Result Ranking Neural Network

> ⚠ MGNLPER-11 - Jira project doesn't exist or you don't have permission to view it.

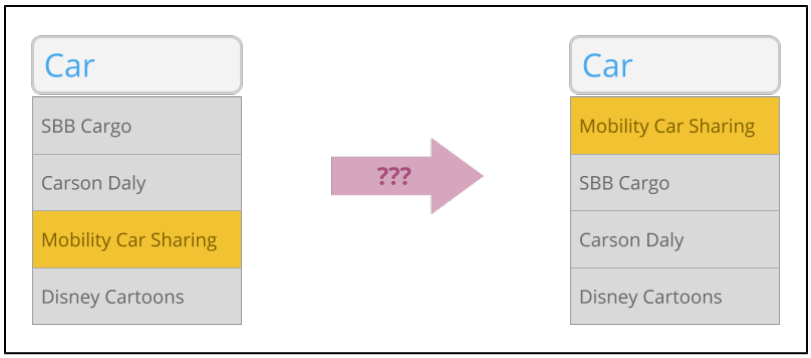
This document describes the architecture of the neural network used for Periscope result ranking, and what intentions lead to it.

It is assumed the reader has some basic knowledge about deep neural networks, and in particular convolutional ones. A good place to get started is **Skymind's Beginner's Guide**.

Corresponding code lives in the machine-learning git repository.

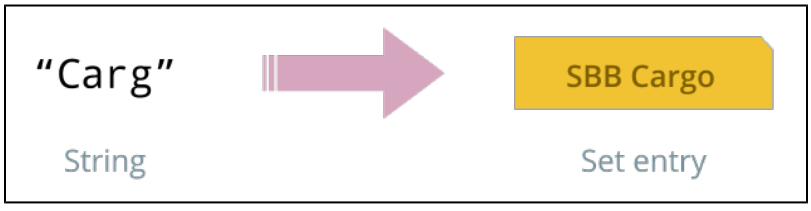
Goal

When searching in a large pool of content, users tend to select certain results more often than others. With a smart ranking solution, we want to learn from what users select and bubble those more relevant items up during the next search, so they are in closer reach for the user.



Implementation

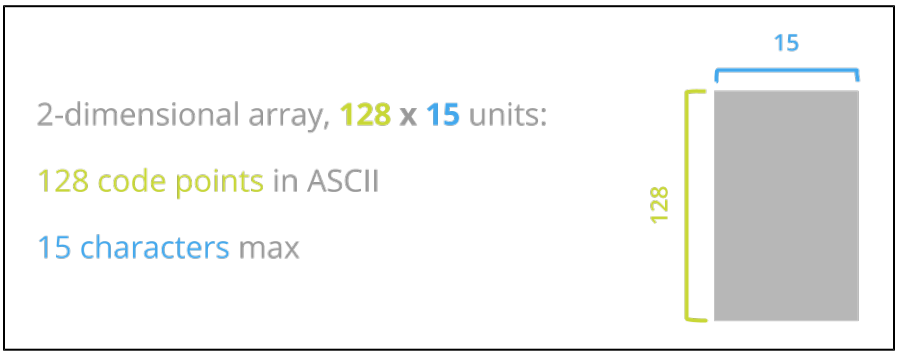
We're dealing with a situation where we want to map a search query to a picked result. Type-wise, that's a mapping from a **string** to a **set entry**.



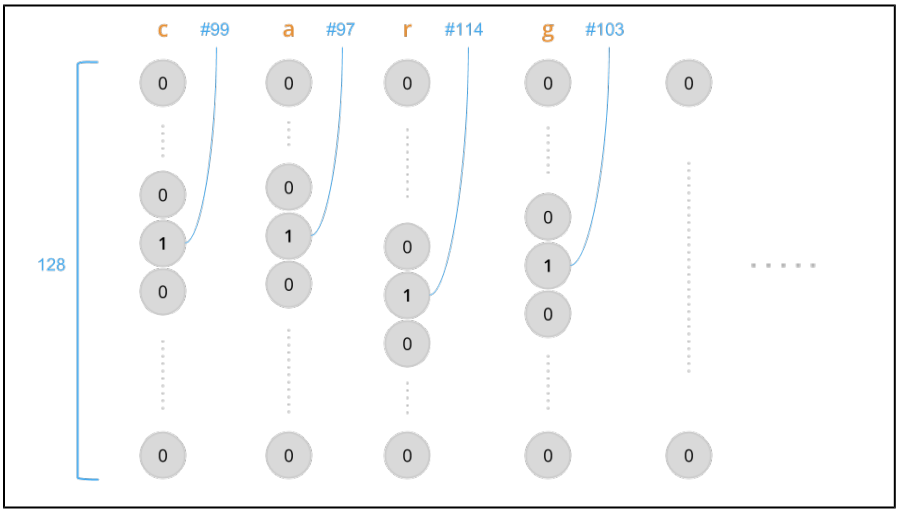
Neural networks operate on numbers, where each unit of the input layer gets receives a floating point number and each output layer unit yields one (during a forward pass). This means we somehow have to map strings to a (fixed-size) list of floats for inputs, and set entries to a list of floats for outputs.

Encoding input queries

As a representation for a query string, we use a two-dimensional input layer consisting of 128 x 15 units. The idea is that each column represent one character (out of 128 from the ASCII set), and we support up to 15 characters in a query (longer ones are cropped).

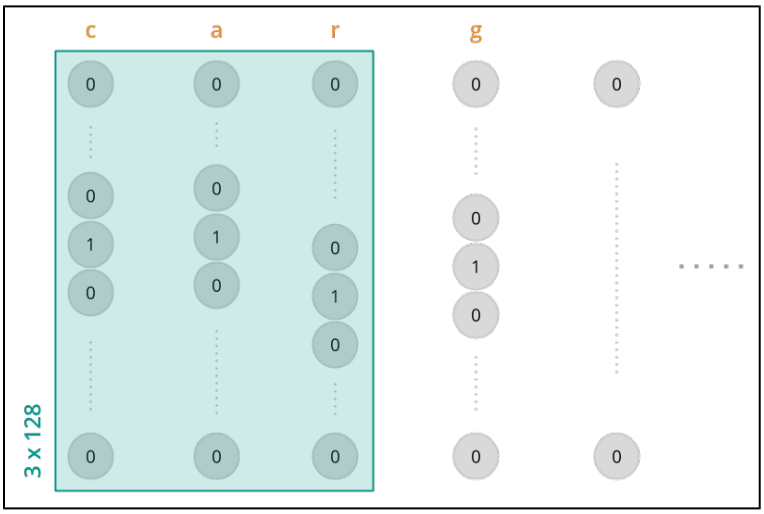


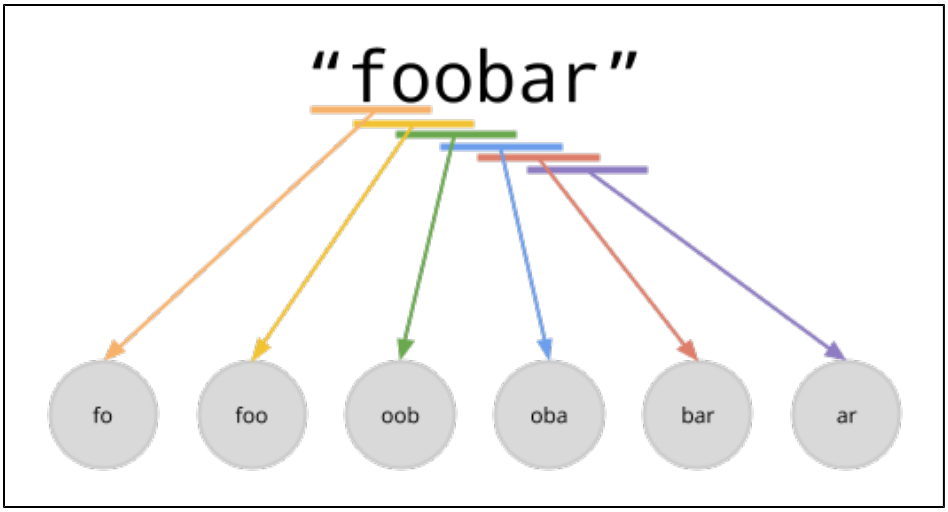
Each character of the query string is mapped to an input layer column using a one-hot encoding. That is, given a character's ASCII code *i*, the corresponding column's unit at row *i* receives 1 as input, while all others are set to 0.



Initial convolution

After the input layer, the first step is a convolution with kernel size **128 x 3**, semantically meaning that always 3 adjacent characters are convolved into one unit of the next layer. At the same time, the two-dimensional input layer data is flattened to one dimension.



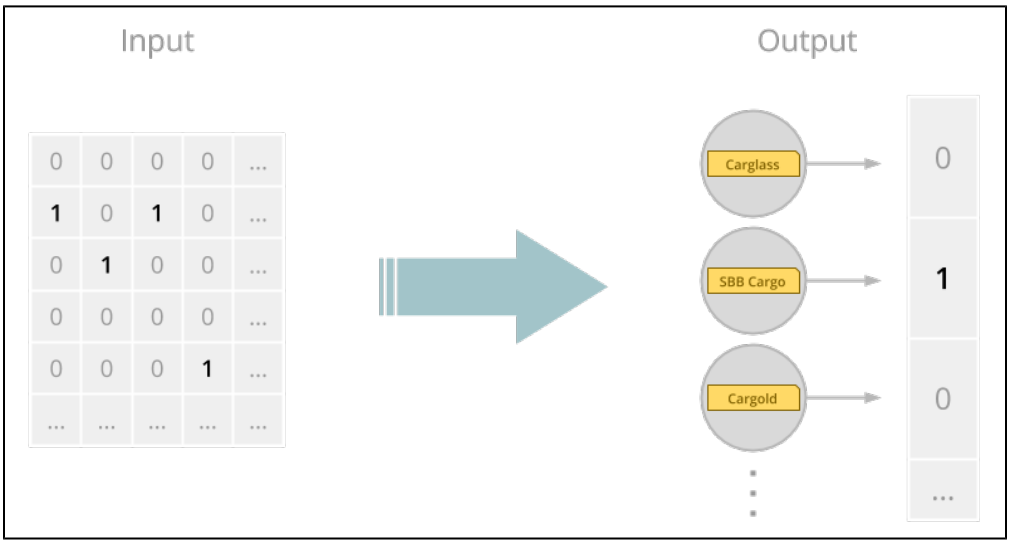


Mapping between results and output units

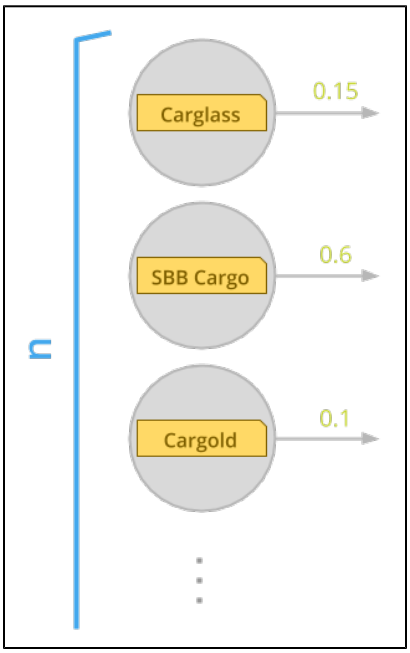
Each potential result from the set is represented by one output layer unit. However, the set of potential results generally keeps on growing over time while neural network output layers are generally fixed in size. To overcome this discrepancy, we do two things: First, we start out with a fairly large output layer (10k units at the time of writing) and dynamically populate the mapping as the potential result set grows. That is, initially most output units have no corresponding result and thus no semantic meaning, even though they are already in place. While new potential results are acquired, they get one-by-one assigned to yet-unused output units. Once all output units are already assigned and we get new results to be mapped, we forget oldest results and remove unit association in order to re-map them with new results. During this step, we also reset weights of edges leading to those units in order to unlearn any previous training. Re-assignment happens in the manner of a circular buffer.

En-/Decoding

For training passes, the result set is mapped to an output layer data array as a one-hot encoding. That is, the picked result's unit has value 1, and all others 0.

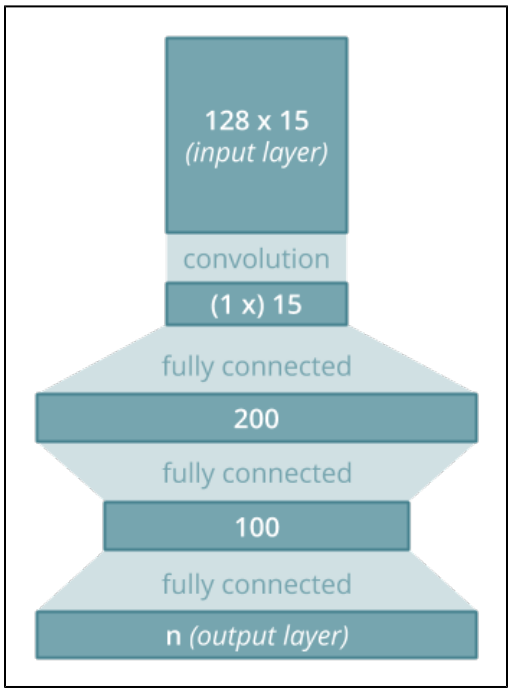


For non-training forward passes, numbers in the output array are interpreted as probability distribution of the user's intention, later to be used as base for result ordering. The higher an output number is, the more relevant looks its corresponding results, and the higher it gets ranked in the result list.



Neural network architecture overview

In addition to input and output layers explained above, we have two fully-connected intermediate layers of sizes 200 and 100, respectively. This is generally useful to approximate mappings without any previously known properties, basically an unopiniated brute force approach.



Learning rate

One special thing about hyper-parameter configuration is the unusually high learning rate of `0.01`. This stems from the fact that we're dealing with small amounts of data. Even after a few (less than 10) times of picking results, users should have significantly better rankings, whereas other neural network training scenarios often deal with thousands or millions of data points for training. In addition, we're doing online learning, which means we learn on-the-go and don't have a predefined, fixed training data set we can iterate over multiple times (epochs). Thus, we need to weight the single run we do stronger.

Persistence

In order to keep learned ranking information across sessions and application runs, we serialize trained neural networks and corresponding label lists to JCR (both on a per-user basis). Because that operation is somewhat computationally expensive, that operation is run in a dedicated thread and debounced such that it only gets executed once per a given time interval (30 seconds by default).