Design YouTube's Recommendation System

Q: Do we have to design YouTube's user interface?

A: No, you can assume that the interface is already designed. Your system should simply provide videos to display as recommendations on the home page.

Q: Do we have to design a way to train the model that we'll use?

A: Yes, you should detail how the model that you'll use will be trained.

Q: Do we have to design a way for scientists and ML engineers to do model development?

A: No, that's out of the scope of this question.

Q: Do we need to design the way that our model will be hosted for real-time users?

A: Yes, you'll need to cover how live users will receive recommendations from your model.

Q: Do we need to have our system perform online evaluations such as A/B tests?

A: No, we're only concerned with designing a model—not measuring the model's performance against an existing model.

Q: Do we need to design a data pipeline that transforms raw data into features and labels?

A: No, that system will be provided for you.

Q: Do we have to support model retraining or online learning?

A: No, you can assume that the model doesn't need retraining or online learning.

Q: Can we assume that we have access to a cloud-compute-infrastructure platform such as Google Cloud Platform?

A: Yes, that's fine to use here.

1. Gathering System Requirements

As with any ML design interview question, the first thing that we want to do is to gather requirements; we need to figure out what we need to build and what we don't need to build.

We're designing YouTube's recommendation system.

To accomplish this we'll need to design 3 components:

- Personalized Recommendation Model

- Distributed Training, Tuning, and Evaluation
- Model Hosting

We'll need to rely on systems which we won't touch on or design:

- Cloud Compute and Storage infrastructure (AWS or GCP)
- Retraining or Online Learning
- User Interface
- Model Development
- Online experimentation such as A/B Testing
- The transformation of raw data into useable features and labels

2. Personalized Recommendation Model

The problem we're trying to solve is video recommendations. However, there's typically a surrogate problem that we optimize ML models for. Our surrogate problem here will be trying to predict the next video that a user will watch. Framing the problem this way allows us to use features which include historical interactions a user has had with videos. Labels will then be assigned based on subsequent videos that a user has watched. This way, a model will predict the next video that a user will watch and that will be presented to the user as a recommendation. To do this, we'll need the following:

- Candidate Generators
- A Ranking Model
- Bias Mitigators
- Cold-start Mitigators

The candidate generator is responsible for narrowing millions of possible video recommendations down to hundreds of recommendation candidates. It aims to provide a relevant candidate subset with a high degree of precision (as opposed to recall). This allows us to later use a model to score these candidates with a more detailed representation of users and videos without having to consider millions of videos. This two-stage approach allows us to provide inferences with reasonable latency also allows us to use more than a single candidate generator. The features we use for the candidate generator should include at least the video watch history. Here, we consider a video as watched if the user watched an entire video to completion. To represent the historical watches, we'll use a continuous bag of words (CBOW) embedding. For the input to the CBOW each video will be represented as a one-hot encoding of

the million most popular videos and the embedding layer creates a dense representation of the video. The CBOW will consider the previous 50 video watches as input. Since each of these one-hot encoded videos will produce a dense vector after the embedding layer, we'll average the 50 embeddings together to get a single, fixed-length representation of a user's watch history. If a watched video is not present in the list of the one million most popular videos, that video will be assigned to a zero vector instead of being one-hot encoded. In addition to historical watches, we can include historical search queries the user made. We can use a similar approach to historical watches with another CBOW embedding. Each of the previous 50 search queries will be tokenized into unigrams and bigrams. Bigrams will be selected by how frequently unigrams are adjacent to each other. These tokens will be one-hot encoded among the most frequent one million tokens. If a token a user searched is not in the set of the one million most frequent tokens, a vector of zeros will be used to represent that query. Similarly, the 50 search query embeddings will be averaged together to obtain one fixed-length embedding of the user search history. The label for the model will be the subsequent video watched by the user. Here, that means the label will be the one-hot encoded 51st video the user watched. Since the output of the model is also a one-hot encoded video, it means that we have a multiclass prediction across one million classes. This is called an extreme multiclass classification problem. Calculating these million class probabilities for hundreds of billions of training examples is intractable so we'll have to use candidate sampling. This just means we'll sample a few thousand of the incorrect class outputs as well as the correct class output and only consider the gradient calculation among the sampled classes (instead of all one million classes). Classes will be sampled in proportion to the frequency that the label appears in the training set. As well, we'll need to use importance sampling to account for only updating samples of the class parameters and not all of them. This just means we'll scale the impact each class has on a parameter update in proportion to how much our sampling technique differs from updating the class parameters without sampling. This allows us to speed up training significantly without heavily influencing or some cases, improving the model performance. Architecturally, a maximum of 50 one-hot encoded vectors for each the video and search histories will be fed into separate CBOW embedding layers to produce a 256 dimension embedding. This will feed into fully-connected layer of 2048 units with the ReLU activation function. Each successive layer will halve the number of units until a 256 unit layer remains. This layer represents an embedding for the user. A 256 unit embedding is appended to the network which feeds into a million unit softmax layer which predicts the next video a user will watch. The 256 unit embedding directly preceding the softmax layer represents an unnormalized distribution of the one million video in a 256 dimensional space. This means that the softmax layer just decodes the video embedding into a normalized probability distribution across all one million videos. Once trained, an unseen example is provided to the model. The model will produce a 256 dimension user embedding representing a single user. The dot product of the user embedding and any of the million video

embeddings represents the similarity between the user embedding and the video embedding. Theoretically, we can find the dot product for all one million videos and the user embedding. The videos producing the highest resulting dot products can be used as recommendation candidates. We'll talk more about how to do this practically in the Model Hosting section to avoid one million dot product calculations during inference. The offline evaluation metric we'll use for the candidate generator is the mean average precision for the top k candidates (MAP@k) produced by the test set. We can improve the MAP by rejecting candidates based on explicit feedback such as dislikes or implicit feedback like partial watches.

The ranking model is used to score each of the recommendation candidates returned by the candidate generator. Since the candidate generator only provides a few hundred videos to the ranking model, we can use a richer feature set than the candidate generator and still maintain a reasonable amount of latency and cost per recommendation. Finally, the ranking model is useful in the case that we want to use more than one candidate generator. Some of the ranking features will include the same features used by the candidate generator such as a user's previously watched videos. We can still average the CBOW embeddings for the videos watched (and searches) by the user to summarize the user's preference. We can also include specific features of how long it's been since a user watched a video from a particular channel, how long it's been since a user watched a video about a certain topic, how many videos have been impressed on the user, and finally we can even provide the candidate generators score of the item (in this case the distance from the user/video embedding dot product to the candidate video). For these continuous features, scaling significantly impacts model performance. Using normalization ensures each feature lies between 0 and 1. As well, we can provide transformations of the continuous features to the model including the squared feature and the square root of the feature. It's important to note that features obtained from videos (last video watched, last video shown to the user) all use the same embedding to learn a generalized representation of videos. The same is true for search terms. Since we're using far more features in the ranking model than the candidate generator, we should use a fewer number of embedding units such as 32 instead of 256 to speed up the latency of inferences. In addition, we'll also want to incorporate the impressed videos that a user was shown for that particular training example. This impressed video is also sent through the CBOW layer but instead of averaging the resulting embedding along with the video watch history, it will be concatenated alongside the other features we provide to the ranking model. The label will inform the network whether the user clicked on that impression. Labels will simply be binary based on whether or not a user clicked on the impressed video. This impressed video is included in the features as an input to the model. Architecturally, the ranking model will be similar to the candidate generator such that it will also take on a "tower" deep neural network. The final layer of the network will use a sigmoid function to output the probability of the user clicking on the candidate video.

Each of the candidates will then be sorted based on these click probabilities. A dozen or so videos at the top of this list will be shown to the user as recommendations. The ranker should be evaluated in terms of its recall.

As our model currently stands, it won't account for several biases. We need to make sure that our model mitigates the following:

- Model the freshness of videos
- Discount popular videos
- Limit the positive feedback loop created by the model
- Prevent exploitation of the site structure
- Prevent highly active users from overinfluencing the loss
- Discouraging click-bait

The first bias we should consider is the fact that users favor videos that are new (or fresh). We should add a feature to both the candidate generator as well as the ranking model which represents the days since the video was posted. This allows the model to build an awareness of how fresh a video is. We can even make the feature value negative in the case that an uploader scheduled a post through the UI. This feature will have to be normalized just as the other features are. As well, our model will likely favor more popular videos since most of the features are derived from video watches. If we don't account for some videos being more popular than others, then our model could over exploit popular videos instead of exploring more relevant videos that are less popular. We can mitigate this bias by downsampling videos in the training examples in proportion to their popularity. In line with the trade-off of exploration vs. exploitation, we should be sure to include video watches that weren't a direct result of our recommendation model. We can accomplish this by providing training examples to the network that came from views of videos embedded on other websites. As well, the site structure can be learned and exploited by the model if search queries are sequentially tied to video views, which we don't want. For instance, if a user searches for something then watches the top video of the search results, then the model would learn that a particular search history is associated strongly with a particular video watch. This will result in the model recommending videos which user has already searched for - which is not a meaningful recommendation. Fortunately, our model already mitigates this by having no sequential relationships between search histories and videos views due to separately averaging each of their embeddings. We also need to prevent highly active users from over influencing the loss during training. This naturally happens because the training examples will include far more instances from users who use the platform often. If we ensure that each user has the same number of associated training examples in the training set,

then each user will be represented the same number of times which will prevent the overinflunce of highly active users. Lastly, the ranking model is trained on click through rate (which is whether a user clicked on a recommended video impression). This can lead to biasing the model toward click-bait. Click-bait is defined as a video which is good at attracting clicks (by means of an attractive thumbnail or title) but which doesn't retain the user once they begin watching the video. This often is the result of the video not delivering on the promise impliplied by the title or thumbnail. To reduce the ranking model's bias for favoring click-bait, we can instead weight the loss in terms of how long that particular user watched the video supplied in the input of the model (and therefore impressed on the user). Videos which were impressed on the user but not clicked will receive a unit weight of one to indicate zero watch time for that video. Framing the loss in this way means that the longer a user watched a video, the more influence that example will have on the model. This will correct for the click-bait bias.

3. Cold-start Mitigators

One shortcoming of the current model is its inability to deal with new users or and newly posted videos. This is often referred to as the cold-start problem. To support recommendations for new users, we can add features to both the candidate generator and the ranking model which don't depend on previous watches or search history. Typically what's used is user demographics such as the user's age, device, gender identity, and nationality. We'll use those features here which can be required during signup. For new videos, we can obtain watches as implicit features and dislikes as explicit features by recommending the videos to an uploader's subscribers. This allows the model to incorporate impression and watch features in future training examples and for immediate inference as well.

4. Distributed Training, Tuning, and Evaluation

We need to train, tune, and evaluate models with hundreds of billions of examples. As well, the models will have billions of parameters. To ensure that our system can properly handle models of this caliber, we'll need to implement:

- Data Parallelism
- Model Parallelism
- Checkpointing

We will need to use a distributed training strategy which can also be leveraged for model tuning. We can use asynchronous stochastic gradient descent or a variant optimization technique thereof. We'll use a parameter server architecture here. This means a parameter server cluster will be responsible for maintaining billions of model parameters. Worker nodes will be responsible for fetching parameters from the parameter server as well as obtaining the

location of a relevant subset of training examples from an example queue. The example queue maintains a queue of mini- batches to be trained on. To produce these random mini-batches, we need a randomly shuffled view of the training set can and this can be handled by a dedicated service which we'll refer to as the Training Manager. The worker nodes will train the parameters fetched from the parameter server on the training examples referenced by the polled message from the example queue. This architecture can also be used for distributed model tuning and evaluation. Since we selected an asynchronous distributed training architecture, we can trade off parameter staleness with the bandwidth requirements between the parameter server and worker nodes. For example, the workers can train on several mini-batches retrieved from the example queue before communicating with the parameter server to update the global parameter set.

As we mentioned earlier, the model parameters can't fit on a single machine so we'll need each worker node to actually be a cluster of GPU-equipped machines which partition the model across each of the machine's GPUs. This can include techniques like pipelining model parallelism and model partitioning.

A machine in a worker cluster can become unresponsive at any time so it will benefit us to have some mechanism of model checkpointing. This means that gradient calculations will be stored outside of the clusters. This way, in the case that a worker machine goes down before it can communicate the parameter update with the parameter server, then a new machine can be spun up and the cluster can pick up where it left off in terms of training.

5. Model Hosting

YouTube has over 100MM daily active users each of which can refresh the recommendation home page any number of times. This means we need an efficient to provide roughly one trillion total recommendations per day. This assumes each user on average will look at the recommendation home page consisting of about 10 videos. Each request should be on the order of tens of milliseconds of latency. As well, if a user refreshes the recommendation page, we should ensure that the same videos are not recommended again. The final recommendations will be the result of inferences from two models:

- Candidate Generator
- Ranker

For candidate generation, we'll take advantage of two strategies. The first is that we can use the dot product of the user embedding and video embedding obtained from the candidate generator network. The higher this dot product, the more similarity there is between the video and the user. Effectively, we want to find the k nearest neighboring videos given a user in terms of their dot product. These k nearest neighbors will be the candidates used for ranking. The

problem is that finding the nearest neighbors this way, referred to as maximum inner product search (MIPS) across one million videos can take prohibitively long when serving candidates in real time to users. So, we'll need a way to trade off the accuracy of the nearest neighbors search in favor of a reduced latency. This is commonly referred to as approximate nearest neighbor search. The method we can use here is a variation of locality sensitive hashing called asymmetric locality sensitive hashing (ALSH). The main idea of ALSH is to partition the embeddings by their similarities and then when searching for nearest neighbors, look only at the partition that the input example falls into. This prevents the need to search through all of the embeddings and instead isolates the search to a partition which will contain similar embeddings. There's often a reduction in accuracy compared to an exhaustive search but that tradeoff is accepted because the speed up makes the model usable for real time use. Instead of implementing this from scratch we can use ScaNN which comes from Google. ScaNN is trained by learning partitions for the existing embeddings. To search for the nearest neighbors of a provided input, the embeddings in the top N closest partitions are sent to be scored in terms of an estimated dot product. Finally, the top of those candidates are then rescored by their exact dot products. The top K of those candidates are selected and sent off to be ranked by the ranking model. Here our K can be a few hundred video embeddings. This approximate nearest neighbors search is tuned for a desired MAP@k. We can do that by adjusting the hyperparameters ScaNN. One hyperparameter is the number of partitions to create during training. As the number of partitions grow, the precision improves at the expense of speed. Another influential hyperparameter is the number of partitions to consider for a provided input. For instance, if we only consider the single closest partition to the input embedding, then we only need to evaluate a fraction of the total embeddings. However, since these partitions are not exact representations of locality, it could be the case that other adjacent partitions contain even closer embeddings (and therefore more relevant candidate videos). Finally, the last hyperparameter is how many embeddings we want to rescore based on their estimated distances from the input embedding. If we chose to only rescore 10 embeddings, then the calculation will be far faster than if we choose to rescore the approximate closest 100 embeddings. After tuning, we should be able to achieve a precision of roughly 95% all while supporting a few thousand queries per second on a moderately sized machine with 16 cores and 32 GB of memory. We'd maintain several servers behind a load balancer to manage almost 100MM requests per day. We can ensure the same videos are not recommended when a user refreshes the recommendation page by rejecting candidates based on explicit feedback such as dislikes or implicit feedback like not clicking on a particular video impression.

For ranking, we'll simply have several machines behind a load balancer which will parallely rank video candidates received from the candidate generator. The candidates will be sorted by their scores according to the ranker's inference and the subset of the highest scoring candidates will

be shown to the user in the form of a recommendation on the IU. To speed up inferences, we can purchase machines equipped with enough RAM to fit our entire model and enough cores to run hundreds of inferences in parallel.

6. Bonus #1

One benefit of using a two-stage modelling approach - one for candidate generation and the other for ranking - is that we can use more than one candidate generator. For instance, another way we can generate relevant candidates with high precision is by using something called a co-visitation graph. In this graph, each node represents a video. An edge between two nodes is number of times those videos are viewed within the same user session over the past 24 hours. As well, the edges are normalized by the product of total views of the connected video nodes. To generate candidates, we can select a video node that a user has interacted with either implicitly (watch time) or explicitly (liking or adding to playlist) and finding the adjacent video nodes with the largest edge values. To avoid recommending videos that are too similar to the original video, we can create a spanning tree around the node which at least some number of edges away from the original video node. These candidates can be included alongside the other candidate generator's candidates to be ranked by the ranking model.

7. System Diagram