Jeremy Stanley Follow
VP Data Science at Instacart, conquering the world one at a time.
yesterday · 11 min read

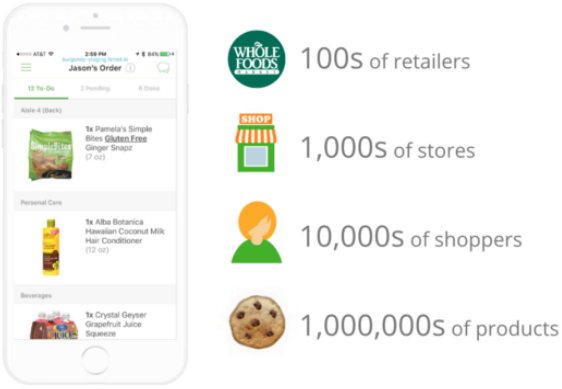
Deep Learning with Emojis (not Math)

Sorting shopping lists with deep learning using Keras and Tensorflow.

Shopping for groceries is hard.

Stores are large and have complex layouts that are confusing to navigate. The hummus you want could be in the dairy section, the deli section, or somewhere else entirely. Efficiently navigating a store can be a daunting task.

At Instacart, our customers can order millions of products from hundreds of retail partners. Our fleet of tens of thousands of personal shoppers must find these items at thousands of store locations. We are always looking for opportunities to enable our shoppers to move faster.

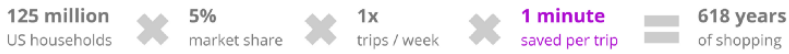


Shopping with Deep Learning

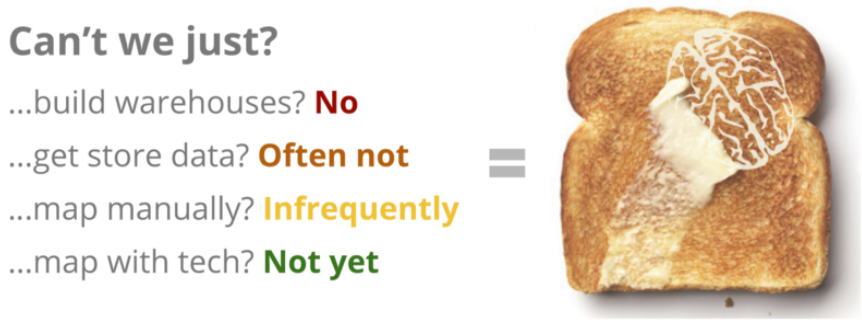
Enter Deep Learning

By observing how our shoppers have picked millions of customer orders through our app, we have built models that predict the sequences our fastest shoppers will follow. Then, when a shopper is given a new order to pick, we use this predicted fastest sequence to sort the items for them.

This approach has reduced our shopping times by *minutes* per trip. At scale, every minute saved will translate into **618 years of shopping time** per year.



So how do we do it? First, we can't build warehouses, get accurate store data or map each store location. Also, traditional machine learning approaches (we XGBoost) don't work either due to the sequential nature of the problem.



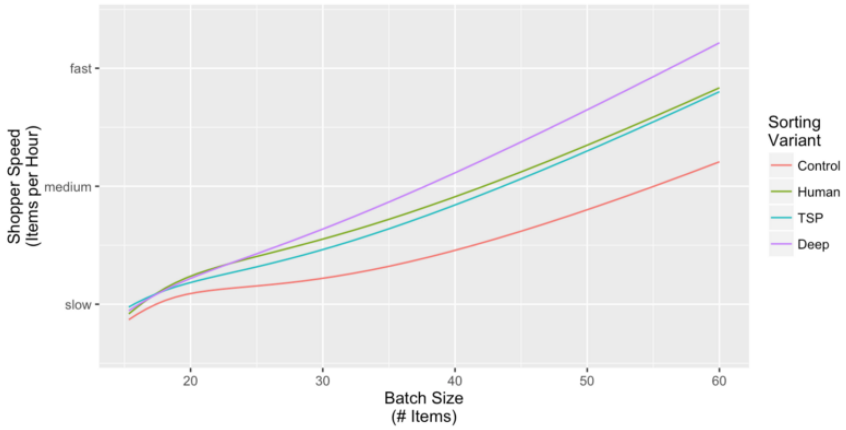
So instead, we spread some deep learning on it.

What We Tested

We ran a test where every batch (a set of items to be picked by a shopper) was randomly assigned to one of four list sorting algorithms:

- *Control (red):* departments sorted in a random order; items sorted alphabetically within departments
- *Human (green):* aisles sorted by humans using store layouts; items sorted alphabetically within aisles
- *TSP (teal):* a traveling salesman solution using average inter-department picking times; items sorted alphabetically within departments
- *Deep (purple)*: our final deep learning architecture, which directly sorts items in the batch

We then look at how each sort performed in terms of shopper speed (y-axis) as a function of the size of the batch picked (x-axis):



Note that we mask the exact picking speeds as this is a competitively sensitive KPI for Instacart.

The control sort performed the worst (as expected). The TSP and Human sorts performed significantly better, but were statistically no different from each-other.

The deep learning model beat them all by a large margin—the increase in picking speed from human to deep learning is 50% higher than from control to human at large batch sizes.

In the remainder of this post, we will define the problem (using emojis of course), and then introduce a naive initial architecture. We will inspect that architecture and point out some key flaws, and then conclude with a final architecture that is more efficient and effective.

List Sorting in Keras

Problem Definition

Suppose a customer ordered 10 items and their personal shopper picked those items in this sequence:

We can observe this sequence, as our shoppers weigh or scan bar codes for every item picked. In order to learn the sequence, we need to formulate this as a supervised learning problem. Suppose that we are looking back in time at this order, and we pause after the shopper picks the :

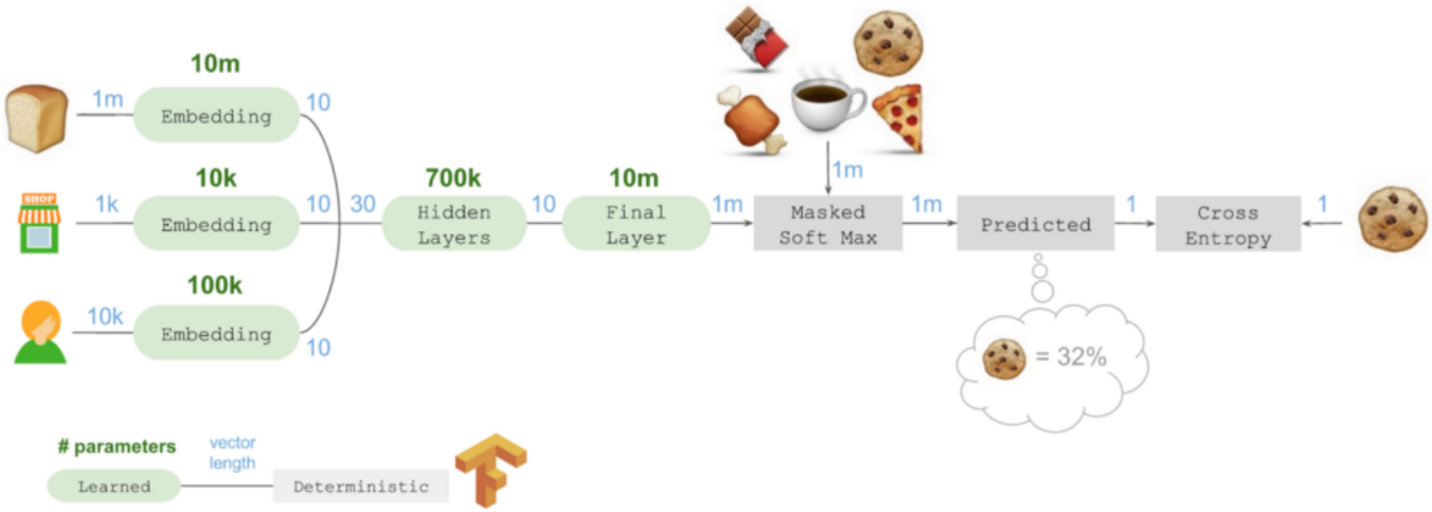
We want to predict the next item that the shopper will pick (a in this case), given they just picked the and can choose from one of the five candidate products remaining ( ). Are you getting hungry yet?

In emoji-math (one of the benefits of working at Instacart is doing emoji math), we can re-write this as:

Note that this probability is non-trivial to compute. It's not enough to ask how often cookies are picked after bread. Cookies may be incredibly common (they are in my household), so this naive probability might be biased high. We want to measure how likely cookies are to be chosen given we can only choose from a fixed set of remaining items.

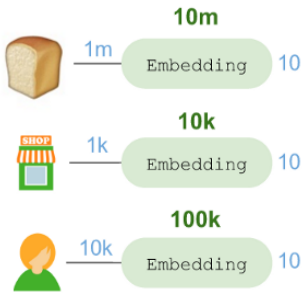
Initial Architecture

Our initial deep learning architecture, implemented in Tensorflow using Keras, was the following:



We begin on the left hand side with the product that was last picked ( ), and end on the right hand side with a prediction for the product that was picked next ( ). Along the way, we have to account for the candidate set of products that could have been selected.

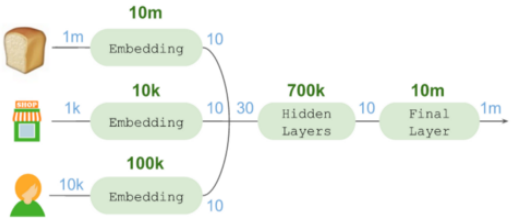
First, let's cover the embeddings:



Using the `Embedding` layer in Keras, we embed the into a 10-dimensional vector space. This uses 10 million parameters since we have 1 million potential products to consider. Similarly, we embed the store locations (1k stores = 10k parameters) and shoppers (10k shoppers = 100k parameters).

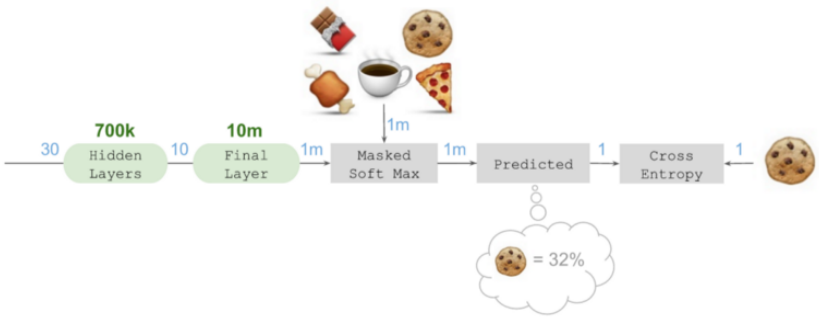
The store location embedding enables the model to learn store layouts and generalize learnings across retailers and locations. The shopper embedding learns that shoppers may take consistently different routes through stores.

We proceed by concatenating all of these embeddings together into a 30-dimensional vector, which is fed into a sequence of fully connected `Dense` hidden layers (700k parameters) with non-linear activations (`relu`). This produces a final vector of length 10 which captures all of the relevant information about the prior product, store location and shopper needed to decide which future item is most likely to be picked next.



To produce a prediction, we have to 'fan out' in a final layer, which operates like a reverse embedding. This projects our 10-dimensional vector into the space of potential future products. This was accomplished with a `Dense` layer using a `linear` activation function.

Then we can compute a masked-soft max, which turns our million length candidate vector into zeros for all non-candidates and positive probability estimates for the 5 candidate products. This was accomplished by combining a `Lambda` layer to exponentiate, followed by a `Merge` layer to mask the candidates, followed by another `Lambda` layer to ensure the probabilities sum to 100%.



Suppose that in this case we predict that (the right answer) is 32% likely to be chosen next. Then this prediction can be fed into a `categorical_crossentropy` loss function along with the indicator that was, in-fact, chosen next in this case. Tensorflow can then backpropogate the errors to train the final, hidden and embedding layers, and we can learn this model at scale.

Keras Code

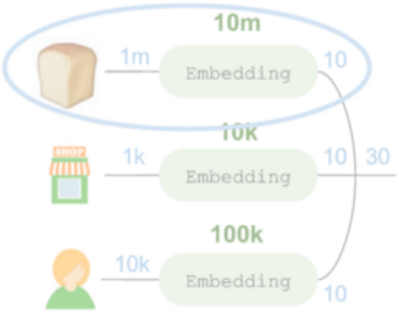
This architecture can be implemented in Keras using the following code:

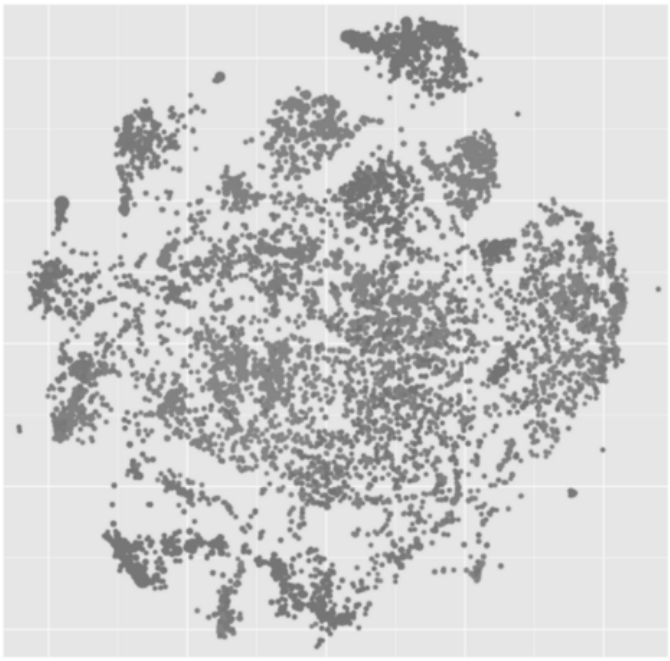
```
from keras.models import Model
from keras.layers.core import Dense, Reshape, Lambda
from keras.layers import Input, Embedding, merge
from keras import backend as K

# Number of product IDs available
N_products = 1000000
N_stores = 1000
N_shoppers = 10000

# Integer IDs representing 1-hot encodings
prior_in = Input(shape=(1,))
store_in = Input(shape=(1,))
shopper_in = Input(shape=(1,))

# Dense N-hot encoding for candidate products
candidates_in = Input(shape=(N_products,))

# Embeddings
prior = Embedding(N_products, 10)(prior_in)
store = Embedding(N_stores, 10)(store_in)
shopper = Embedding(N_shoppers, 10)(shopper_in)

# Reshape and merge all embeddings together
reshape = Reshape(target_shape=(10,))
combined = merge([reshape(prior), reshape(store), resh
                  mode='concat')

# Hidden layers
hidden_1 = Dense(1024, activation='relu')(combined)
hidden_2 = Dense(512, activation='relu')(hidden_1)
hidden_3 = Dense(256, activation='relu')(hidden_2)
hidden_4 = Dense(10, activation='linear')(hidden_3)
```

The final trick to training this model is to use `fit_generator` to limit computing the dense million-length `candidate` input to small batch sizes.

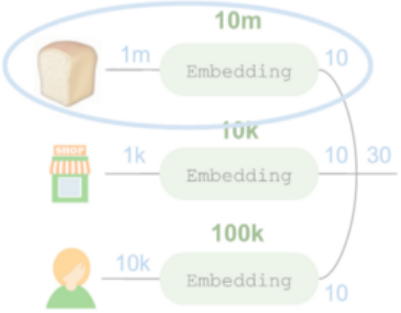
Inspection & Limitations

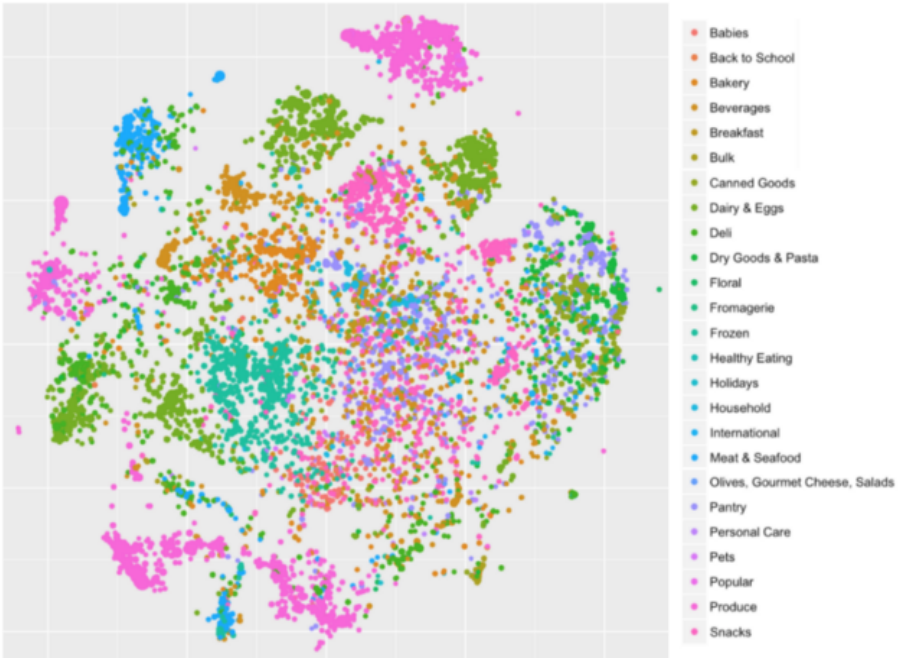
The most interesting part of this network is the product embedding on the left hand side. We can project this embedding down to a 2-dimensional space using t-SNE for dimensionality reduction:





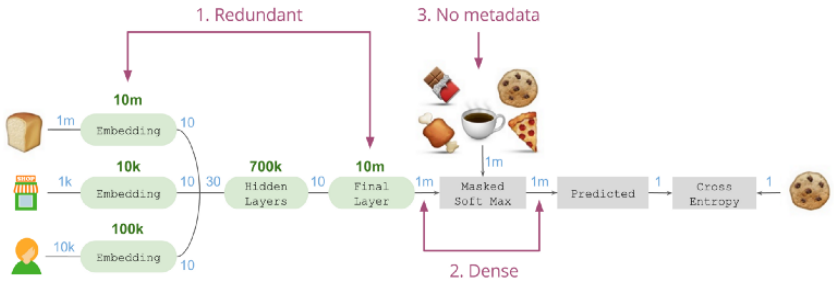
Each circle represents a product, and is sized in proportion to how often it is picked. Clearly the model has learned an interesting structure. We can reveal most of this structure by color-coding every department:





Most of these clusters correspond to departments, even though the department data was never used to learn the embeddings. Further, we can zoom into a region, like the blue meat and seafood department in the upper left. There are other products that appear near the meat and seafood, but aren't meat and seafood. Instead, these are products (like spices, marinades, deli or other items) that are sold at the meat and seafood counter. The model is learning the organization of the store better than the department and aisle encoding data we have.

While this architecture works, it suffers from three deficiencies:



First, we embed products into 10-dimensions on the left hand side, then later implement what essentially amounts to a reversal of this embedding in the final layer to project into the "next product" space. Thus, the product embedding and final layer are trying to learn similar relationships.

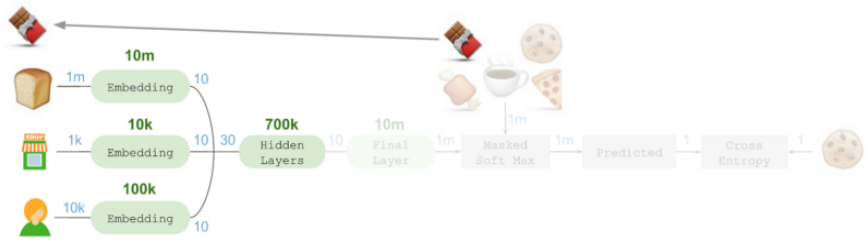
Second, we have a very long and dense 1-million length vector being operated on in the middle, but in this example only 5 of those million values are relevant for the final outcome. This represents a lot of inefficient memory use and wasted computations.

Finally, there is no way to inject additional metadata about the candidate products into this architecture. If we want to learn from the aisle and department we have associated with products, we can feed them into the left hand side of this network based on the last product (bread). But if cookies are in the same aisle, there is no way to represent this information in this architecture. The candidate items are simply binary masks applied to the million length score vector.

Final Architecture

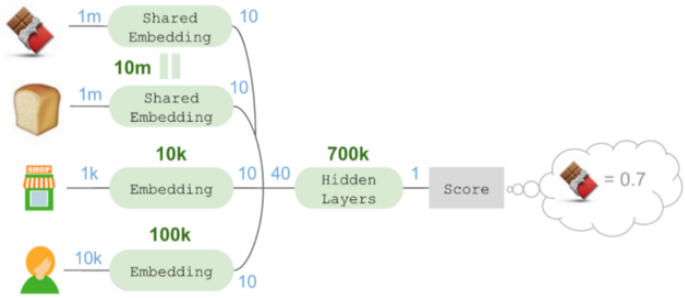
Our final architecture addresses these limitations, and produced significant performance and efficiency gains.

We will create a 'scoring generator' that will be re-used with the `TimeDistributed` layer in Keras for each candidate. To illustrate this, let's take the and move it to the left hand side of the architecture:



Then we can use a Keras shared layer in the functional API to use the product embedding for the to also project the into the same 10-dimensional space:

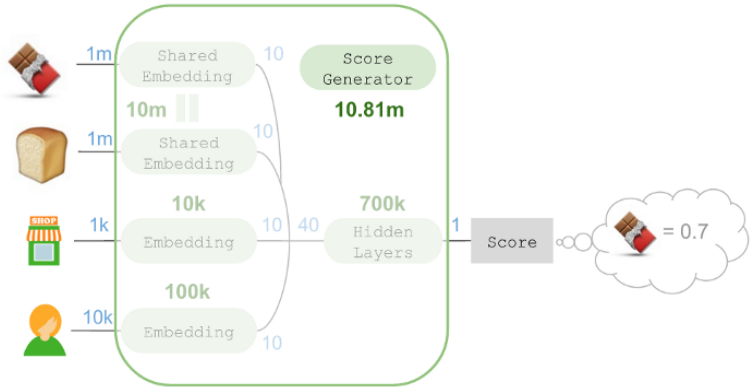
Keep in mind that the number of parameters do not increase when we do this—they are simply re-used for multiple purposes (and can be optimized jointly). We can then use a `Merge` layer to concatenate all of these embeddings together into a 40-dimensional vector, and feed that into hidden layers, which then produce a score:



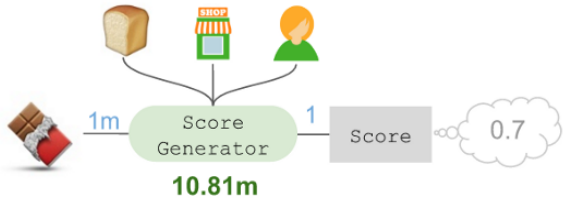
In this case, is given a score of 0.7. We will design the rest of the architecture so that this score is used as a real-valued indicator of how

likely is to be picked next given we just picked . Positive values are more likely to be picked next, and negative values are less likely to be picked next.

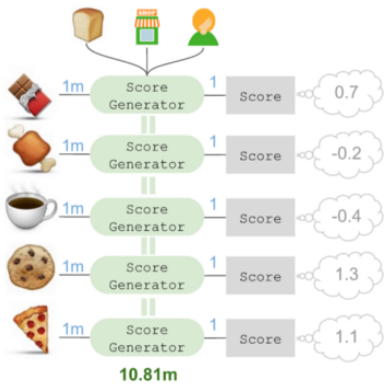
Let's hide the complexity of this component to the architecture under a 'score generator' module:



Which will have in total 10.81 million parameters, and can be simplified as:

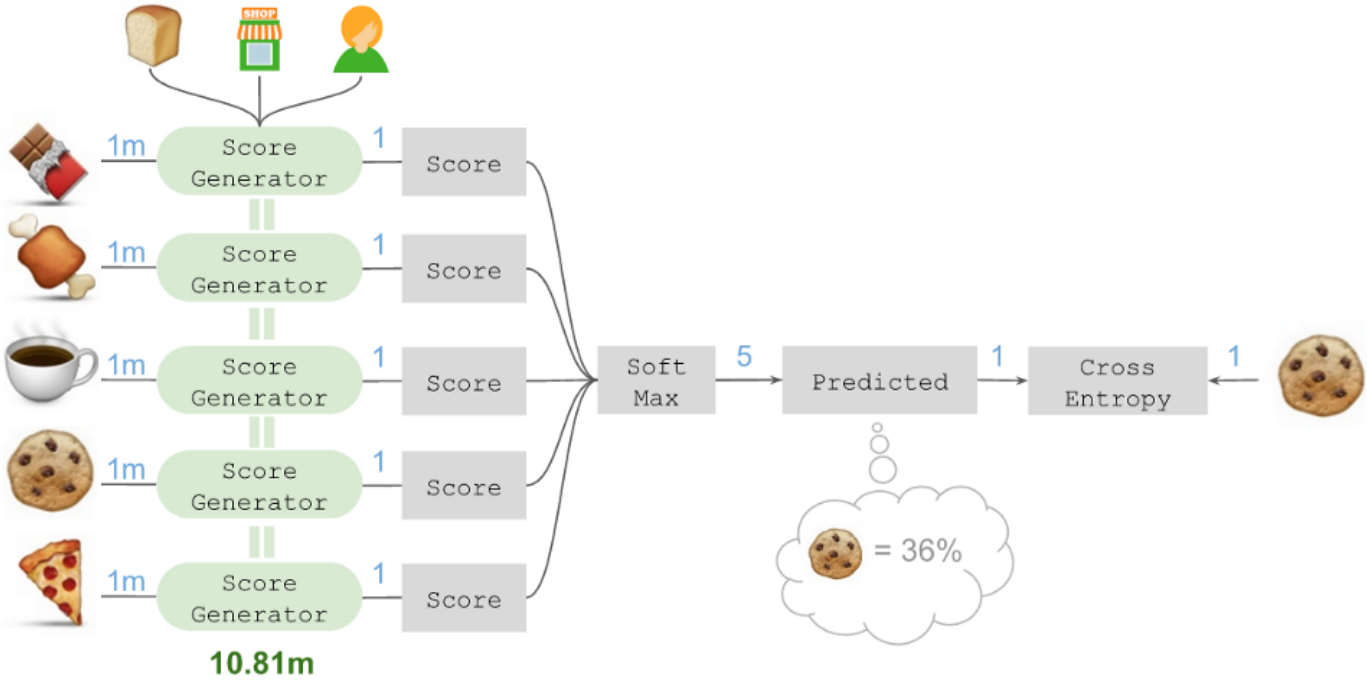


This score generator depends on the candidate ( ), the prior product ( ), the store and shopper. We can replicate this process to score the remaining candidate items:



Suppose we find that the is less likely to be selected next (-0.2), the is least likely to be selected next (-0.4), and the is most likely to be selected next (1.3), followed closely by the (1.1).

All of these scores can be fed into a simple soft-max layer, which produces a prediction that the has a 36% probability of being selected next:



This prediction is then fed into the cross-entropy loss function, along with the indicator that a was, in-fact selected next.

What is key to this architecture is that we can share the score generator over all of the candidates, using the `TimeDistributed` layer in Keras. Thus, we still only use 10.81 million parameters, and TensorFlow can compute the gradient updates for the cross entropy loss jointly over all of the score generators for all of the candidates for a single sequence position.

For now, we leave the Keras code for producing this final architecture as an exercise for the reader .

Final Results

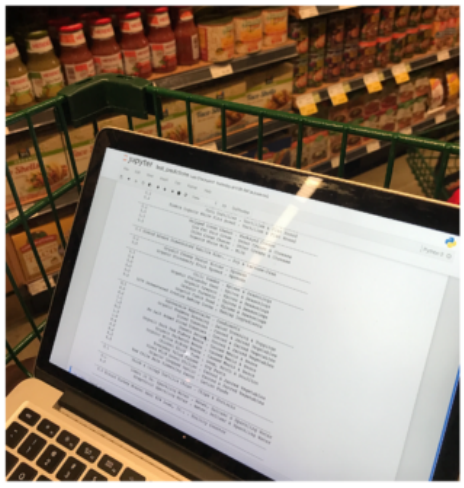
Using this architecture, we achieved the following improvements over the initial architecture:

10x faster training
10% better accuracy
2x slower prediction

The model trains 10x faster because (a) we have cut the number of parameters in half and (b) we don't have the wasteful 1 million length intermediate vectors. We achieved a 10% higher prediction accuracy because we were able to inject aisle and department metadata about both the prior product picked and the candidate product.

Finally, our prediction times were slower by a factor of 2x, from ~50ms to ~100ms for generating the full predicted sequence for a 20-item order (requires 20 calls to `predict`). This was acceptable given our production requirements, where lists need not be sorted in real-time.

We were thrilled with this reduction in training time and accuracy increase despite the reduction in prediction speed!

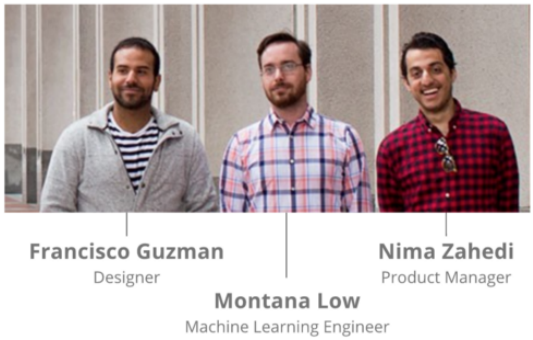


This work has been a very quick and iterative process over the past four months. At times, we went to Whole Foods with a Jupyter notebook running an early iteration of the model sitting in a shopping cart to test the quality of the generated sequences. Nobody even batted an eye—that's the bay area for you!

We are currently testing whether lists that are personalized to the shopper perform better than those based on our fastest shoppers. Next up on our roadmap is to use LSTMs to embed product descriptions

and CNNs to embed product images, to increase generalization performance over rarely shopped for products. We are also planning to test using LSTMs to model the full picking sequence.

A huge thanks to our core team focused on improving our shopping app list sorting:



If you are interested in working on one of the many challenging problems we have at Instacart, check out our careers page at careers.instacart.com.

. . .

This post was significantly improved because of the feedback of many. A special thanks to Greg Brockman, Ilya Sutskever and Andrej Karpathy from OpenAI, and to Daniel Gross of Y Combinator for their suggestions!