ISSUES IN RECOMMENDATION SYSTEMS

1. COLD START PROBLEM: (user-item interaction not known)

a) COLD START OF ITEMS:

- We usually know the metadata of items (e.g. color, category, price, etc.) hence find similar items and show them together (content based filtering)
- Learning period: boost the presence of items in recommendations for a certain period of time and decay it over time (randomly or after content based filtering)

b) COLD START OF USERS:

- Don't know anything about user
  - Popular content ~~%~~/in/~~that~~ ~~region~~
  - Trending content
  - Session based recommendation
- Know very little abt user (context, basic profile)
  - Use only context features (geography, ~~age~~ grp, device used, day of week)
    e.g. popular/trending content in that region etc.
  - Fall back Model (FB & Instagram uses this)
    Model based only on context/features basic profile
  - Session based Recommendation (using only profile of user + context)
  - Contextual bandit (Exploration-Exploitation)
    - Takes into account whatever info is available for user and item
    - Selects items to maximize reward
    - Adjusts its strategy based on outcome
    - E.g. ϵ greedy (~~user~~ select ϵ beforehand)
      random number r: $r < \epsilon$ → randomly choose item (exploration)
      $r \geq \epsilon$ → Choose item which has higher prob. of maximizing reward (exploitation)

2. EXPLORATION - EXPLOITATION | DIVERSITY | SERVING BIAS (ranking model) or ALGORITHMIC BIAS | FEEDBACK LOOP ISSUE

- Labels generated from recommendations of RecSys becomes training data for next iteration of RecSys training
    - Feedback loop issue
    - Serving bias
    - Diversity ↓

- To mitigate above issues, we need to have strategy of exploration vs exploitation

1) Multiple Candidate generators
2) ϵ greedy approach to select when to explore and when to exploit
3) Heuristics : Certain % of times in the recommended list use O/P of RecSys and remaining $(100-X)\%$ of times use random items

4) Re-ranking : Penalize items in the final ranked list which are above a certain threshold in similarity to ranked above items

Measuring Diversity: Avg pairwise similarity of an item to all other items ranked above it
low value → high diversity

Theoretically, ranking models should be trained solely on unbiased datasets. However, the exploration dataset may be too small.

3. BIAS IN RECSYS

1. POSITION BIAS: Occurs when items at the top of the list are more likely to be selected by users

MITIGATION:

1) Add positional features:
   Training: Add position of items as feature
   Serving: set all items to position = 1 as feature for ranking (to negate impact of position)

2) Model position bias: Model position and relevancy of item separately and then normalizes using position

$$P_{i,p} = R_i * P_p$$

$P_{i,p}$: Observed prob. of click on item i on position p
R_i: True relevance of item i
P_p: Prob of click on position p

$$\therefore R_i = \frac{P_{i,p}}{P_p}$$

3) Use Interleaving in A/B Tests

2. POPULARITY BIAS: Final Ranked list shown to user has large number of popular items

MITIGATION:

1) Penalize items in training set by their popularity i.e. # of times bought, clicked, etc (sort all items in training set by # of times engaged with and select ~~limited~~ a number of pop. items)
2) Add diversity in training set / candidate generators
3) If classification algo used in ranking phase, penalize the O/P with freq. of item engagement i.e. bought, clicked etc
4) Re-ranking: only keep a certain % of items with popularity above a certain threshold

3. SERVING BIAS – (Mentioned in 2)

4. CLICKBAIT BIAS: Label based only on clicks ✗

MITIGATION: watch length or combination of factors (e.g. likes, comment, etc.)

5. DURATION BIAS: If watched or not watched → label criteria then shorter videos more likely to be watched than longer videos

MITIGATION: watch length in quantile buckets & predict watch quantile

4. NEGATIVE SAMPLING: (More # of -ve than +ve labels available then how to sample -ve labels)

- Popularity based sampling: Sample -ve items based on their popularity where less popular items are preferred over highly popular items.

- Hard -ve sampling: Selects -ve items that are similar to +ve items

- Cluster -ve items based on criteria such as genre, popularity, item creator, etc and pick certain % from each cluster

FRESHNESS: Item age as feature
Re-ranking stage → bump up ranking based on freshness

CANDIDATE GENERATION

SOURCES :

1) Non-personalized sources

- Popular content
- Trending content
- New content
- Trending/Popular content in a geography, age grp or topics followed

2) Personalized ~~network~~ sources:

- In-network content : items generated by users/businesses that the user is currently following/connected to
- Historical content/ content based filtering : Items that are similar to items that the user has liked, previously bought/engaged with or shown interest in. E.g. based on characteristics such as genre, topic, keywords, media, etc.
- Collaborative filtering : Items that are recommended based on the similarity of the user's preferences to those of other users in the system

ALGORITHMS FOR GENERATING CANDIDATES: (can be pre-computed as well)

1. Matrix Factorization
2. 2 Tower Neural Network (either only user features or item features or both)
3. Graph based
4. Neighborhood based (with or without embedding of user or items) interaction matrix only & using Pearson correlation coeff

LIMITING/PRUNING CANDIDATES:

1. Heuristic: Ratio of candidates from each cand. generation method based on business objectives (e.g. more/less diversity, more/less non-personalized sources etc)

2. Thresholding: Limit # of candidates from each cand. gen. method based on thresholding of scoring function from that alg

3. Universal Score: Calibrate all algorithms score and put a Universal threshold

SAMPLING NEGATIVE EXAMPLES IN RETRIEVAL PHASE FOR 2 TOWER

1. Batch Random Negative:

For a query/user an item is clicked (+ve) and rest of the items that are not clicked (-ve)

If sample from these not clicked items randomly : random -ve

However, there is computational cost attached to
- Fetching features for each of -ve items
- Each -ve item has to go thru item tower

For a given batch of queries, if the same random -ve set is used for all +ve examples = Batch Random Negative
|| training speed is much faster

2. Batch Negative:

query 1 → clicked item 1 : +ve example
query 2 → clicked item 2 : -ve example (for query 1)
⋮

- We can use clicked item 2, ... etc as -ve example for query 1 - clicked item 1 +ve example
- Each item embedding in the batch serves once as +ve example and $|B|-1$ times -ve example for other queries in the batch ($|B|$ = size of batch)

Adv: Same advantage as batch random -ve

Disadv: Training data has only clicked items and users mostly interact with small set of popular items leading to bias (not so popular items will be at disadv.)

Items that are not favored by the existing system are less likely to get user feedback. Accordingly, sampling only batch -ves will end up with a model lacking resolution for long-tail items, which seldom appear in training data

3. Hard -ve Mining: (Batch Hard Negatives)
   - To make model better at differentiating b/w easy and hard examples, an iterative approach can be utilized

     Pass 1: Generate -ve candidate/item set by using in-batch random -ve and/or in-batch -ve and training the two-tower encoders (along with +ve candidate/item set)

     Pass 2: Use ANN to get -ve examples/items which are closer to input query (hard -ves) and re-train

NOTE: The dataset need not be balanced at this stage. Can use focal loss or balanced cross-entropy to account for it.

DEEP HASH EMBEDDING

→ Embedding tables or look-up tables that map sparse id features to low-dimensional dense vectors are core component of modern recsys.

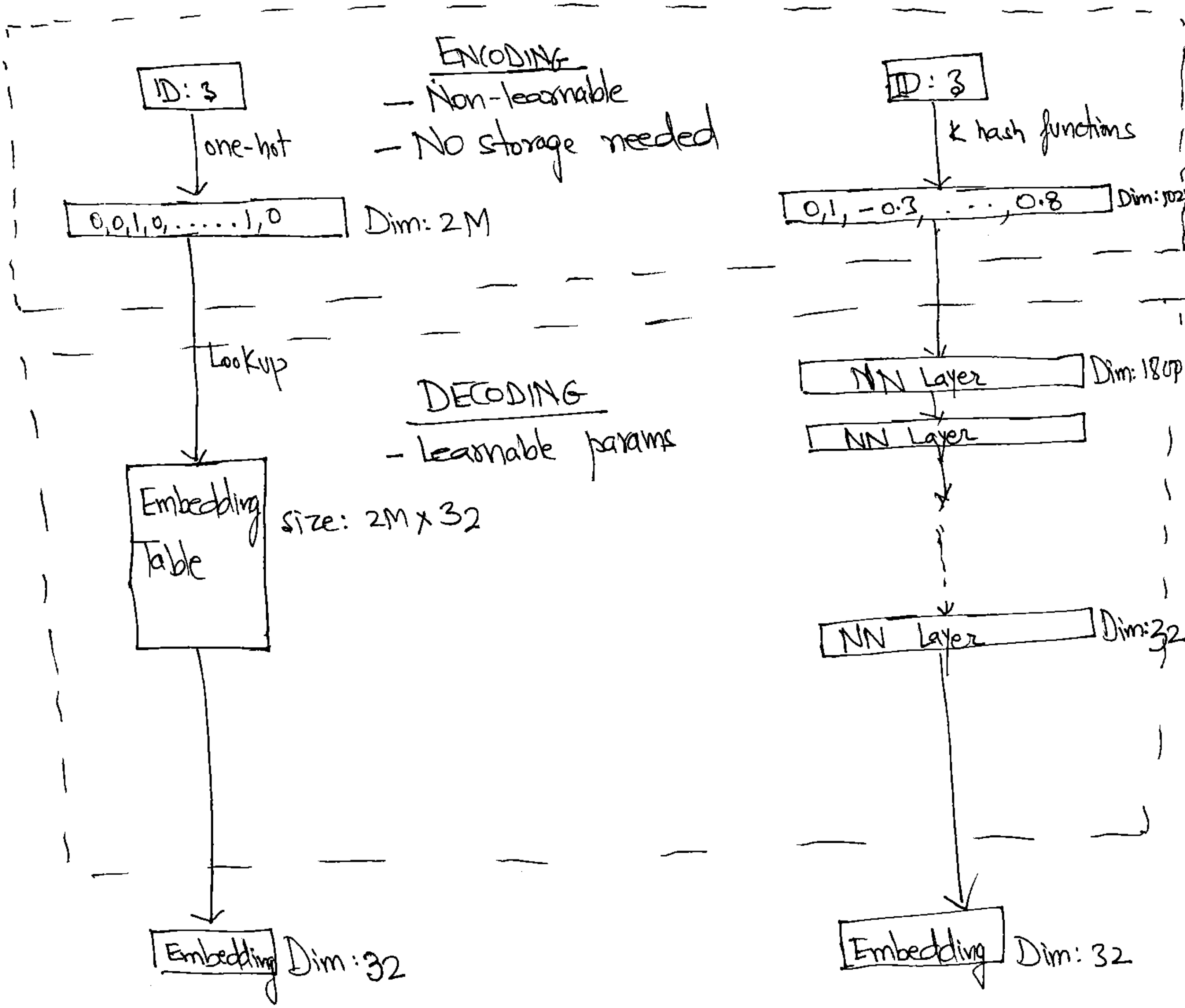
→ We first hash the id into a one-hot hash vector, Use an embedding layer (fully connected layer) during training and convert into low-dimensional dense vector

→ At serving time, this embedding layer can be used as a lookup table i.e. embedding table for sparse ids.

→ By design, embedding tables are memory-hungry
e.g. A feature with 5000 sparse ids converted to 128 dimension vector is of size (5000, 128)

→ For fast inference, these embedding tables reside in primary memory (RAM, L1/L2 cache or SSDs).
CPU memory

→ The key idea behind Deep Hash Embedding (DHE) is to replace embedding tables with an encoder-decoder architecture that has a much smaller memory footprint.


ENCODING
- Non-learnable
- No storage needed
ID: 3
one-hot
0,0,1,0,.....1,0
Dim: 2M
Lookup
DECODING
- Learnable params
Embedding Table
size: 2M x 32
Embedding
Dim: 32
ID: 3
k hash functions
0,1, -0.3, . . . , 0.8
NN Layer
NN Layer
NN Layer
Dim: 32
Embedding
Dim: 32

One-hot

DHE

→ Encoding Step : Input id ⟶ apply K hash fns → K-dimensional dense vector normalized in [-1, 1]

Decoding Step : K-dimensional dense vector normalized in range [-1, 1] → small NN (maps) → semantically meaningful embeddings

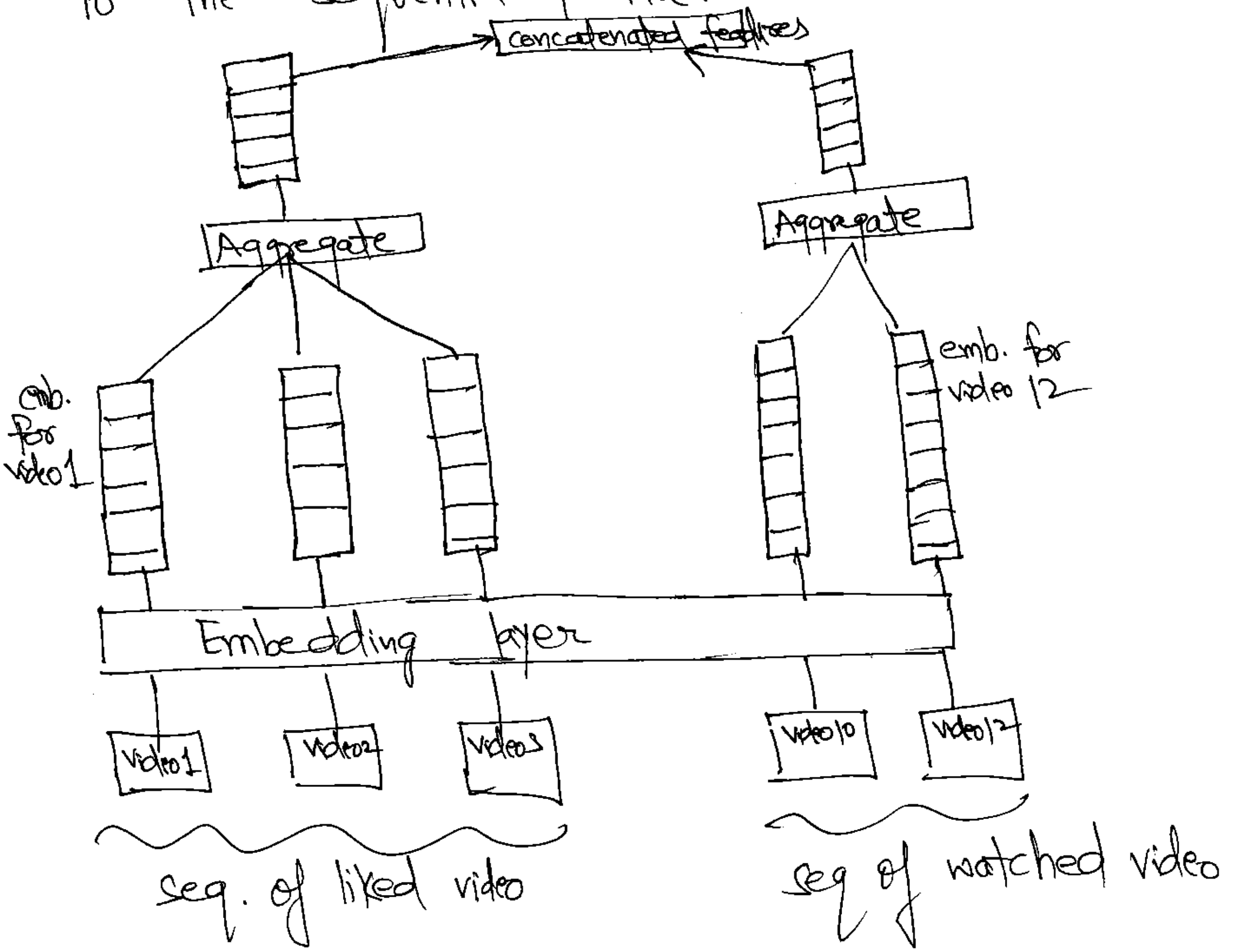
→ Advantages : 1) No hash collisions (due to large no. of hash fns e.g. 1024) equal to no. of dimensions

2) Low memory footprint since no storage required for embedding table

WAYS TO ENCODE SEQUENCE FEATURES

1. Seq. features e.g. list of liked videos, list of watched videos.

Each item id is a sparse feature encoded via an embedding layer i.e. o/p of embedding layer is the dense, low-dimensional representation of item Id. Now for a sequence of item ids, they are fed as separate feature into embedding layer and then aggregated (avg.) to get embedding vector corresponding to the sequential feature.



2. If 2 tower is used in Cand. generation then we have embeddings corresponding to each item in the item seq. → aggregate them (avg.)

3. Use an attention block and the o/p of attention block (K,V) → concatenate them to get one feature vector. (E.g. as used in MMoE network)

QUANTIZATION AND INDEXING FOR ANN

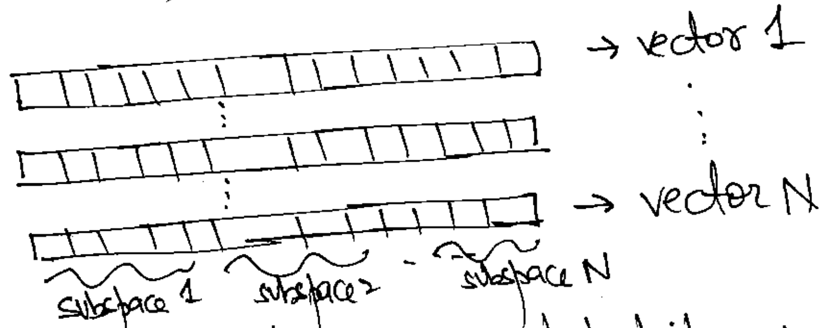
→ If the no. of items or users is huge for a use case then **Storage** and **search** for vector embeddings is problematic and needs scaling

→ STORAGE SOLVE: 1) Use less no. of bits to store embeddings (e.g. float16 etc.)
2) Product Quantization (PQ)

→ SEARCH SOLVE : INDEXES
1) Inverted Index
2) HNSW (Hierarchical Navigable Small World)

combination yields highly scalable search (e.g. > billion item search)

PRODUCT QUANTIZATION:



→ Divide each vector into smaller segments/subspaces

→ Run K-means clustering on each subspace (e.g. $K=256$ centroids)

→ For each subspace/segment of vector, find the nearest centroid (e.g. centroid 240) and substitute it with that centroid Id

→ Example: if 256 centroids then 8 bits can represent the indices of these 256 centroids ($2^8 = 256$)

→ Essentially, we have encoded our original vectors with Centroid Ids (i.e. grp of centroid ids of vector = PQ code for the vector)

INVERTED INDEX: → Run K-mean clustering on PQ codes of vectors & the centroids so obtained are indexes/partition centroid

stored in memory

stored in database

→ [ ] → List of PQ codes for vectors

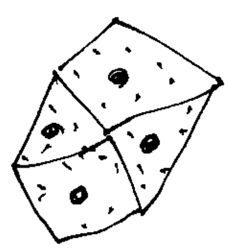
[ ] → List of PQ codes for vectors

⋮

[ ] → List of PQ codes for vectors

Partition Centroids/Inverted Index

→ This structure is also referred to as "Voronoi Cell"



• Partition Centroid

∴ ~~PQ~~ code of vectors (usually in database)

2 imp. params:

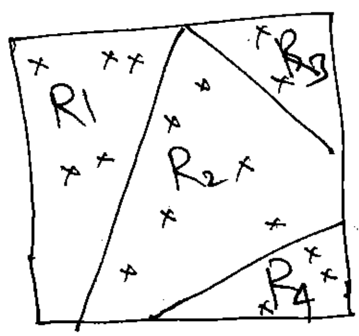
i) no. of cells to create (K in K-means): if it is high, more cells are created but fewer vectors to search within each cell → prioritizes search-speed

ii) no. of cells to search: if it is high, more time needed for search but better quality, especially for edge query vectors → prioritizes search quality

APPROXIMATE NEAREST NEIGHBOR (ANN)

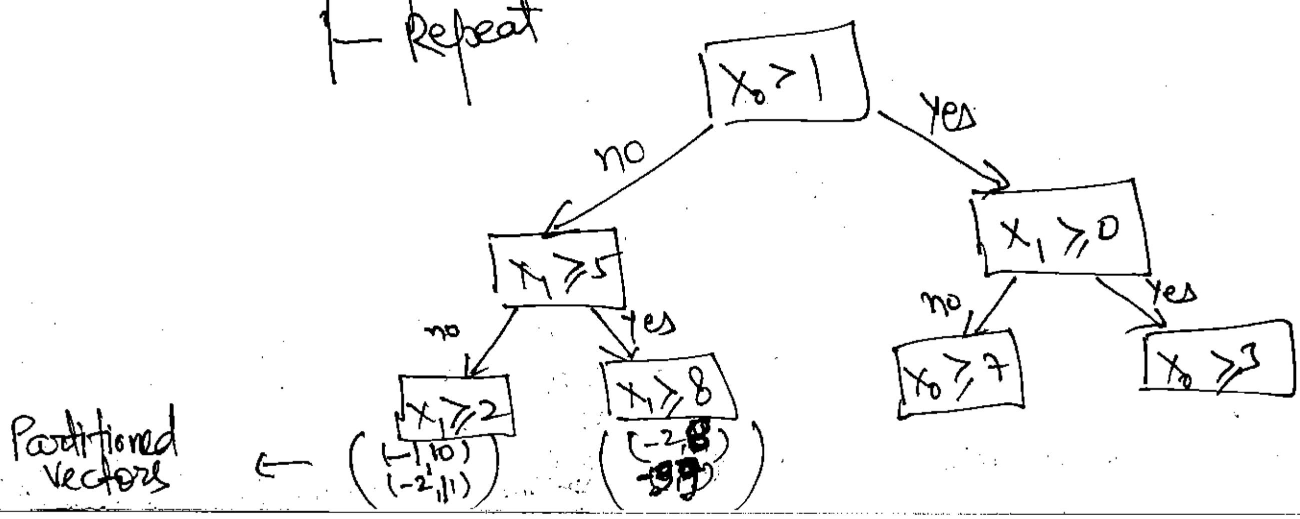
1. Tree based ANN (ANNOY - SPOTIFY)

- Split the space into multiple partitions using tree structure & leverage characteristics of tree to perform faster search
- Non-leaf nodes split the space into two partitions given the criterion
- Leaf nodes represent a particular region (partition) in space
- To find nearest neighbors, the algo only searches the partition the query point belongs to



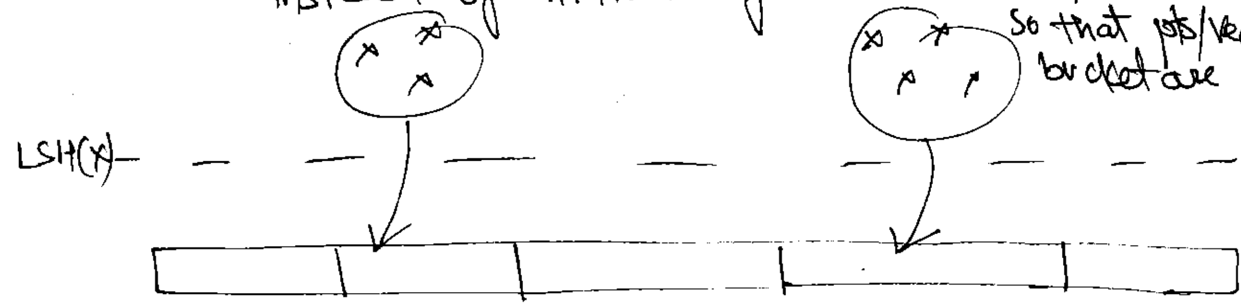
Example: K-d tree, Annoy

- Pick a random dimension from your K-dim vector
- Find the median of that dimension across all of the vectors in your current collection
- Split the vector on the median value
- Repeat



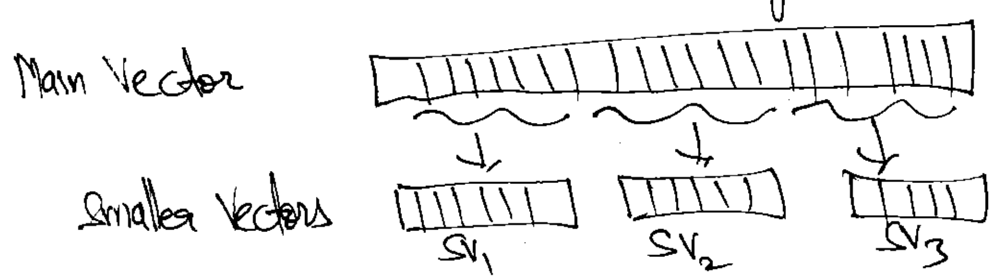
2. Locality Sensitive Hashing (LSH) → FAISS (Facebook)

- Uses hash fn to bucket pts
- Pts in the same bucket are closer to each other than pts in another bucket
- LSH only searches pts belonging to the same bucket as query pt.
- Instead of minimizing hash collisions, "maximize hash collisions" so that pts/vectors in same bucket are closer



3. Quantization based → SCANN (Google) / FAISS (facebook)

- Represent pts closer to each other by a "representative" vector
- Only performing KNN on all pts and representing pts belonging to the cluster by its centroid → accuracy ↓
- Instead divide the vector into smaller vectors and find centroids for each of the smaller vectors and represent the vector by a combination of the centroids of these smaller vectors



Find K centroids/clusters of all $SV_1 = SK_{11}\ SK_{12}\ SK_{13}$ (3 centroids)

$SV_2 = SK_{21}\ SK_{22}\ SK_{23}\ SK_{24}$ (4 cent...)

$SV_3 = SK_{31}\ SK_{32}$

Main vector $\boxed{\text{3 values possible}}$ $\boxed{\text{4 values possible}}$ $\boxed{\text{2 values possible}}$ $= 3 \times 4 \times 2 = 24$ ce...

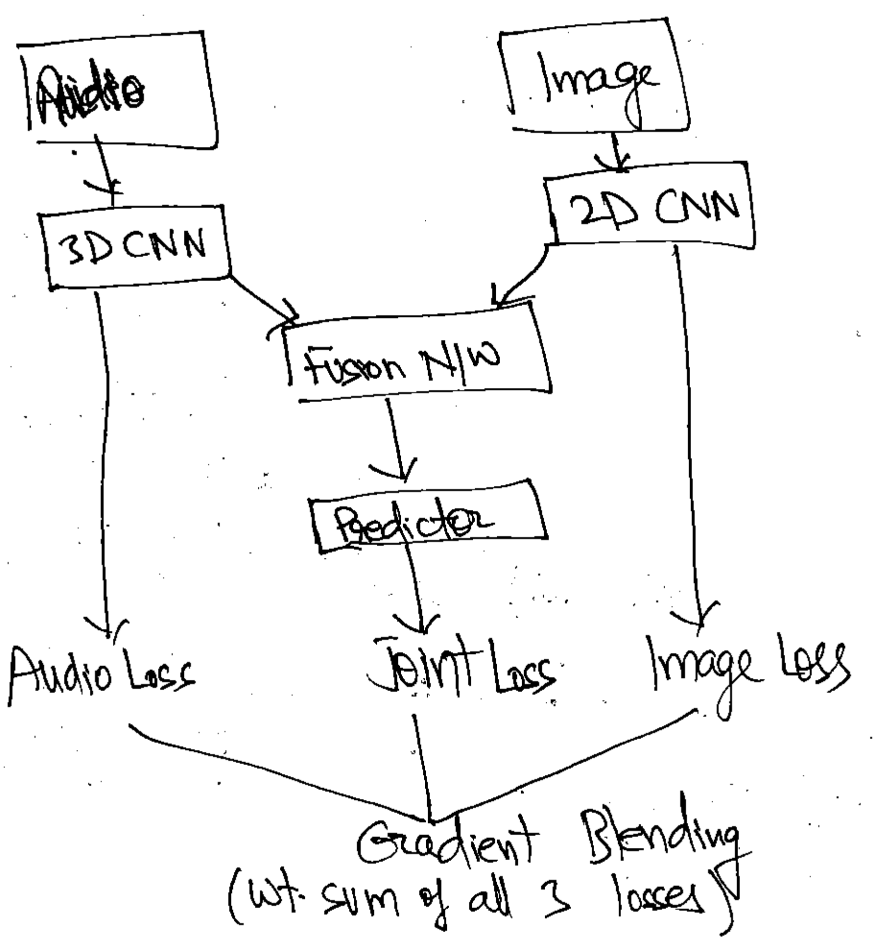
= comb. of 24 cent to represent each...

CHALLENGE IN TRAINING MULTI-MODAL MLSYSTEMS:

- Over-fitting (when the learning speed varies across different modalities, one modality (e.g. image) can dominate the learning process

- Techniques to overcome overfitting in multi-modal settings
  - Gradient Blending
  - Focal Loss

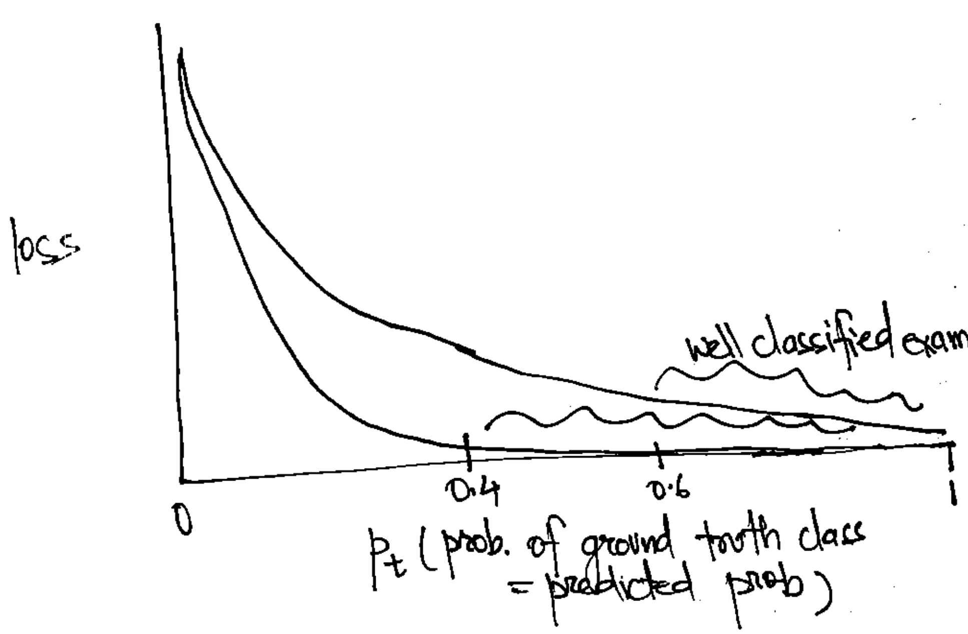
1) Gradient Blending: Adjusting loss function to take into account each model's individual wts as well as combined wts.



2) Focal Loss: (Better than Cross Entropy Loss)

- Cases where Cross-entropy loss performs badly:
  1) Class imbalance :- Majority class examples will dominate the loss function and gradient descent, causing wts to be updated in the direction of the model becoming more confident in predicting majority class while putting less emphasis on minority class.
     - Balanced Cross-Entropy solves above problem by adding a wt. factor(α) to each class (could be inverse class freq. or a hyper-parameter determined by cross-validation) : Same is done by focal loss

  2) Fails to distinguish between hard and easy examples
     - Hard examples are those in which the model repeatedly makes huge error whereas easy examples are those which are easily classified



For ground truth value=

- $CE = -p_t$
- $FL = -(1-p_t)^{\gamma} \log p_t$
  where γ = modulating facto

→ when $\gamma = 0$, $FL = CE$

→ Loss is low when p_t [0.4, 1]
↓
area of well classified easy exam
||
even examples that are easil classified ($p \gg 0.5$), incu a loss with non-trivial mag

→ Now, as we increase the value of γ, we slowly extend that range of predicted prob. where loss is low to $\sim [0.4, 1]$

$\Downarrow$

This means we are "extending" or "relaxing" our criteria of well classified examples

→ This means with standard CE loss, the model will push scores even for well-classified examples further & further away till the predicted prob reaches 1

Whereas

the model trained with FL will not care much abt well classified examples (will not push their prob to 1), instead will focus on improving / reducing loss for hard examples (whose error is higher)

wt. factor, modulating factor

$$\text{Hence } FL(p) = \begin{cases} -\alpha (1-p)^{\gamma} \log p & ; y = 1 \\ -(1-\alpha)\, p^{\gamma} \log(1-p) & ; \text{otherwise} \end{cases}$$

MULTIPLE OBJECTIVE CLASSIFICATION SYSTEM

→ Example:

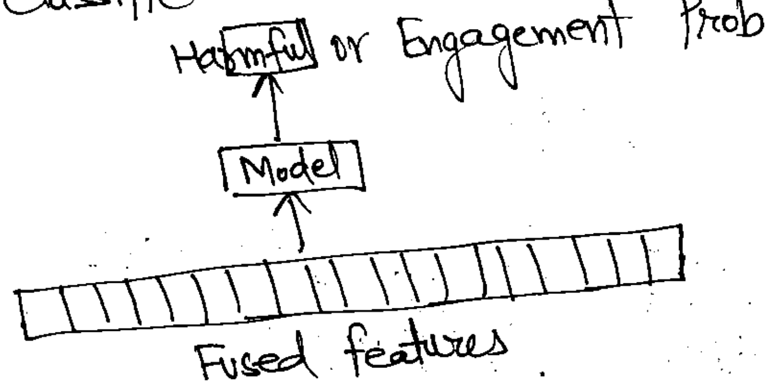
1) Detect harmful content
   - Nudity
   - Violence
   - Hate

   ⋮

2) Increase Engagement
   - Likes
   - Comment
   - share
   - Click
   - Dwell time

→ OPTIONS:

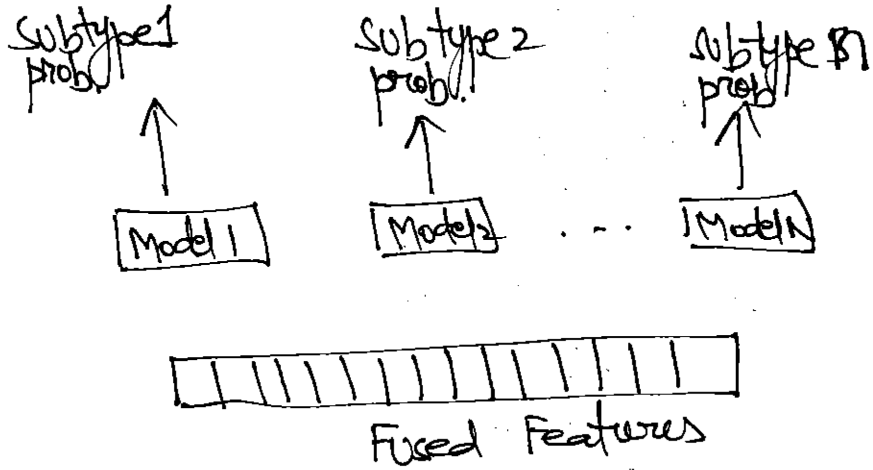
1) Single binary classifier:



Issues: Difficult to determine which subtype of harm or engagement

- Cannot inform users or business which subtype resulted in the outcome
- Cannot improve performance in a subtype, if the system is bad in some subtypes

2.) One binary classifier per subtype

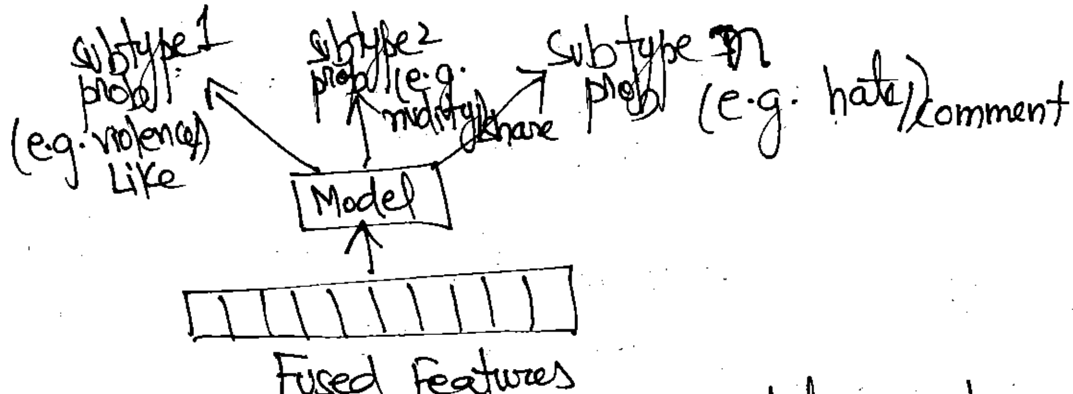


Adv: 1) Can explain the outcome in terms of subtype e.g. why a post was taken down

2) Can improve the model for a particular subtype, if the subtype model performance is poor

Disad: 1) Training & maintaining n models is time consuming & expensive

3) Multi label Classifier



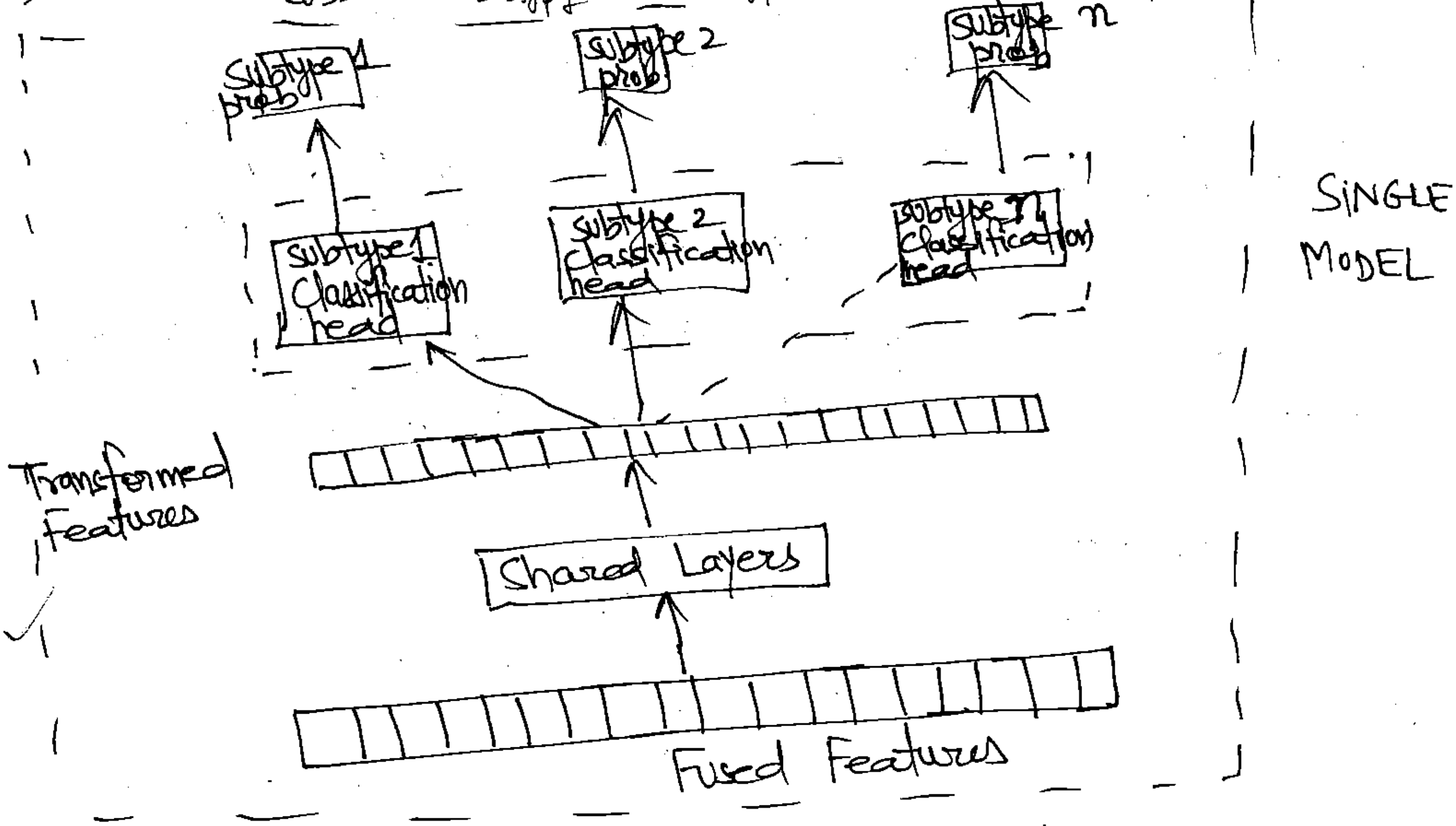
Adv: 1) Training & Maintaining the model is less costly

Disadv: 1) Each observation in training data have to be labeled for all subtypes

2) If diff. subtypes require diff. feature transformations, not possible

4) MULTI TASK CLASSIFIER

$Loss = Loss_{subtype1} + Loss_{subtype2} + \dots Loss_{subtype\,n}$



Shared Layers: Common across all subtypes

Task Specific Layers: Set of independent ML layers (Classification head)
Each classification head transforms features in a way that is optimal for predicting a specific subtype (e.g. harm) probability

Adv: 1) Not expensive to train or maintain since it is single model

2) Shared layers transform features in a way that is beneficial for each task (prevents redundant feature transformations)(+ decreases training time compared to separate layers)

3) Each obs. in training data "need" not be labeled for all subtypes (e.g. weather and animal type prediction, but needed in post → comments, like, sets, view) and training data of each subtype contributes to the learning of other subtypes (especially useful if one subtype has less training data)

~~NOTE ON MULTI-TASK CLASSIFICATION:~~

~~- Usually employed for scenarios where multi-labels (or binary) for different tasks (usually related tasks) on the same input~~
~~E.g. predict type of animal and weather on a given pic~~
~~cat dog rat snake~~ ~~cloudy sunny raining~~
~~dataset 1~~ ~~dataset 2~~
~~Concatenate both dataset (1&2) to get full dataset for training~~

~~- For feed-based systems: Diff. tasks~~ ~~Like Comment Share~~ ~~on the same post~~

HOW TO COMPILE TRAINING DATASET FOR MULTI-TASK LEARNING

- Usually employed in scenarios where multiple labels for multiple tasks on the same input. E.g. predict animal and weather on an image
  - animal: cat, rat, bat, ..., lion
  - weather: cloudy, sunny, rain

- Training Dataset:

① — Animal Dataset (or letter) + (concatenated) Weather Dataset (or alphabet)

(note: animal dataset images should not show weather and vice-versa) but at prediction time, prediction needs to happen for both tasks

———— Animal Dataset example 1
———— Animal Dataset example 2
———— Weather Dataset example 1
———— Weather Dataset example 2

② Feed based systems: Like, Comment, Share, etc. on a post

	Like	Comment	Share
Post #1	1	1	0
Post #2	0	0	1

(similar to multi-label dataset)

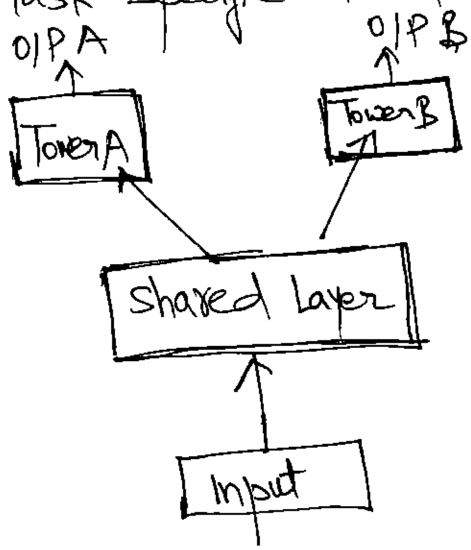
(Can have multiple reactions on a post)

MULTI-TASK LEARNING (AND MULTI-TOWER)

SHARED BOTTOM MODEL:

→ Same/shared base

→ Task specific head



→ Issue: If the tasks are not co-related then the performance is sub-par as the shared layer with common representation introduces noise

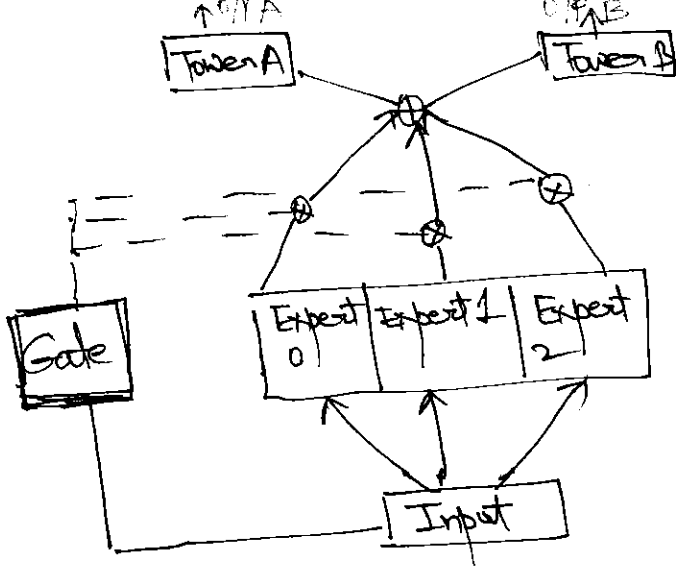
MIXTURE OF EXPERTS (MoE):

→ Solves the issue ~~of~~ shared bottom model i.e. performs better even if the tasks are not co-related

→ Multiple expert n/w + a gating mechanism to weight each expert's O/P

→ Each expert in the n/w is able to learn different patterns in data and focus on diff. things

→ The gating n/w acts as a weighting scheme and takes the wt. avg. of expert n/w conditioned on input data i.e. the model is able to activate parts of n/w differently on a per sample basis!



MoE

MULTI-GATE MIXTURE OF EXPERTS (MMoE):

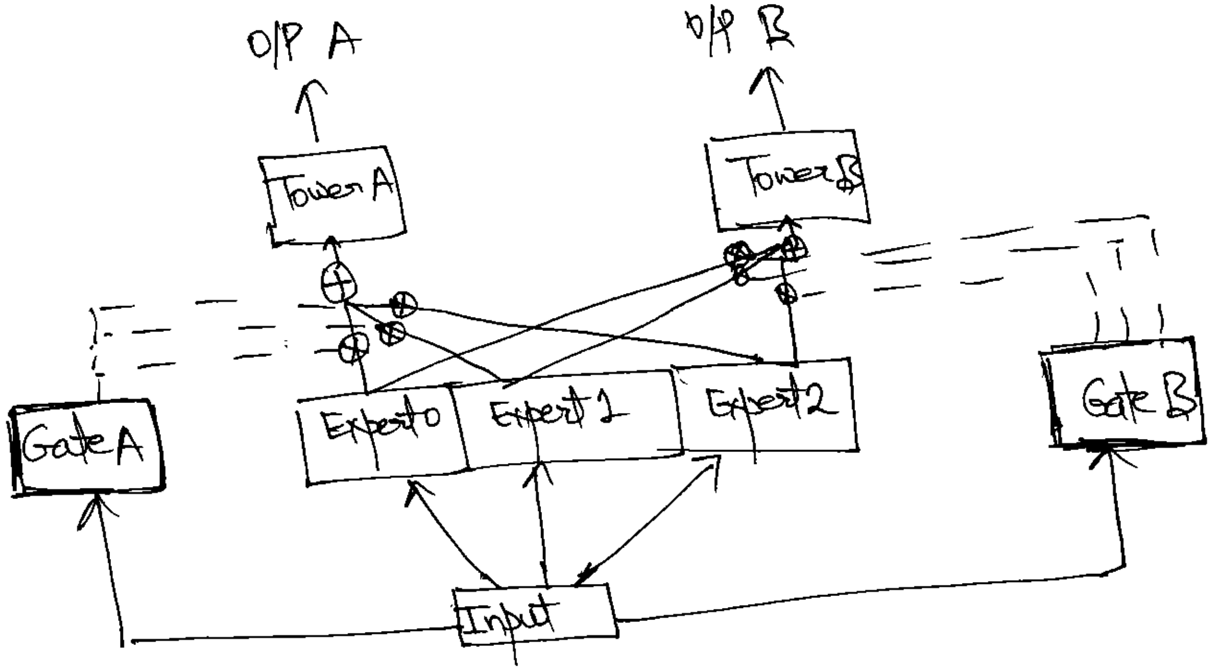
→ Performs even if the tasks are not correlated and better than MoE

→ MMoE is diff. than MoE : MMoE has a gating n/w for each task
adaptation of MoE to multi-task learning

→ This allows the model to learn per task and per sample weighting of each expert n/w (instead of just per sample weighting like MoE)

→ MMoE essentially learns to model relationships b/w different tasks i.e. tasks which have little in common with each other will result in the gating n/w of each task learning to use diff. expert n/ws.

→ Experts are shared across all tasks and the gating n/w is task-specific. Each task also has a task-specific "tower" to decouple the optimization for tasks



MMoE

Note: Expert N/w is any DNN architecture

→ TRAINING DATASET FOR MULTI-TASK LEARNING:

INPUT features						TARGET			
User features			Context features	Item features		Binary Classification			Regression
sparse			dense	sparse					
User_id	gender	age		Item_id	category	click	follow	like	watch_Time

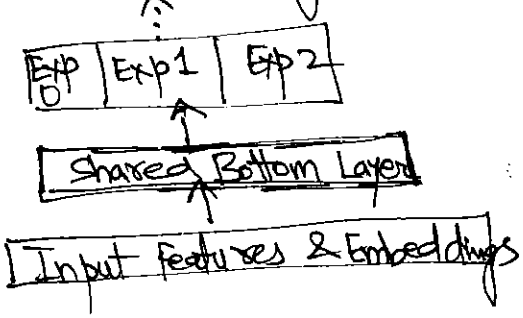
→ Cross-features b/w user-item not needed

→ User features (dense + sparse) ✓
Item features (dense + sparse) ✓
Context features ✓

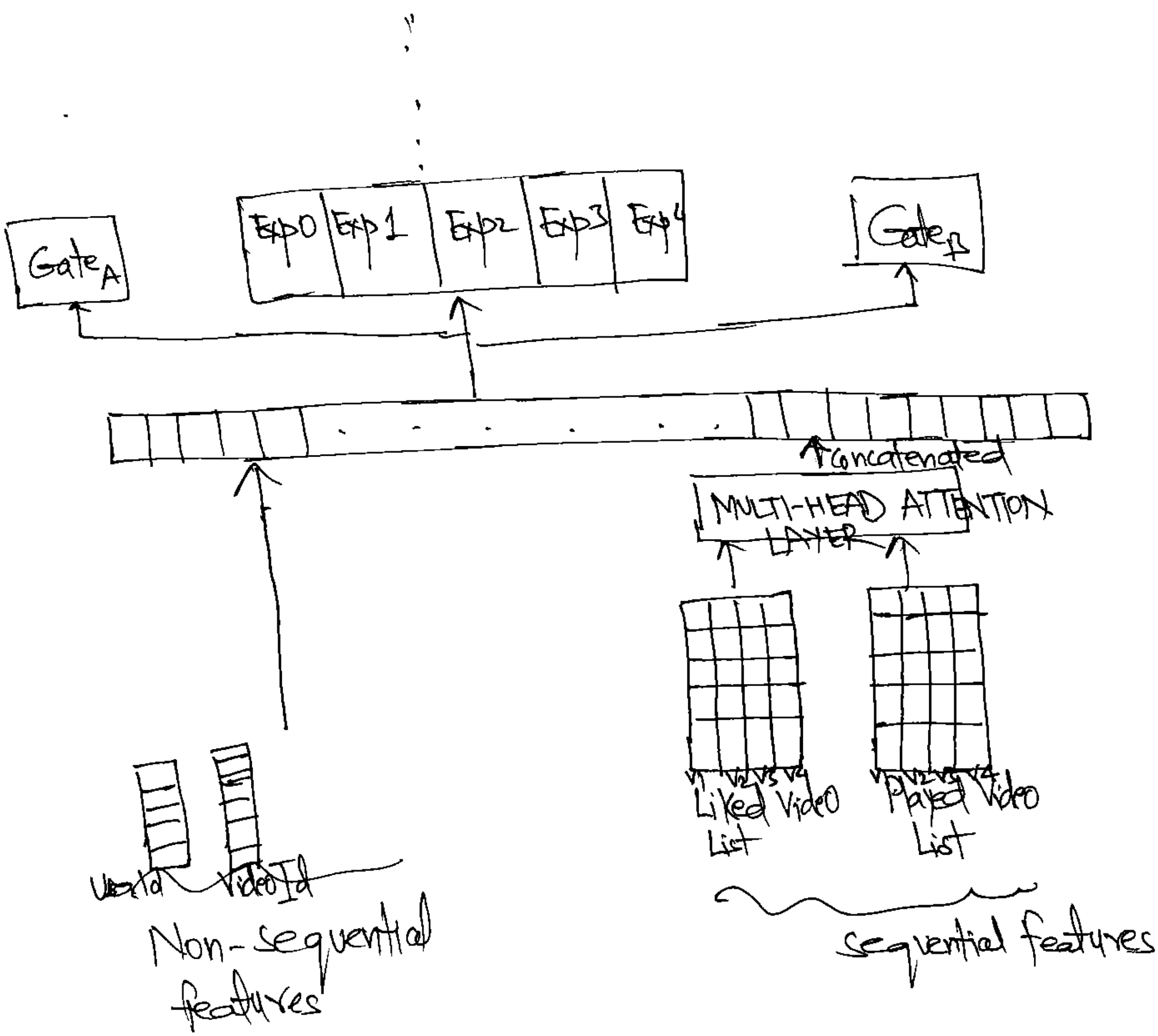
→ Target can be a combination of classification & regression

→ Loss FN = $\sum$ Cross-entropy loss for each task
(or MSE loss)

→ In practice, the lowest layer in MMoE is kept as 'shared' bcz the high dimensionality of I/P would lead to significant model training & serving costs.



HOW TO INGEST SEQ FEATURES INTO MMoE ARCHITECTURE



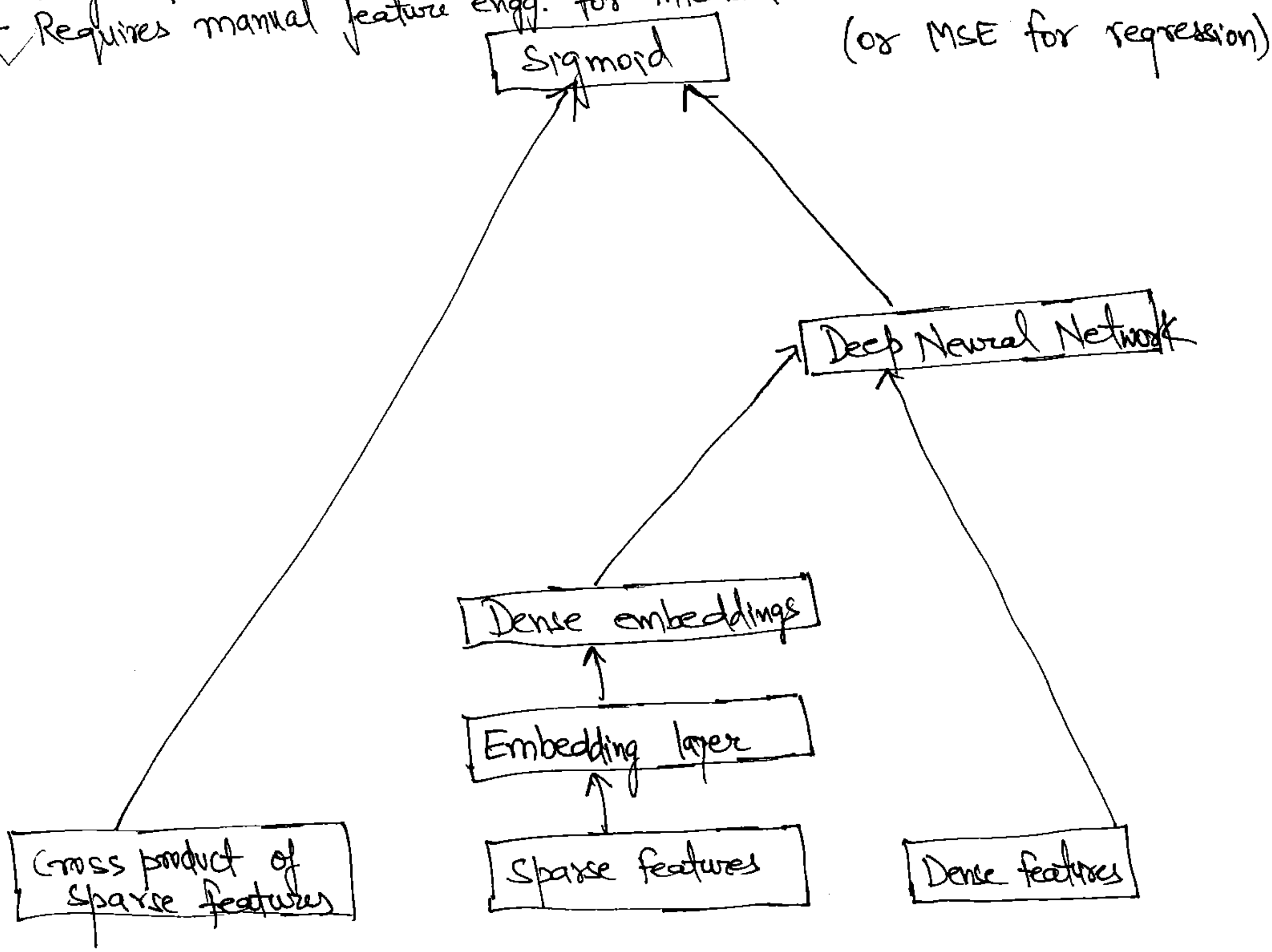
→ For each sequential feature, a multi-head attention module is used to fuse the embeddings into a concatenation of vector o/ps

→ These sequential and non-sequential vectors are then concatenated and fed into MMoE layer

WIDE AND DEEP MODEL (Google Play)

- Wide part can "memorize" seen feature interactions using cross-product feature transformations (need to manually hand craft these cross-features) unlike DeepFM or DCN

- Deep part can "generalize" to previously unseen feature interactions
- Jointly trains wide linear model and Deep NN
- Useful if dataset has several sparse features
- Requires manual feature engg. for interaction terms



FACTORIZATION MACHINES

- Extension of linear model that is designed to capture interactions between features within high-dimensional sparse datasets
- Mathematically,

$$\hat{y} = \underbrace{w_0}_{\text{global bias}} + \underbrace{\sum_i w_i x_i}_{\text{linear term}} + \sum_i \sum_{j>i} \langle v_i, v_j \rangle \, x_i x_j$$

factorization term that models pairwise interaction between i^{th} and j^{th} variable (dot product of i^{th} and j^{th} feature embeddings)

- Requires no manual feature-engg for interaction terms (unlike Wide & Deep Model)
- Can be used for classification as well as regression
- PROS: 1) The interaction between features can be estimated even under high sparsity

  2) The number of params as well as time for prediction and learning is linear

  3) Diminishes cold start of user or item i.e. if you know either some user features or some item features or some auxillary features, FM can kick start recommendations (unlike Matrix Factorization where you need user interaction for items)

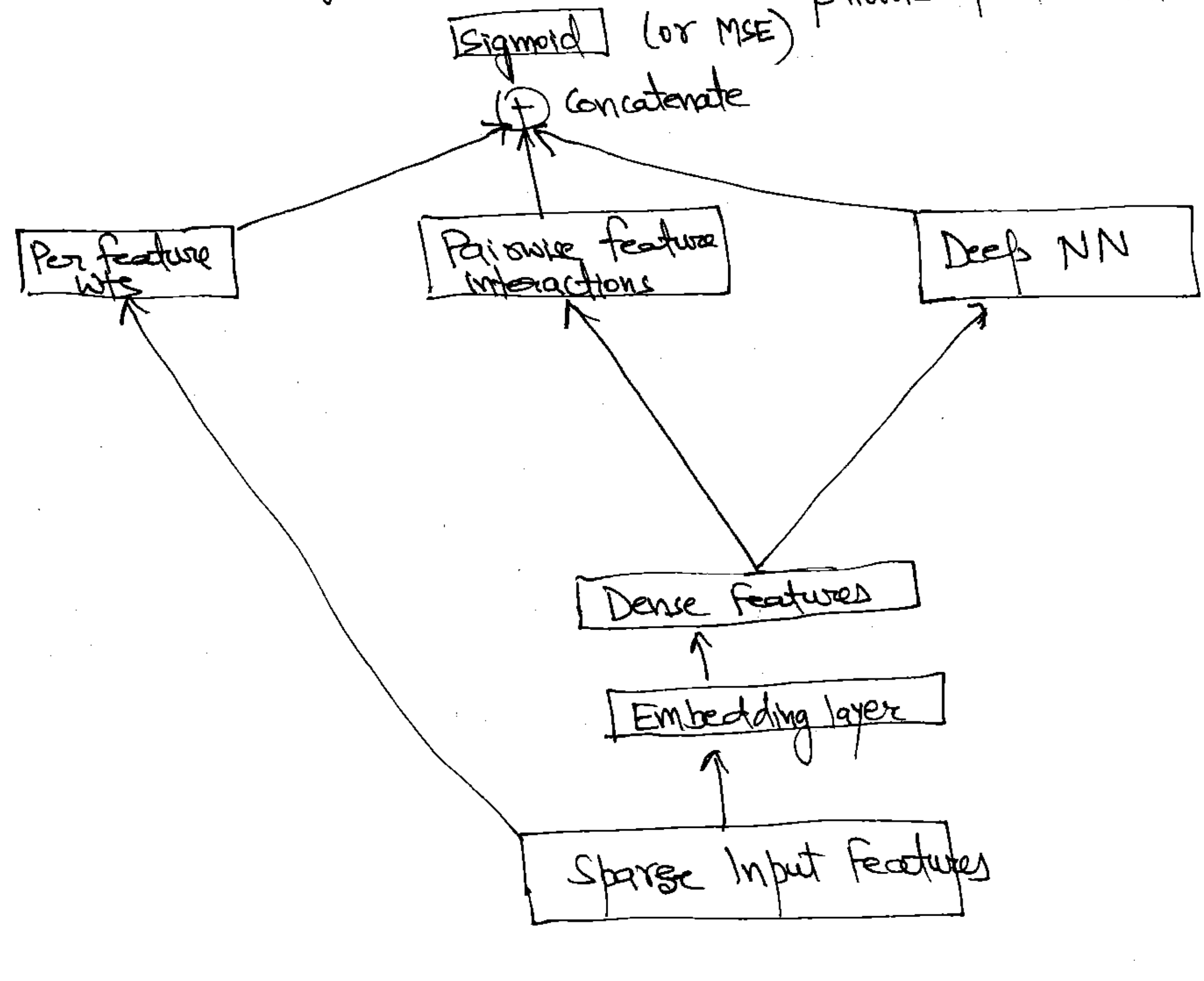
CONS: Only captures low level pairwise feature interactions (most implementations only support pairwise interactions e.g. Sagemaker's FM)

- DATASET looks something like this

1 0 0 0	1 1 1 0 0	1 1 1	1 0 0	5
0 0 1 0	1 0 0 0 0	0 0 1	0 1 1	4
User features	item features	context features	[illegible] features	Label (e.g. rating or binary label)

DEEP FACTORIZATION MACHINES (DeepFM)

- To capture higher order features along with FM's low level pairwise feature interactions

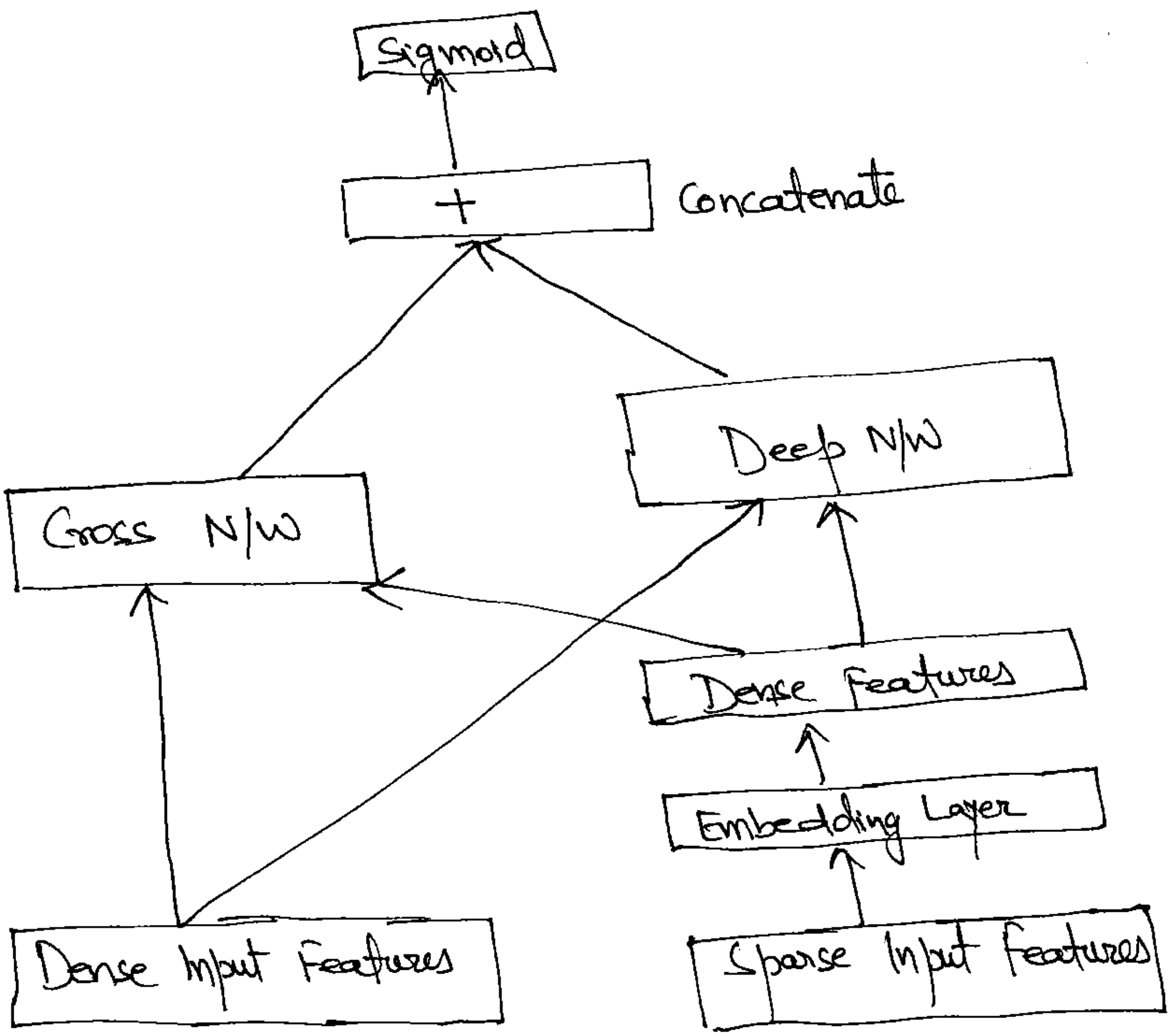


DEEP AND CROSS NETWORK (DCN)

- Improvement over Wide & Deep Network
  - no manual feature-engg required to find feature interactions

- Diff from DeepFM:
  - The polynomial degree of interaction term increases at each layer and is determined by layer depth (Generalization of FM to higher degree)

- CROSS N/W: Learns feature Crosses
  DEEP N/W: Learns complex and generalizable features



ONLINE EVALUATION OF RANKERS - INTERLEAVING

- Interleaving <u>speeds up</u> the process of determining which ranker is better since it requires smaller sample size (<u>same user</u> is presented interleaved result from multiple rankers)

- Usually, the following process is used

[offline metrics (e.g. NDCG)] → [interleaving] → [A/B Test]

interleaving: Fast online assessment (prune the rankers to only the promising ones)

<u>WHY</u>

- i) Interleaving is highly sensitive to ranking algo quality i.e. it reliably identifies the best algo with considerably <u>smaller sample size</u> compared to traditional A/B testing

  ii) Interleaving is predictive of success in second stage i.e. A/B test

- APPROACH (Team Drafting and Competitive Pairs)

HOW

Ranker 1 (ranked list)	Ranker 2 (ranked list)
A	X
B	B
C	A
D	Y
(Team A)	(Team B)

Flip a coin to determine which ranker goes first, then alternate b/w both rankers

A } Competitive Pair
X }
B
C
Y } Competitive Pair
D }

i) From both rankers, only select the highest ranked result that has not been selected yet

ii) If ~~the~~ the items in both list are different
→ Create a competitive pair (metrics will be calc. only on competitive pairs)

iii) If the items in both list are same, then select it but do not assign to either team (metrics on this item will not be taken into account to determine which ranker is better)

Metrics such as viewing hrs, clicks etc. are calculated for both teams from competitive pairs and statistical tests (such as t-test) are used to determine verdict

- NOTEWORTHY POINTS: BENEFITS

1) Flipping a coin to determine which ranker goes first and then alternating the turns → reduction of position bias since both rankers have equal prob. of going first

2) Only calc. metrics on competitive pairs and leaving non-competitive items in list improves sensitivity in determining better ranker. E.g. if 2 rankers produce the exact ranklist, the entire search query can be ignored since the preference is exactly zero. In traditional interleaving would still associate clicks to teams and add noise to the result

3) Same user sees results from both rankers hence variation in user characteristics is minimized compared to traditional A/B testing where one grp of users see variant A and another grp of users (with possible diff. characteristics) see variant B.

4) Needs engineering infrastructure to run this setup, if not already present

A/B TESTING

EXPERIMENTAL DESIGN:

1. → Metric : Pick a metric to test along with guardrail metrics

2. → Decide on significance level (α), power threshold ($1-\beta$) and Minimum Detectable Effect (MDE)

Significance level (α) [or ($1-\alpha$) confidence level]: Percent of the time a difference will be detected, assuming one does not exist (Type I or false +ve error). Usually α set to 0.05

Power : Percent of time minimum effect size will be detected assuming it exists.
$\beta \rightarrow$ Type II error or false -ve error
$1-\beta \rightarrow$ Power (prob. of not making a type II error)
Usually set to 0.8

Minimum Detectable Effect (MDE)

- Represents relative minimum improvement over the baseline that you are willing to detect in an experiment to a certain degree of statistical significance
- It is the smallest effect that will be detected $(1-\beta)$% of the time
- The smaller your MDE is, the larger the sample size required to reach statistical significance
- MDE is decided in consultation with stakeholders (e.g. 5%.)
- The smaller your baseline is, the larger the sample size required to detect the same MDE

Note: effect size = metric delta b/w control & treatment

(3) Stopping Cond. for A/B Test: Sample size & Experiment Duration

SAMPLE SIZE:

- Depends on a) baseline metric, b) statistical significance level (α), c) Statistical power ($1-\beta$) and d) MDE
- Sample size calculators can determine sample size needed per variation

EXPERIMENT LENGTH:

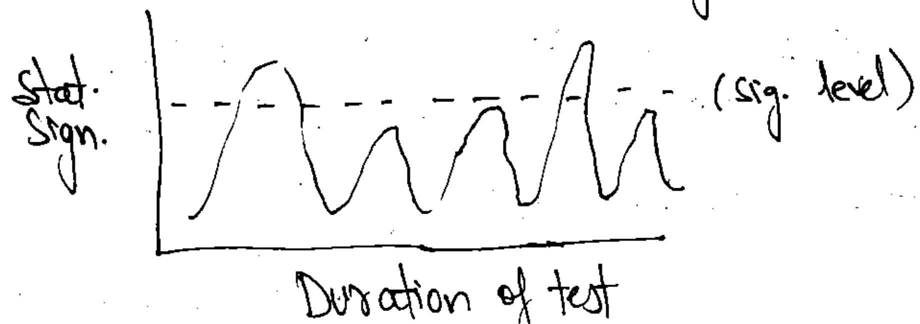
- Depends on daily traffic $\left(\frac{\text{total traffic for the feature under test}}{\text{daily traffic for the feature under test}}\right)$
- Should account for differences in usage pattern e.g. day of week, holidays etc. (Usually set at 2 weeks)

(4) Assign Grps.: Randomize control & treatment sufficiently to account for confounding variable down the line (e.g. day of week, season, word of mouth publicity, press (+ve or -ve))

A/B TESTING MISTAKES

→ **Stopping A/B Tests Early:**

Reaching statistical significance alone is not enough to declare a winner. bcz stat. sig. varies for the duration of test



Declare a winner only if:

1) Sample size requirement is met
2) Duration of test covered variations in day of week, holidays (etc) i.e. usually one business cycle (usually 2 weeks)

→ **Running multiple tests at the same time on overlapping traffic**

If you run multiple tests at the same time, some tests will succeed with statistical significance much larger than α

Family-wise Error Rate (FWER)

FPR or α = 0.05

Prob. of not committing Type I error : $1 - 0.05$

Prob of not committing Type I error for 10 tests = $(1-0.05)^{10}$

Prob. of committing Type I error atleast once for 10 tests

$= 1 - (1-0.05)^{10}$

$= 0.40$

$= 40\%$

Sol^n 1) Bonferroni Correction (BC)

Adjusts α level required based on total number of tests running

i.e. α/n where n = no. of tests running

E.g. we conduct multiple comparison of 10 metrics

By default $\alpha = 0.05$ for each test

Using Bonferroni correction new $\alpha = \frac{\text{old } \alpha}{n}$

$$= \frac{0.05}{10} = 0.005$$

Now, if we want to claim any statistically significant results, the respective p-value has to be smaller than 0.005

As seen, we have lowered the rejection bar from 0.05 to 0.005, making it much harder to reject Null hypothesis.

This method is criticized for being too conservative & lacking statistical power

2) Holm Bonferroni (HB)

Due to conservative nature of BC method, rarely used in practice

It controls FWER (family-wise error rate) by

i) ranking the p-values from each test

ii) adjusting rejection criteria for each null hypothesis

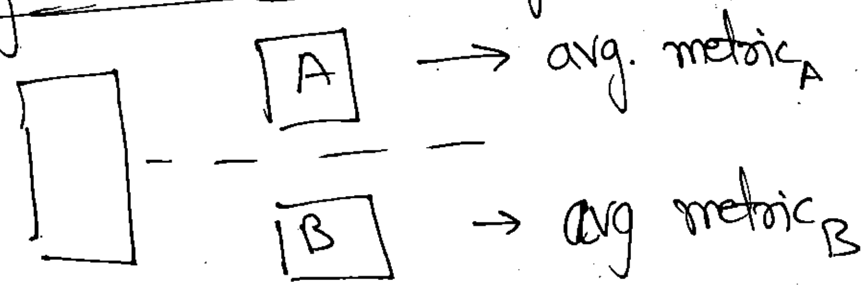
Diff b/w BC & HB: HB's rejection threshold depends on all p-values (unlike BC method)

HB has more statistical power than BC method

3) FDR (False Discovery Rate)

- Proportion of false +ves among all significant results
- Eg. if you have made 100 significant results and set FDR to 0.05 then 5 of them are false discoveries (or false rejection)

→ Dealing with non-normality



A/B tests are dressed up Z-tests and t-tests and these tests assume the var. of interest (metric we are testing for) is normally distributed.

This assumption is usually true due to large sample size + Central Limit Theorem.

But if the normality assumption of metric does not hold true then we have to use other techniques:

- Running alternate test: Wilcoxon rank-sum test does not assume normality

- Gather more data to invoke CLT

→ **<u>Dealing with Novelty Effects / Primary Effects</u>**

A/B tests may have an exaggerated initial effect due to novelty effect where a new feature attracts social media attention and PR hype causing inquisitive users to rush to checkout new changes and thereby leading metric to jump spuriously

In the opposite way, there could be primary effect where users are fixed on what is familiar and have an aversion to changes making test metrics to be stagnant / low initially.

To detect novelty or primary effect, slice & dice the A/B test results on <u>new users</u> only

→ **<u>Dealing with network effects</u>**

Generally assumed that each user in control & treatment grps are independent i.e. no interference b/w control & treatment. However, this assumption does not hold true all the time.

Example: in social n/ws – the action of users are likely impacted by those around them (who may be in other grp (control/treatment))

Facebook live : treatment grp users may find it entertaining & post abt it & send links to friends (who may start using the feature) → can lead to increased engagement in control grp

To solve for it → Create clusters of connected people and assign them as control or treatment grp

→ Nuances of taking action on A/B Test Results:

→ Sample size & duration of test ✓

→ Statistical significance reached ✓

→ No significant drops in guardrail metrics ✓

Also, time & effort to build & maintain scale feature < business impact of feature

→ A/B Test Universal Holdouts

Useful in quantifying the business impact of entire Org

E.g.

Team A → 5% ↑ in conversion

B → 8% ↑ in conversion

⋮

N → N% ↑ in conversion

Universal Holdout : No tests are run for the grp of users

Grp of users part of any test	Universal Holdout
↓	↓
Conversion	Conversion

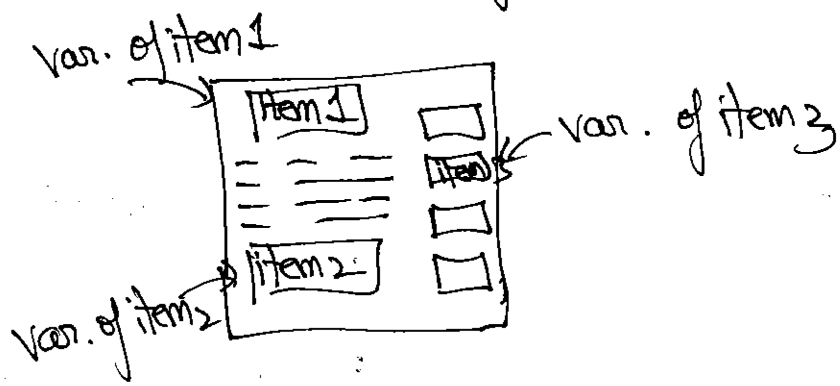
Δ in conversion metrics = O/P of entire Org

BANDIT BASED A/B TESTING: (MAB)

- Bandit algorithm starts sending ~~more~~ traffic to "winning" variant to minimize "regret"
- Useful in scenarios where you can't wait to finish regular A/B test to determine decrease in metric or the metric ↓ has a huge cost
- Explore vs Exploit
  - ↑ traffic allocation to "winning" variant until that pt. in time
- Significance level may not be reached

MULTIVARIATE TESTS (these are not A/B/n tests)

→ Test multiple versions of ~~diff~~ ~~elements~~ same page to isolate which attributes cause the largest impact



→ Ton of traffic needed to perform Multivariate test

→ They test original against variation but each variation contains diff. design elements (e.g. which combination of item 1, 2 and 3 variation ↑ metric) [figures out interaction effect]

MONITORING

INFRASTRUCTURE:
- * — COST
- Amount of CPU/GPU
- Memory
- Disk I/O
- N/W I/O

SERVING METRICS
- Throughput (MBps) (no. of concurrent users)
- Latency (client ⇄ server)

PREDICTION/SCORING PIPELINE:
- Kafka → Flink consumer lag
- Real-time processing pipeline (Flink): Error handling, Timeout
- Populating into feature store: Distribution drift b/w training & prediction data (data drift) training-serving skew
- * — Featurization code similar to training pipeline
- Outages in pipelines
- Upstream schema changes in data stores

MODEL:
- Model performance metric change e.g. precision, recall (if label is available in an acceptable time frame)
- Model prediction metrics (i.e. flag 99% of users suddenly) (if label not available in an acceptable time frame)

TRAINING PIPELINE: Base table schema changes / Unavailability (e.g. using Deequ to monitor)

MODEL TRAINING MONITOR: (E.g. Sagemaker Debugger)
- Vanishing gradient, overfitting, poor_wt_initialization, dead_relu, exploding tensor, loss_not_decreasing
- Data leakage

RE-TRAINING FREQUENCY

1. Periodic (e.g. weekly)
2. When model metrics degrade

REDUCING LATENCY (WHILE SCORING)

1. PRE-COMPUTING FEATURES, EMBEDDINGS
2. CACHING FEATURES, EMBEDDINGS AND PREDICTIONS
   if used by several downstream models
3. MODEL COMPRESSION :- Quantization
   (reduce wts and activations from float32 to int 8)
   - Knowledge distillation
   - Pruning wts and connections that contribute least to model performance
4. USE SIMPLER FEATURE ENGINEERING
5. USE MORE HARDWARE | INFRASTUCTURE RESOURCES FOR INFERENCE PIPELINE

SCALING SCORING

- Docker Container + Kubernetes, Load balancer + Auto-scaling
  (e.g. No. of requests threshold)

REDUCING TRAINING TIME:

1. Distributed Training
2. Large Batch sizes (reduce the no. of times parameters are updated)
3. Utilize GPU acceleration: GPUs are optimized for parallel processing, have large memory bandwidth and are designed specifically for matrix calc.

4. Downsample training data
5. Use optimized implementation of algorithms e.g. tensorflow, pytorch
6. Optimize learning rate schedule: fine tune learning rate to balance speed of convergence and stability of optimization process

MODEL CALIBRATION: (Especially useful in multi stage recommendation system)

- Necessary when
  1) Comparing experimental vs production model (or any two versions of model)
  2) Model O/P is consumed by a downstream model (or subsequent models in multi-stage settings)

- Example: Consider 2 models; (e.g. for CTR prediction)
  - 1^{st} model : score varies b/w 0 to 0.2
  - 2^{nd} model : score varies b/w 0 to 1

  Both models have same test set performance (AUC) bcz their relative ordering of predictions is the same

  Their scores are not directly comparable, a score of 0.2 means a very high CTR for the first model and relatively low CTR for the second model

  If production and experimental models are not calibrated to the same distribution then downstream systems (e.g. ranking or re-ranking functions) that rely on model score distributions will need to be re-tuned (unsustainable in the long run)

  If calibrated then model experiments and launches can be decoupled from the changes to the rest of recommendation system

CALIBRATION: Converting model scores to real-world probability in the range [0,1]

To all the samples that a classifier gave 0.8 (pred. prob) approx 80% actually belong to +ve class
→ well-calibrated binary classifier

Calibrating a classifier consists of fitting a regressor that maps
O/P of classifier → calibrated prob in [0,1]

Two methods
- Platt Scaling : fitting a sigmoid fn
- Isotonic regression : fitting a discrete, stepwise, monotonically inc. fn

(We know O/P of uncalibrated classifier & true label hence can fit a fn)

PROS AND CONS OF ML ALGOS

LOGISTIC REGRESSION:

PROS:
- FAst training
- fast inference
- Interpretable
- Can be update thru continuous learning (online learning)

CONS:
- Non-linear problems can't be solved
- Inability to capture feature interactions

DECISION TREES:

PROS:
- Fast training
- Fast inference
- Little to no data prep : Normalizing ✗
  Scaling ✗
  Missing values ✗
- Interpretable

CONS:
- Overfitting

RANDOM FOREST (BAGGING + "random" selection of features)

PROS
- Does not significantly increase training time bcz DT can be trained in parallel
- Does not add much latency at inference time bcz DT can process input in parallel
- Reduces overfitting (high variance)
- Interpretable (feature importance ✓, coeff. ✗)
- Non-linear decision boundary
- Online learning possible (Mondrian Forests)
- Easy data prep.

CONS:
- Not helpful in high bias scenarios

BOOSTING (GBDT, xgboost)

PROS:
- Though it builds trees sequentially, training time can be reduced with efficient implementations like xgboost (builds multiple branches of tree in parallel) + distributed training possible
- Implementations available for fast inference (catboost, lightGBM)
- Reduces both bias & variance
- Easy data prep: Normalization ✗ Scaling ✗ Missing values ✗
- Works well with structured data

CONS:
- Unsuitable for continuous learning from streaming data
- Does not work well on unstructured data
- Lots of hyperparams to tune

Neural NETWORKS

PROS
- Continual learning possible
- Works well with unstructured data + embeddings
- Can learn very complex tasks and non-linear decision boundaries

CONS:
- Large training data is required
- Computationally expensive to train
- Some effort in data prep. Eg. if input features are in very different ranges, the model may converge very slowly.
  normalization, log scaling, one-hot encoding, embedding
- Black-box nature in interpretability

DELAYED LABELS

Issue
- model becomes stale
- increases false -ves

SOLUTION:

i) Proxy label : If there is some info/feature that co-relates strongly with label and is available before

ii) Re-formulate the ML problem with a time horizon
Instead of whether fraud will be committed by a particular user?
reformulate it as
whether fraud will be committed in the next 30 days by this user?

iii) Supervised Anomaly Detection:
Using historical data of bad actors, use anomaly detection algos like one-class SVM (trained only on one class) and essentially learns the decision boundary/distribution of one class

iv) Unsupervised : Auto-Encoders
Re-construction loss above a threshold

v) Domain Experts : craft/annotate labels