#ml-papers June 2019

Applying Deep Learning To Airbnb Search

2 - Model Evolution

NDCG (normalized discounted cumulative gain)

Cumulative Gain (CG) is the predecessor of DCG and does not include the position of a result in the consideration of the usefulness of a result set. The CG at a particular rank position is defined as:

$$\mathrm{CG_p} = \sum_{i=1}^{p} rel_i$$

Discounted Cumulative Gain (CG)

$$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)} \qquad \mathrm{DCG_p} = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

normalized discounted cumulative gain, or nDCG, is computed as:

$$\mathrm{nDCG_p} = \frac{DCG_p}{IDCG_p},$$

where IDCG is ideal discounted cumulative gain,

$$\mathrm{IDCG_p} = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

2 - Model Evolution

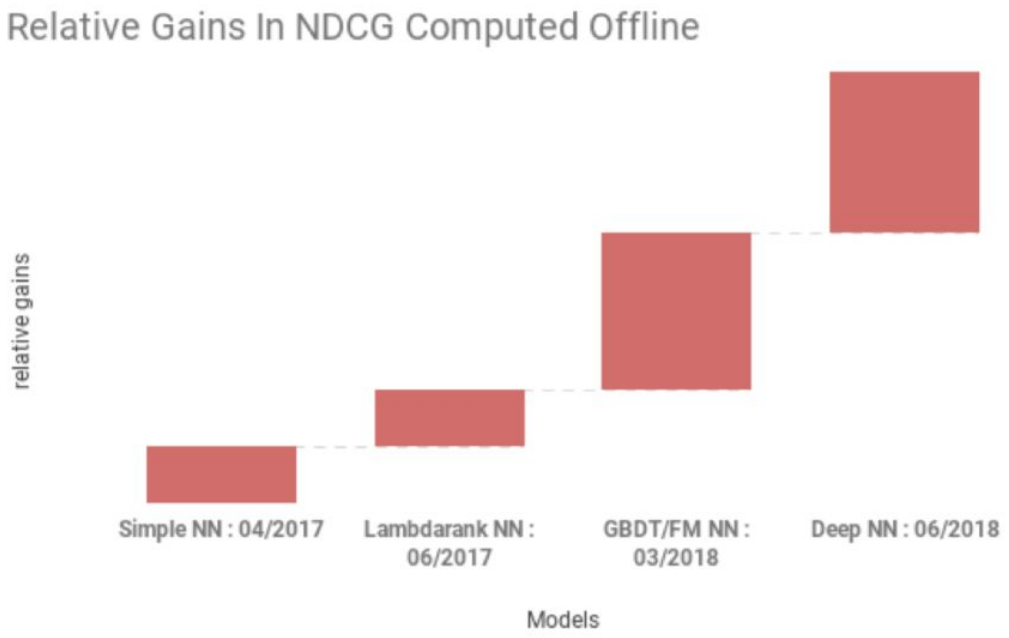
The loss function

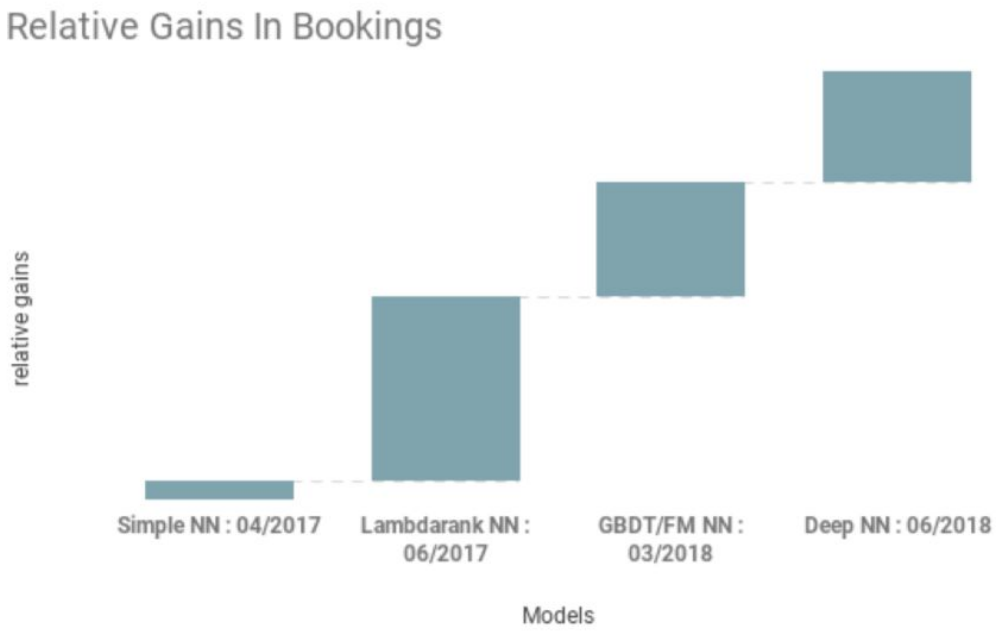
$$J(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N}\sum_{n=1}^{N}\left[y_n \log \hat{y}_n + (1-y_n)\log(1-\hat{y}_n)\right],$$

The true (observed) probabilities can be expressed similarly as $p_{y=1} = y$ and $p_{y=0} = 1 - y$.

$$H(p, q) = -\sum_i p_i \log q_i = -y \log \hat{y} - (1-y)\log(1-\hat{y})$$

2 - Model Evolution





3 - Failed Models

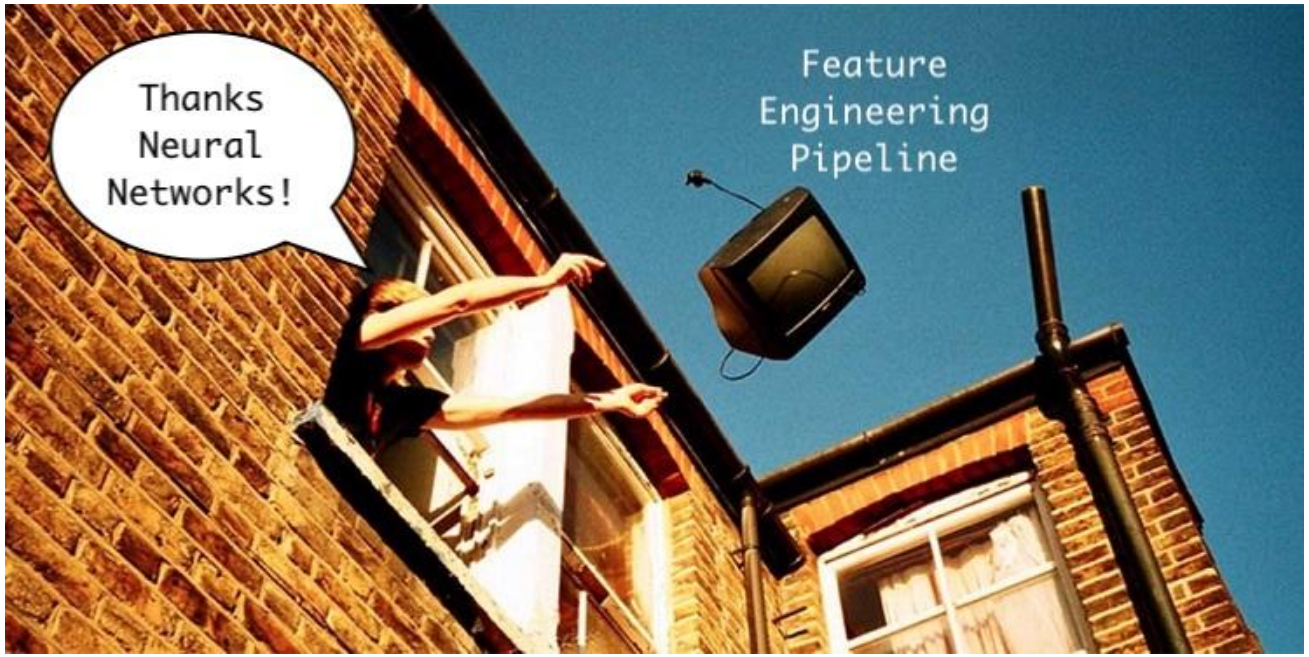
- Predicting long views and bookings as two separate output layers in the neural network. Long views increased, but bookings did not. Predict the outcome you care about directly.

- Using set of listings viewed by a user as the window for training embeddings of listing features. They believed the groupings were representative of the order in which users consumed listing information, but there is no limitation to how often a user can view the listing. Further, they believe they did not have enough data for the training for the embeddings to converge. details

- Extensive feature engineering with the gradient boosted model. Their advances with GBT plateaued; switching to the neural network allowed them to abstract away the feature engineering.

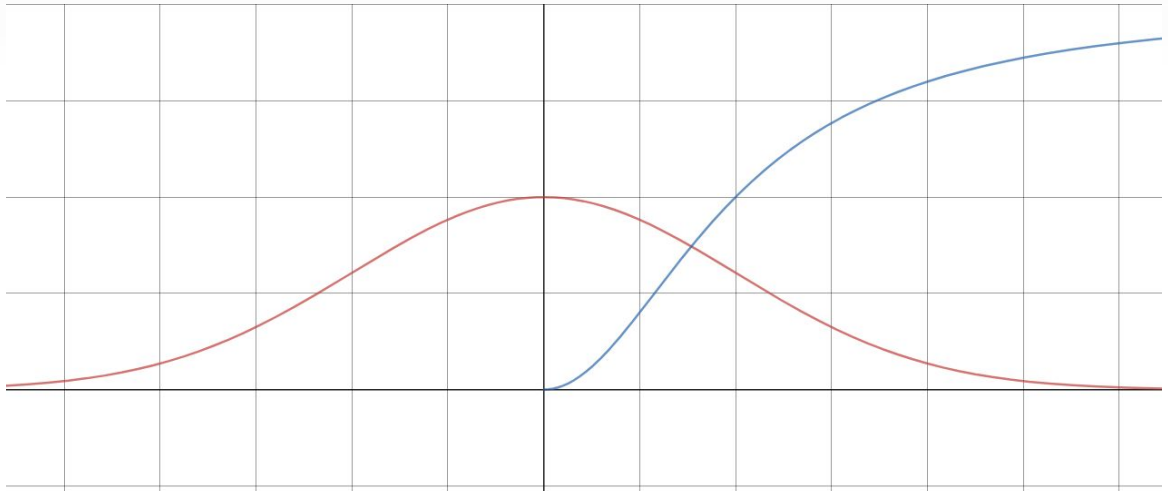
4 - Feature Engineering

- Team has previously implemented extensive feature engineering "tricks" for GBDT pipeline
  - Became hard to validate independently
- Main attraction to Neural Networks: allows for some kind of **feature automation**
  - Feature engineering occurs within the hidden (interaction) layer of the NN
- **But**, not a magic bullet!
  - Needed to shift focus to ensuring features follow certain properties so they play nice with NN
- 3 things to consider:
  - Feature normalization
  - Feature distribution
  - Categorical features with high cardinality



4.1 - Feature Normalization

- **First attempt**: take all features used in GBDT and throw straight into NN
  - Result: Loss saturated in middle of training
  - Reason: Features not properly normalized
  - *But why*? Large feature values => large gradients get back propagated => the vanishing gradient problem
- **Second attempt**: Normalize features using intuition about their distribution shape
  - Result: Fewer training valleys
  - Reason: Avoiding short-term influences to dominate => better learning of long-term dependencies (bias compounding)
  - *But why*? Read more here & here.



If **X** ~ N then apply:

$$f(x) = \frac{x - \mu}{\sigma}$$

If **X** ~ Power then apply:

$$f(x) = log\left(\frac{1 + x}{1 + median}\right)$$

4.2 - Feature Distribution

- Wanted also to ensure **smoothness**
  - To spot bugs upstream
  - Generalization: given smoothness of input values allows for interpolation of interactions given unseen feature values combinations (think of size of combinatorial space!)
  - Completeness
    - Presence of interacting variable
- Sometimes not easily done, e.g. latitude & longitude (right)
  - Engineered to be log(distance from center)
  - Note destruction of information: San Jose != Berkeley
  - *fix*: include categorical neighborhood features

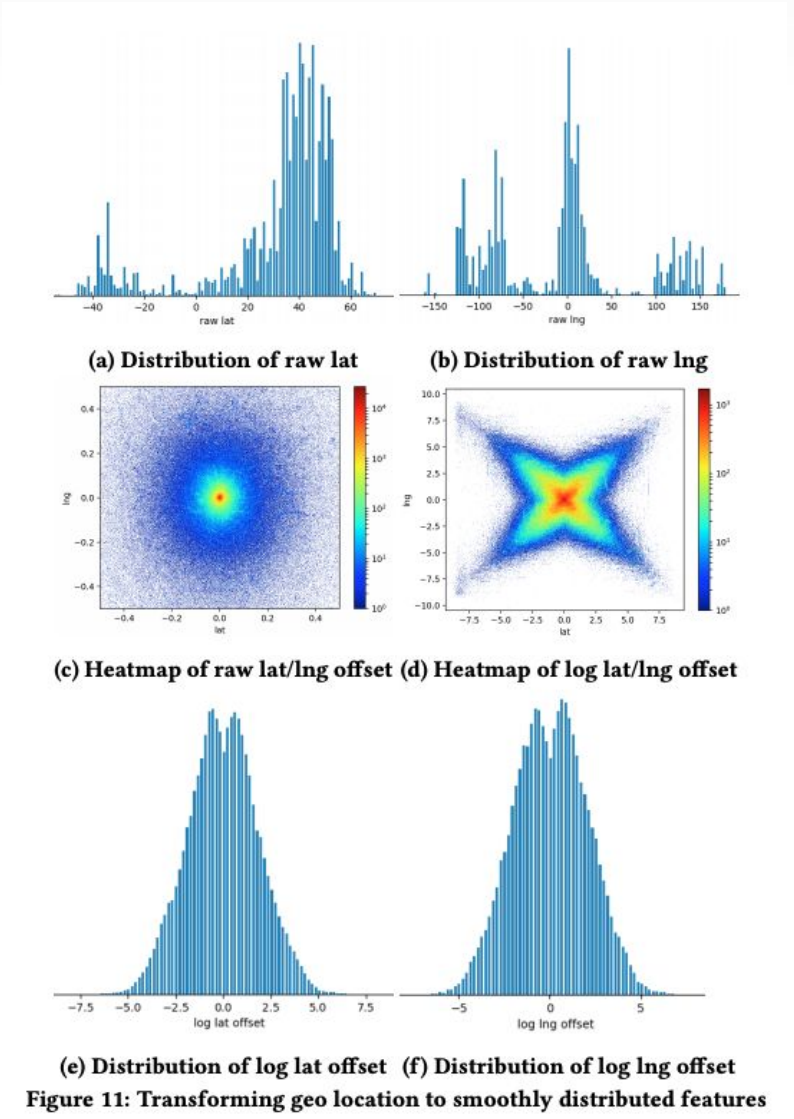


Figure 11: Transforming geo location to smoothly distributed features

4.3 - High Cardinality Categorical Features

- Save time, again, by having NN "do" engineering
- *Example*: guests express preference for specific neighborhoods
  - In GBDT model, information fed by a heavily engineered pipeline tracking hierarchical distribution of books over neighborhoods and cities
    - Lots of work & maintenance!
  - In NN world, create new categorical features that have desired properties (right)

$$f(city, s2\,cell) = \mathbf{hash}\{(city, s2cell)\}$$

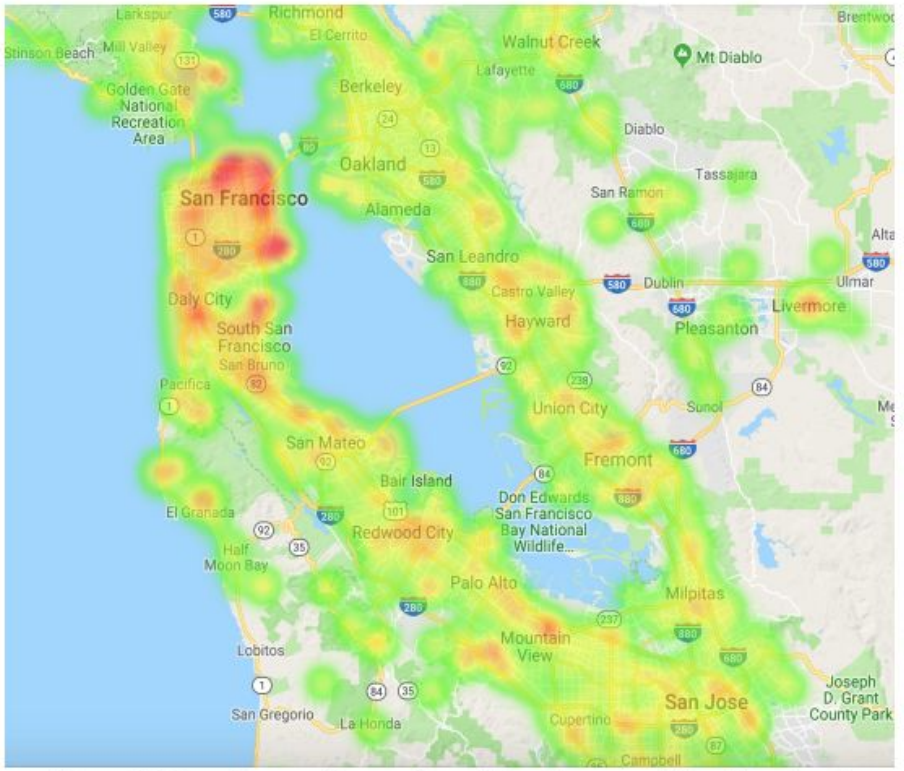


Figure 13: Location preference learnt for the query "San Francisco"

5 - Systems Engineering - Serving

- **Java**: popular cross-platform enterprise language, middle ground between Python and C++ in terms of speed/expressiveness. As far as I know it's not "best in class" for anything, except maybe for finding large quantities of programmers that already know it, wish they included some reasons why they used Java, because they mention having to write their own scoring libraries to support it

- **Thrift** logs: are written by the Java server and used as training data. Thrift is a binary interface definition language (think Parquet without column optimizations & RPCs built in), they don't explain how they're using RPCs but I'm assuming they do.

5 - Systems Engineering - Training

- **Spark**: is used for processing Thrift logs into training data.
- **TensorFlow**: is used for training, Scala/Java are used for offline model evaluation.
- **Protocol Buffers**: are used for the training data, these are 30% smaller binaries than Thrift logs but don't have RPC support, they cite a 17x speedup and improved GPU efficiency over using CSV which seems like an unfair comparison b/c that's not what CSV is designed for.. why not mention a comparison against Thrift or Avro that might actually be relevant?

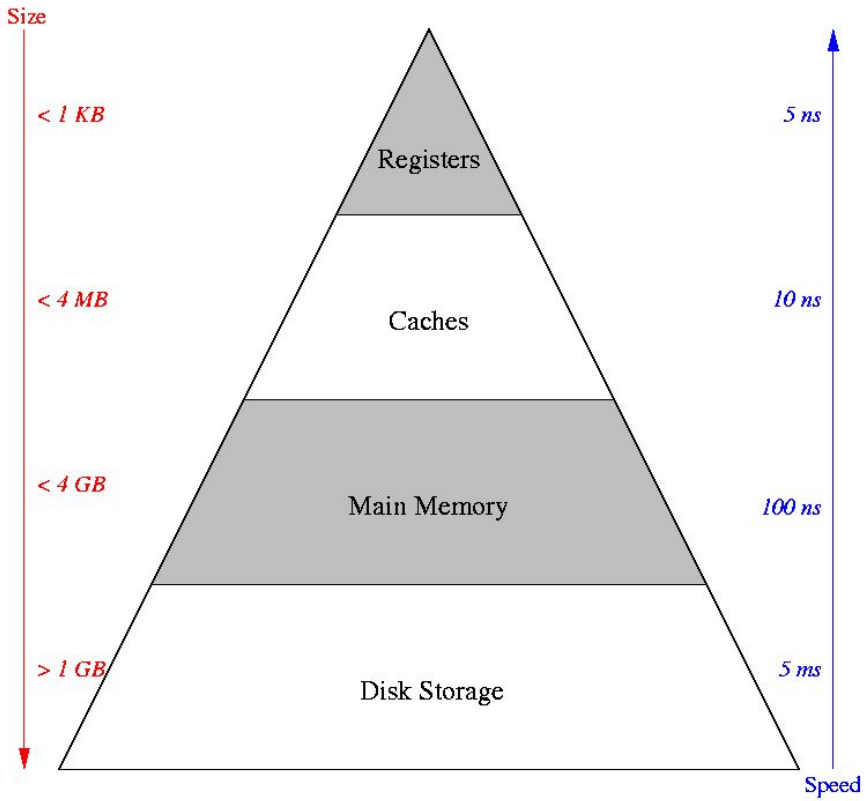
5 - Systems Engineering

Rental unit features like location, number of bedrooms, guest rules are relatively static and are accessed over and over again (ie user 1 looks up AirBnbs in Hell's Kitchen, user 2 does the same search and the same listing features are accessed) during prediction.

To reduce I/O bottleneck of reading these from disk every time, they used listing_ids to reference them and kept the features in GPU memory. Reading from memory is about 100k x faster than disk.

They still need to read the listing_id from disk, but 1 int is much smaller than the size of all their static features (4 bytes vs multiple kb cited in paper, holy crap how wide is this NN??) so this is a big speed improvement in their online serving (where speed actually matters), and it allows them use models that are slow to predict like LambdaRank without sacrificing user experience.

We do something similar in the learn recommendation endpoint, where we cache article similarities in memory to avoid DDB reads when we can.



6 - Hyperparameters

- **Dropout**
  - regularization for neural networks
  - randomly remove neurons on each layer
  - result: degraded offline metrics
  - explanation: randomness produced invalid scenarios, which distracted the model

- **Initialization**
  - initializing all weights & embedding to 0 => worst way
  - Xavier initialization: make variance remain the same for x & y
  - random uniform {-1, 1} for embeddings

- **Learning rate**
  - variant LazyAdamOptimizer: faster with large embedding
  - “lazy”: only update moving average accumulator for sparse features

- **Batch size**
  - Fixed

7 - Feature Importance

- **score decomposition**
  - take final score and decompose it to contributions from each node
  - **however** there's no clear way to separate influence

- **ablation test**
  - remove feature and extract the difference in performance
  - **however** because of feature redundancy, it's impossible to exclude the noise

- **permutation test**
  - randomly permute the value of a feature
  - **however** features are not independent from each other
  - this approach is useful in determining if a feature is pulling its weight at all

- **topbot analysis**
  - use model to rank listings in test set, and compare distribution between high vs low rank

7 - Feature Importance

Figure 14: Comparison of feature distribution for top and bottom ranked listings in test set.

8 - Retrospective

- DL is not a drop in replacement for traditional models like GBDT
- DL is about scaling the system
- For small to medium sized problems, traditional models are at par in performance and easier to handle

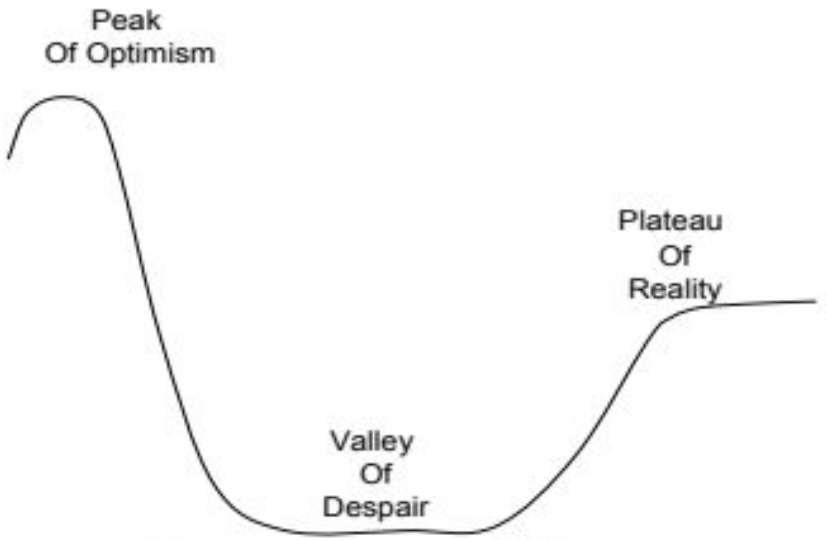


Figure 15: Anatomy of a journey

8 - Retrospective

- Airbnb still recommends DL because...
- Better performance
- Shifts focus from feature engineering to investigating problems at a higher level
- *How can we improve our optimization objective?*
- *Are we accurately representing all of our users?*