Using PyTorch to mimic Shakespeare: creating a recurrent neural network to emulate famous works of literature and other texts

Jason Bard, Samrat Baral, Pedro Maia, Deokgun Park, Nethali Fernando
Departments of Math and Computer Science, The University of Texas at Arlington, Arlington, TX 76019

Abstract

Recurrent neural networks (RNNs) are a class of artificial neural networks whose nodes form a directed graph. They contain an internal state memory for modeling sequence data and processing sequences of vectors, which allows them to recognize handwriting, analyze audio, predict text, or emulate literature in the styles of different authors. In 2015, Karpathy published code for an RNN heavily utilizing Numpy with the end result of emulating the texts of Shakespeare. We utilize the softmax function as our saturation and cross-entropy between layers. In our work, we transcribe this code into PyTorch to leverage its capability. PyTorch is a Python package that serves as an open source machine learning framework for subjects like Deep Learning. We use an autoencoder to produce 1-of-k encoding for each individual text character. We demonstrate our results with the networks trained using predetermined parameters. Furthermore, we take advantage of the flexibility of PyTorch to experiment with different cost and activation functions, learning and training protocols, and other improvements. This could lead to using and/or favoring PyTorch to create and distribute RNNs with GPU acceleration in future studies.

Background

In 2015, Andrej Karpathy published "The Unreasonable Effectiveness of Neural Networks," and with it, he published a basic RNN/LSTM on GitHub that analyzes the complete works of William Shakespeare as a text file and produces a character string attempting to create new prose in Shakespeare's style. It utilizes NumPy, a package of the Python programming language that is useful in handling arrays and matrices. However, PyTorch was soon developed by Facebook and was first released to the public more than a year later. It uses tensors in lieu of arrays, and it was built to ease the creation of neural networks, which heavily depend on array manipulation. We aim to mimic the structure of Karpathy's network using PyTorch rather than NumPy in order to see if we can evaluate and mimic other text files of a similar structure.

Character-Level Recurrent Neural Networks

Recurrent neural networks allow previous outputs to be used in the next input while retaining its own distinct hidden state. This is especially useful when the data is not a static image but is instead a text file. When being trained by a neural network, text needs all its previous outputs as well as its new inputs to be fed through the algorithm so that predictive character patterns can be picked up and emulated. In a neural network, a text file (or any file) of any size can be processed and used to train the machine. In addition, our model size does not have to increase with the size of the data. In order to turn the text characters into vectors, we must put them through a 1-of-k encoding, also known as a one-hot encoding. This involves taking the set of characters that appear in the text file and identifying them with a series of vectors; these vectors have a 1 in one position to identify the unique character and 0s in every other position.

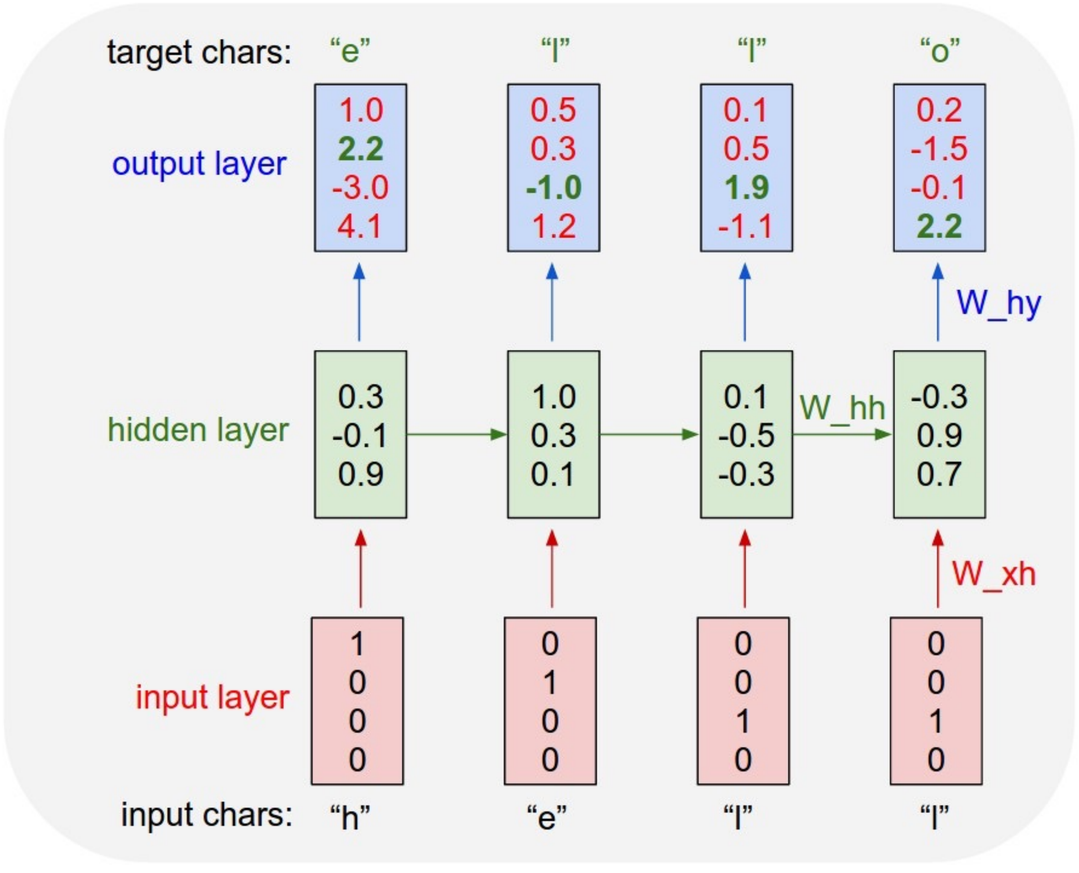

Figure 1. A representation of the "inner workings" of the RNN. Encoding letters as 1-of-k vectors, running them through the algorithm, and getting the output. For example, the word "hello."

Methods

We use cross-entropy in order to turn each hidden state into a probability distribution, which will be used in predicting the next letter in the sequence. Additionally, we use autoencoders, which are used to take unlabeled data, possibly in many dimensions, and run them through a dimensionality reduction algorithm in order to keep as much of the significance data as possible while taking up little CPU/GPU space. Furthermore, we run each hidden state through the softmax function, as shown below. It is useful as a probability distribution because it makes all the vector components sum to 1 as well as preserving the overall structure of the tensor. Cross-entropy involves a logarithmic function, so doing a scalar division would not be optimal. Using the softmax function is the way to keep everything on a scale.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Figure 2. The mathematical definition of the softmax function.

Acknowledgements

Thank you to Dr. Pedro Maia and Dr. Deokgun Park for their guidance during this project and in its future completion. In addition, thanks to Nethali Fernando for helping us conceptualize autoencoders, and to SM Mazharul Islam for helping us with some of the basics.

For further information, please contact us at jason.bard@mavs.uta.edu and samrat.baral@mavs.uta.edu.

Link to the original article by Andrej Karpathy