(t)$ and $o(t)$. For each node n we need to compute the gradient $\frac{\partial L}{\partial n}$ recursively, based on the gradient computed at nodes that follow it in the graph.

Gradient with respect to output $o(t)$ is calculated assuming the $o(t)$ are used as the argument to the softmax function to obtain the vector $\hat{y}$ of probabilities over the output. We also assume that the loss is the negative log-likelihood of the true target $y(t)$.

$$(\nabla_{o^{<t>}} L)_i = \frac{\partial L}{\partial L^{<t>}} \frac{\partial L^{<t>}}{\partial o_i^{<t>}} = \hat{y}_i^{<t>} - \mathbb{1}_{y^{<t>}=i}$$

Please refer here for deriving the above elegant solution.

Let us now understand how the gradient flows through hidden state $h(t)$. For this we need to draw the below diagram that at time t, hidden state $h(t)$ has gradient flowing from both current output and the next hidden state.

Gradient at hidden state $h(t)$ at time t

We work our way backward, starting from the end of the sequence. At the final time step τ, $h(\tau)$ only has $o(\tau)$ as a descendant, so its gradient is simple:

$$\nabla_{h^{<\tau>}} L = V^T \nabla_{o^{<\tau>}} L$$

We can then iterate backward in time to back-propagate gradients through time, from $t=\tau-1$ down to $t=1$, noting that $h(t)$ ($t<\tau$) has as descendants both $o(t)$ and $h(t+1)$. Its gradient is then given by:

$$\nabla_{h^{<t>}} L = W^T (\nabla_{h^{<t+1>}} L)(1 - h^{<t+1>2}) + V^T \nabla_{o^{<t>}} L$$

Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes. The gradient calculations using the chain rule for all parameters is:

$$\nabla_c L = \sum_t \frac{\partial o^{<t>}}{\partial c} \nabla_{o^{<t>}} L = \sum_t \nabla_{o^{<t>}} L$$
$$\nabla_b L = \sum_t \frac{\partial h^{<t>}}{\partial b} \nabla_{h^{<t>}} L = \sum_t \nabla_{h^{<t>}} L (1 - h^{<t>2})$$
$$\nabla_V L = \sum_t \nabla_{o^{<t>}} L \, h^{<t>T}$$
$$\nabla_W L = \sum_t \nabla_{h^{<t>}} L (1 - h^{<t>2}) \, h^{<t-1>T}$$
$$\nabla_U L = \sum_t \nabla_{h^{<t>}} L (1 - h^{<t>2}) \, x^{<t>T}$$

We set our interested to derive these equations here, rather implementing them. There are very good posts here and here providing detailed derivation of these equations.

Implementation

We will implement a full Recurrent Neural Network from scratch using Python. We will try to build a text generation model using an RNN. We train our model to predict the probability of a character given the preceding characters in a sequence. Given enough training data and sufficient training time we sample a new character from the predicted probabilities, and repeat the process until we have a full sentence. The implementation is from Andrej Karpathy's post with building a character level RNN. Here we will discuss the implementation details step by step.

General steps to follow:

1. Initialize weight matrices U, V, W from random distribution and bias b, c with zero
2. Forward propagation to compute predictions
3. Compute the loss
4. Back-propagation to compute gradients
5. Update weights based on gradients
6. Repeat steps 2-5

Step 1 Initialize

To start with the implementation of the basic RNN cell, we first define the dimension of the various parameters U,V,W,b,c.

Dimensions Let's assume we pick a vocabulary size `vocab_size` = **3000** and a hidden layer size `hidden_size` = **100**. Then we have:

Vocabulary size can be the number of unique chars for a char based model or number of unique words for a word based model.

With our five layers parameters and other model parameters, let us start defining our RNN cell.

Step 2: Forward pass

Straightforward as per our equations for each timestamp t, we calculate hidden state h(t) and output o(t) applying softmax to get the probability for the next character.

Proper initialization of weights seems to have an impact on training results: there has been lot of research in this area. It turns out that the best initialization depends on the activation function (tanh in our case) and one recommended approach is to initialize the weights randomly in the interval from $[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$, where n is the number of incoming connections from the previous layer.

Step 3: Forward pass

Straightforward as per our equations for each timestamp t, we calculate hidden state h(t) and output o(t) applying softmax to get the probability for the next character.

Computing softmax and numerical stability

Softmax function takes an N-dimensional vector of real numbers and transforms it into a vector of real number in range (0,1) which add upto 1. The mapping is done using the below formula.

The implementation of softmax is:

Though it looks fine however when we call this softmax with a bigger number like below it gives 'inan' values

The numerical range of the floating point numbers used by Numpy is limited. For floatst, the maximal representable number is on the order of 10^{308}. Exponentiation in the softmax function makes it possible to easily overshoot this number, even for fairly modest-sized inputs. A nice way to avoid this problem is by normalizing the inputs to be not too large or too small. There is a small mathematical trick applied with Numpy for details. So our softmax looks like:

Step 3: Compute Loss

Since we are implementing a text generation model, the next character can be any of the unique characters in our vocabulary. So our loss will be cross-entropy loss. In multi class classification we take the sum of log loss values for each class prediction in the observation.

- M = number of possible class labels (unique characters in our model)
- y_{c} = a binary indicator (0 or 1) of whether class label c is the correct classification for observation o
- p_{c} = the model's predicted probability that observation

Step 4: Backward pass

If we refer to the BPTT equations, the implementation is as per the equations. Sufficient comments added to understand the code.

Issues

While it is principle the RNN is a simple and powerful model, in practice, it is hard to train properly. Among the main reasons why this model is so unwieldy are the vanishing gradient and exploding gradient problems. While training using BPTT the gradients have to travel from the last cell all the way to the first cell. The product of these gradients can go down or increase exponentially. The exploding gradients problem refers to the large increase in the norm of the gradient during training. The exploding gradients problem refer to the opposite behavior, when long term components go exponentially fast to norm 0, making it impossible for the model to learn correlations between temporally distant events.

Whereas the exploding gradients can be fixed with gradient clipping technique as it and is the example code here, the vanishing gradient issue is still a major concern with an RNN.

The vanishing gradient limitation was overcome by various networks such as long short-term memory (LSTM) and gated recurrent units (GRUs), and residual networks (ResNet), where the first two are the most widely used variants in NLP applications.

Step 5 Update weights

Using BPTT we calculated the gradient for each parameter of the model. It is now time to update the weights.

Step 6 Update weights

Using BPTT we calculated the gradient for each parameter of the model. It is now time to update the weights.

Let us see how our RNN is learning after a few epochs of training.

The output looks more like real text with word boundaries and some grammar as well. So our baby RNN has started learning the language and able to predict the next few words.

The implementation presented here just meant to be easy to understand and grasp the concepts. In case you want to play around the model hyper parameters, the notebook is here.

Bonus: Want to visualize what's actually going on while training an RNN, watch here.

Hope it was useful for you. Thanks for the read.

References:

[1] http://www.deeplearningbook.org/contents/rnn.html

[2] http://colah.github.io/posts/2015-August-Understanding-LSTMs/

[2] http://gist.github.com/karpathy/d4dee566867f8291f086

[3] http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

[4] https://medium.com/towards-artificial-intelligence/whirlwind-tour-of-rnns-a11effb7808f

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter

Machine Learning | Deep Learning | Recurrent Neural Network | Text Generation | Neural Networks

337 4

More from Towards Data Science

Follow

Your home for data science. A Medium publication sharing concepts, ideas and codes.

Read more from Towards Data Science

More From Medium

BERT BERT BERT!

Tony Jesuthasan

Diabetes Prediction

Choudharyuttam

Microsoft Malware Prediction Using Classical Machine Learning Algorithms

Jugaloza in The Startup

Principal Component Analysis in Dimensionality Reduction with Python

Amit Chauhan in Towards AI

[Archived Post] Deep RL Bootcamp Lecture 7 SVG, DDPG, and Stochastic Computation Graphs (John...

generaluseage9091

Connor Shorten — Florida Atlantic University & Founder at Henry AI Labs on Deep Learning

Dmitry Kan

Machine Learning.... What is it?

Ji Hoon Chung

Text-to-Image Synthesis

Nikunj Gupta in AI Club @IIITB

Medium

About Write Help Legal