Feature Engineering

Let's engineer features for the candidate generation and ranking model.

We'll cover the following

- Features
  - User-based features
  - Context-based features
  - Media-based features
  - Media-user cross features

To start the feature engineering process, we will first identify the main **actors** in the movie/show recommendation process:

1. Logged-in user
2. Movie/show
3. Context (e.g., season, time, etc.)

Features

Now it's time to generate features based on these actors. The features would fall into the following categories:

1. User-based features
2. Context-based features
3. Media-based features
4. Media-user cross features

A subset of the features is shown below:

User-based features

Let's look at various aspects of the user that can serve as useful features for the recommendation model.

- **age**

  This feature will allow the model to learn the kind of content that is appropriate for different age groups and recommend media accordingly.
- **gender**

  The model will learn about gender-based preferences and recommend media accordingly.
- **language**

  This feature will record the language of the user. It may be used by the model to see if a movie is in the same language that the user speaks.
- **country**

  This feature will record the country of the user. Users from different geographical regions have different content preferences. This feature can help the model learn geographic preferences and tune recommendations accordingly.
- **average_session_time**

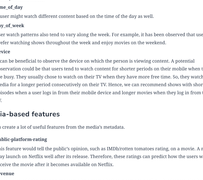
  This feature (user's average session time) can tell whether the user likes to watch lengthy or short movies/shows.
- **last_genre_watched**

  The genre of the last movie that a user has watched may serve as a hint for what they might like to watch next. For example, the model may discover a pattern that a user likes to watch thrillers or romantic movies.

The following are some user-based features (derived from historical interaction patterns) that have a *sparse representation*. The model can use these features to figure out user preferences.

- **user_actor_histogram**

  This feature would be a vector based on the histogram that shows the historical interaction between the active user and all actors in the media on Netflix. It will record the percentage of media that the user watched with each actor cast in it.



- **user_genre_histogram**

  This feature would be a vector based on the histogram that shows historical interaction between the active user and all the genres present on Netflix. It will record the percentage of media that the user watched belonging to each genre.
- **user_language_histogram**

  This feature would be a vector based on the histogram that shows historical interaction between the active user and all the languages in the media on Netflix. It will record the percentage of media in each language that the user watched.

Context-based features

Making context-aware recommendations can improve the user's experience. The following are some features that aim to capture the contextual information.

- **season_of_the_year**

  User preferences may be patterned according to the four seasons of the year. This feature will record the season during which a person watched the media. For instance, let's say a person watched a movie tagged "*summertime*" (by the Netflix tagger) during the *summer season*. Therefore, the model can learn that people prefer "*summertime*" movies during the *summer season*.
- **upcoming_holiday**

  This feature will record the upcoming holiday. People tend to watch holiday-themed content as the different holidays approach. For instance, Netflix tweeted that fifty-three people had watched the movie "A Christmas Prince" daily for eighteen days before the Christmas holiday. Holidays will be region-specific as well.
- **days_to_upcoming_holiday**

  It is useful to see how many days before a holiday the users started watching holiday-themed content. The model can infer how many days before a particular holiday users should be recommended holiday-themed media.
- **time_of_day**

  A user might watch different content based on the time of the day as well.
- **day_of_week**

  User watch patterns also tend to vary along the week. For example, it has been observed that users prefer watching shows throughout the week and enjoy movies on the weekend.
- **device**

  It can be beneficial to observe the device on which the person is viewing content. A potential observation could be that users tend to watch content for shorter periods on their mobile when they are busy. They usually have more time on their TV when they have more free time. So, they watch media for a longer period consecutively on their TV. Hence, we can recommend shows with short episodes when a user logs in from their mobile device and longer movies when they log in from their TV.

Media-based features

We can create a lot of useful features from the media's metadata.

- **public-platform-rating**

  This feature would tell the public's opinion, such as IMDb/rotten tomatoes rating, on a movie. A movie may launch on Netflix well after its release. Therefore, these ratings can predict how the users will receive the movie after it becomes available on Netflix.
- **revenue**

  We can also add the revenue generated by a movie before it came to Netflix. This feature also helps the model to figure out the movie's popularity.
- **time_passed_since_release_date**

  The feature will tell how much time has elapsed since the movie's release date.
- **time_on_platform**

  It is also beneficial to record how long a media has been present on Netflix.
- **media_watch_history**

  Media's watch history (number of times the media was watched) can indicate its popularity. Some users might like to stay on top of trends and focus on only watching popular movies. They can be recommended popular media. Others might like less discovered indie movies more. They can be recommended less watched movies that had good implicit feedback (the user watched the whole movie and did not leave it midway). The model can learn these patterns with the help of this feature.

We can look at the media's history for different time intervals as well. For instance, we can have the following features:

- media_watch_history_last_12_hrs
- media_watch_history_last_24_hrs

> 📝 The media-based features listed above can collectively tell the model that a particular media is a blockbuster, and many people would be interested in watching it. For example, if a movie possesses a large revenue, has a good IMDb rating, came to the platform 24 hours ago, and a lot of people have watched it, then it is definitely a blockbuster.

- **genre**

  This feature records the primary genre of content, e.g., comedy, action, documentaries, classics, drama, animated, and so on.
- **movie_duration**

  This feature tells the movie duration. The model may use it in combination with other features to learn that a user may prefer shorter movies due to their busy lifestyle or vice versa.
- **content_set_time_period**

  This feature describes the time period in which the movie/show was set in. For example, it may show that the user prefers shows that are set in the 90s.
- **content_tags**

  Netflix has hired people to watch movies and shows to create extremely detailed, descriptive, and specific tags for the movies/shows that capture the nuances in the content. For instance, media can be tagged as a "Visually-striking nostalgic movie". These tags greatly help the model understand the taste of different users and find the similarity between the user's taste and the movies.
- **show_season_number**

  If the media is a show with multiple seasons, this feature can tell the model whether a user likes shows with fewer seasons or more.
- **country_of_origin**

  This feature holds the country in which the content was produced.
- **release_country**

  This feature holds the country where the content was released.
- **release_year**

  This feature shows the year of theatrical release, original broadcast date or DVD release date.
- **release_type**

  This feature shows whether the content had a theatrical, broadcast, DVD, or streaming release.
- **maturity_rating**

  This feature contains the maturity rating of the media with respect to the territory (geographical region). The model may use it along with a user's age to recommend appropriate movies.

Media-user cross features

In order to learn the users' preferences, representing their historical interactions with media as features is very important. For instance, if a user watches a lot of Christopher Nolan movies, that would give us a lot of information about what kind of movies the user likes. Some of these interaction-based features are as follows:

User-genre historical interaction features

These features represent the percentage of movies that the user watched with the same genre as the movie under consideration. This percentage is calculated for different time intervals to cater to the dynamic nature of user preferences.

- **user_genre_historical_interaction_1month**

  The percentage of movies that the user watched with the same genre as the movie under consideration in the last 3 months. For example, if the user watched 6 comedy movies out of the 52 he/she watched in the last 3 months, then the feature value will be:

  $\frac{6}{12} = 0.5$ or 50%

  This feature shows a more recent trend in the user's preference for genres as compared to the following feature.
- **user_genre_historical_interaction_1year**

  This is the same feature as above but calculated for the time interval of one year. It shows a more long term trend in the relationship between the user and genre.
- **user_and_movie_embedding_similarity**

  Netflix has hired people to watch movies and shows to create incredibly detailed, descriptive, and specific tags for the movies/shows that capture the nuances in the content. For instance, media can be tagged as "Visually-striking nostalgic movie".

  You can have a user embedding based on the tags of movies that the user has interacted with and a media embedding based on its tags. The dot product similarity between these two embeddings can also serve as a feature.
- **user_actor**

  This feature tells the percentage of media that the user has watched, which has the same cast (actors) as that of the media under consideration for recommendation.
- **user_director**

  This feature tells the percentage of movies that the user has watched with the same director as the movie under consideration.
- **user_language_match**

  This feature matches the user's language and the media's language.
- **user_age_match**

  You will keep a record of the age bracket that has mostly viewed a certain media. This feature will see if the user watching a particular movie/show falls into the same age bracket. For instance, movie A is mostly (80% of the times) watched by people who are 40+. Now, while considering movie A for a recommendation, this feature will see if the user is 40+ or not.

Some *sparse features* are described below. Each of them can show results in their respective domains and also the preferences of individual users. We will go over how these sparse features are used in the ranking chapter. You can also go over the embedding chapter about how to generate vector representation of this sparse data to use them in machine learning models.

- **movie_id**

  Popular movie IDs are be repeated frequently.
- **title_of_media**

  This feature holds the title of the movie or the TV/ web series.
- **synopsis**

  This feature holds the synopsis or summary of the content.
- **original_title**

  This feature holds the original title of the movie in its original language. The media may be released for a different country with a different title keeping in view the preference of the nationals. For example, *JapaneseKoreanmovieshows* are released for English speaking countries with English titles as well.
- **distributor**

  A particular distributor may be selecting very good quality content, and hence users might prefer content from that distributor.
- **creator**

  This feature contains the creator/s of the content.
- **original_language**

  This feature holds the original spoken language of the content. If multiple, you can record the choose the majority language.
- **director**

  This feature holds the director/s of the content. This feature can indicate directors who are widely popular, such as Steven Spielberg, and it can also showcase the individual preference of users.
- **first_release_year**

  This feature holds the year in which content had its first release anywhere (this is different from production year).
- **music_composer**

  The music in a show or a film's score can greatly enhance the storytelling aspect. Users may fancy the work of a particular composer and may be more drawn to their work.
- **actors**

  This feature includes the cast of the movie/show.

Sparse features like the distributor and synopsis can be made into dense features.

- Dense representation of distributor feature: Does the distribution of the movie fall in the list of top ten distributors?
- Dense representation of synopsis feature: Perform text summarization of synopsis to pull out the keywords.

However, this process requires extra effort on the engineer's part so you will let the neural network figure out the relationships (This approach will be discussed in the ranking lesson).

Candidate Generation

The purpose of candidate generation is to select the top k (let's say one-thousand) movies that you would want to consider showing as recommendations to the end-user. Therefore, the task is to select these movies from a corpus of more than a million available movies.

We'll cover the following

In this lesson, we will be looking at a few techniques to generate media candidates that will match user interests based on the user's historical interaction with the system.

Candidate generation techniques

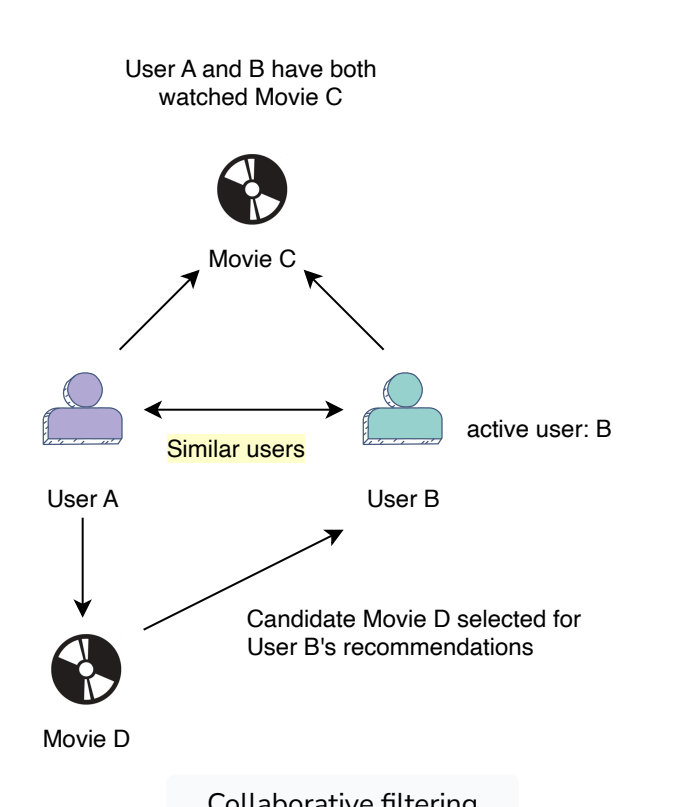
The candidate generation techniques are as follows:

1. Collaborative filtering
2. Content-based filtering
3. Embedding-based similarity

Each method has its own strengths for selecting good candidates, and we will combine all of them together to generate a complete list before passing it on to the ranked (this will be explained in the ranking lesson).

Collaborative filtering

In this technique, you find users similar to the *active user* based on the intersection of their historical watches. You, then, collaborate with similar users to generate candidate media for the active user, as shown below.



Collaborative filtering

Minimize

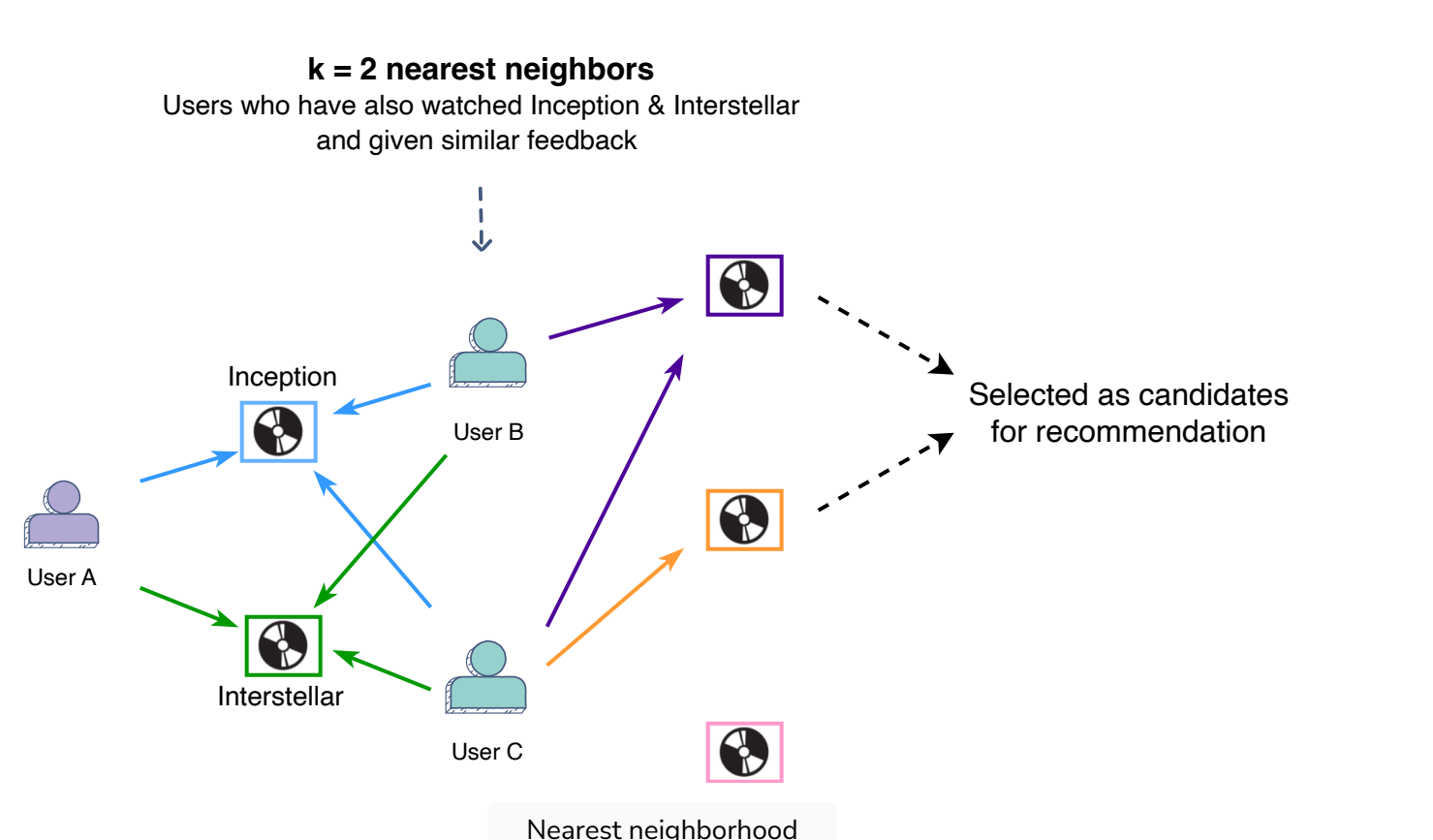
If a user shares a similar taste with a group of users over a subset of movies, they would probably have similar opinions on other movies compared to the *opinion of a randomly selected person*.

There are two methods to perform collaborative filtering:

1. Nearest neighborhood
2. Matrix factorization

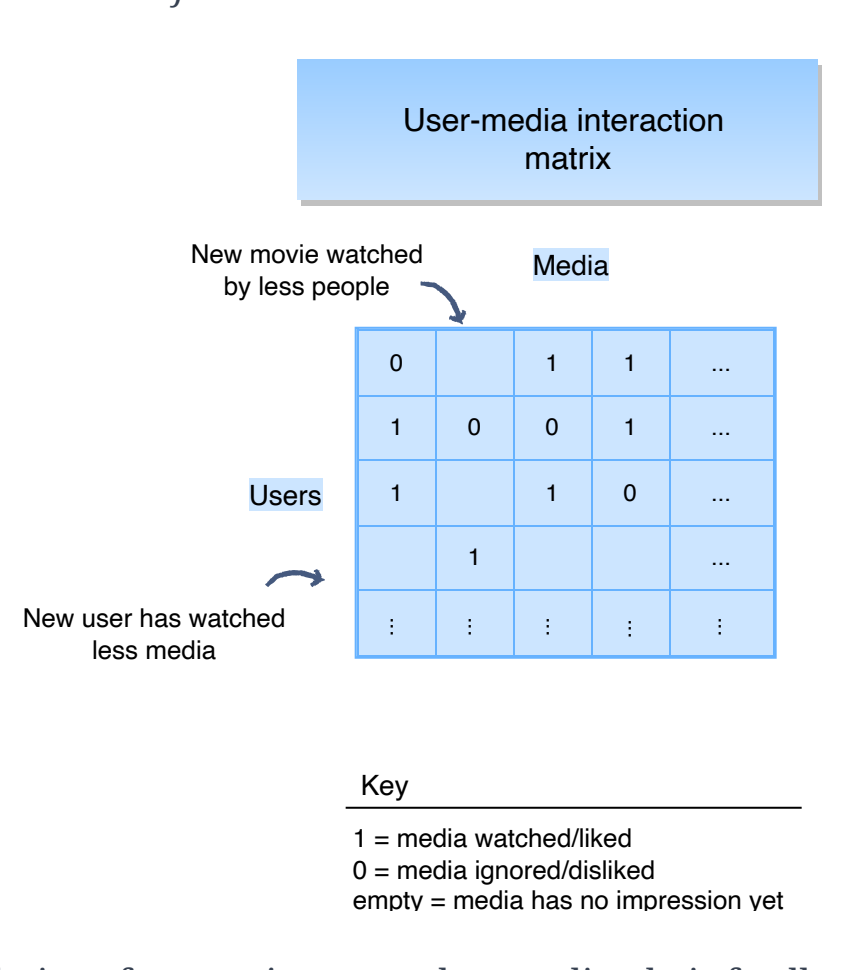
Method 1: Nearest neighborhood

User A is similar to user B and user C as they have watched the movies Inception and Interstellar. So, you can say that user A's nearest neighbours are user B and user C. You will look at other movies liked by users B and C as candidates for user A's recommendations.



Nearest neighborhood

Let's see how this concept is realized. You have a (n x m) matrix of user $u_{i(i=1 \ to \ n)}$ and movie $m_{j(j=1 \ to \ m)}$. Each matrix element represents the feedback that the user *i* has given to a movie *j*. An empty cell means that user *i* has not watched movie *j*.



To generate recommendations for user *i*, you need to predict their feedback for all the movies they haven't watched. You will collaborate with users similar to user *i* for this process. Their ratings for a movie, not seen by user i, would give us a good idea of how user *i* would like it.

So, you will compute the similarity (e.g. cosine similarity) of other users with user *i* and then select the top k similar users/nearest neighbours (KNN(u_i)). Then, user *i's* feedback for an unseen *movie j* (f_{ij}) can be predicted by taking the weighted average of feedback that the top k similar users gave to movie *j*. Here the feedback by the nearest neighbour is weighted by their similarity with user *i*.

$$f_{ij} = \frac{\sum_{v \in KNN(u_i)} Similarity(u_i, u_v) f_{vj}}{k}$$

The unseen movies with good predicted feedback will be chosen as candidates for user *i's* recommendations.

Minimize

We were looking at the **user-based approach** of collaborative filtering (CF) where you were identifying similar users. There is another approach known as the **item-based approach** where we look at the similarity of the items(movies/shows) for generating candidates. First, you calculate media similarity based on similar feedback from users. Then the media most similar to that already watched by user A will be selected as candidates for user A's recommendation.

The user-based approach is often harder to scale as user preference tends to change over time. Items, in contrast, don't change so the item-based approach can usually be computed offline and served without frequent re-training.

It is evident that this process will be computationally expensive with the increase in numbers of users and movies. The sparsity of this matrix also poses a problem when a movie has not been rated by any user or a new user has not watched many movies.

Method 2: Matrix factorization

As explained above, you need to represent the *user-media interaction matrix* in a way that is scalable and handles sparsity. Here, matrix factorization helps us by factoring this matrix into two lower dimensional matrices:

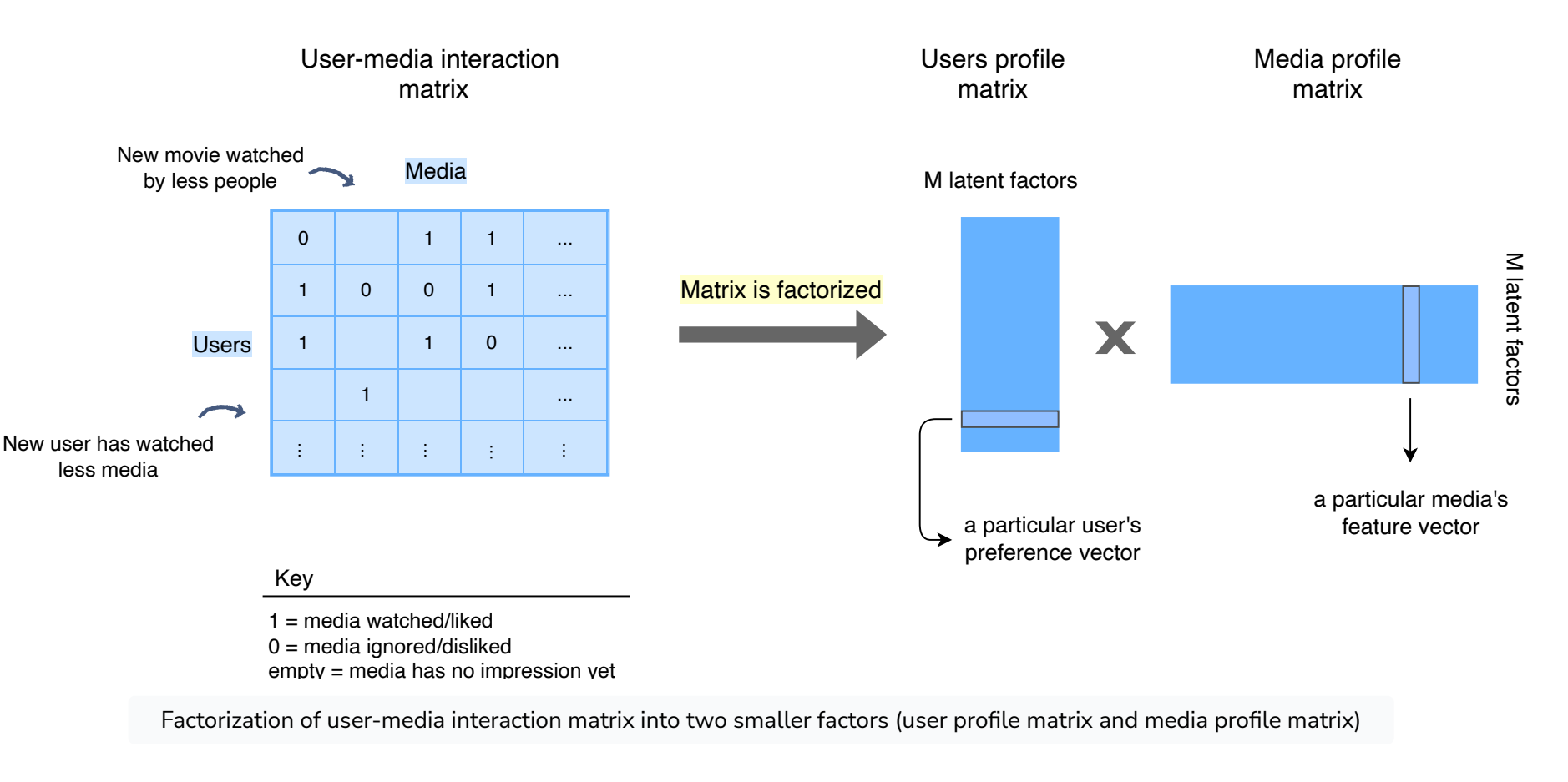
1. User profile matrix (n x M)

   Each user in the user profile matrix is represented by a row, which is a *latent vector of M dimensions*.

2. Media profile matrix (M x m)

   Each movie in the movie profile matrix is represented by a column, which is a latent vector of M dimensions.

The dimension M is the number of latent factors we're using to estimate the user-media feedback matrix. *M* is much smaller than the actual number of users and number of media.

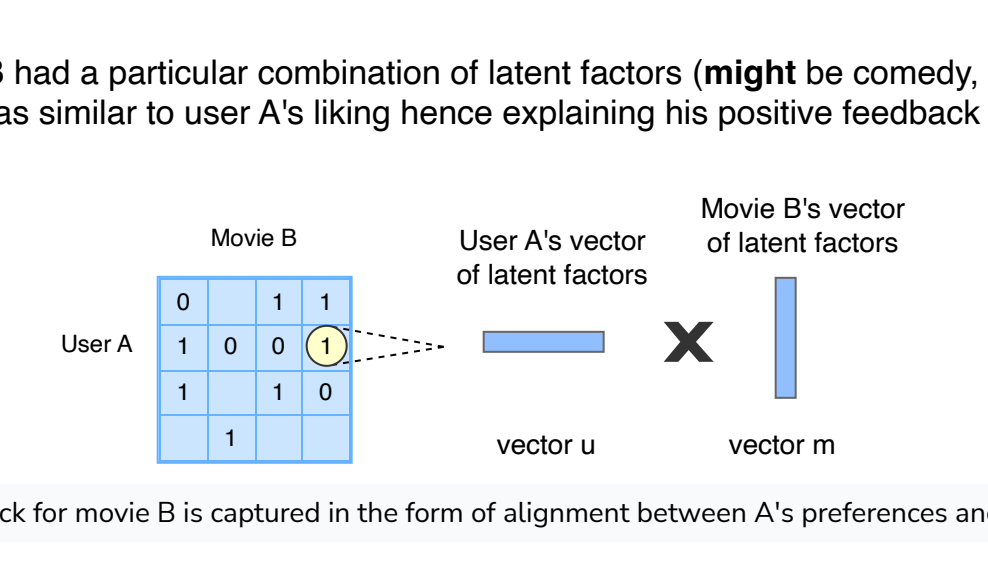


Factorization of user-media interaction matrix into two smaller factors (user profile matrix and media profile matrix)

The representation of users and media in the form of a *vector of latent factors* aims to explain the reason behind a user's particular feedback for a media. Latent vectors can also be thought of as features of a movie or a user. For example, in the diagram below, two of the latent factors might loosely represent the amount of comedy and quirkiness.

> 📝 Note that we say the word "might" and "loosely" for latent factors because we don't know exactly what each dimension means, it is "hidden" from us.

User A's vector has their preferences: 1 for the "comedy" and 0 for "quirkiness". Movie B's vector contains the presence/absence of the two factors in the movie, e.g., 1 and 0. The *dot product of these two vectors* will tell how much movie B aligns with the preferences of the user A; hence, the feedback.



User A's feedback for movie B is captured in the form of alignment between A's preferences and B's characteristics

The first step is to create the user profile and movie profile matrices. Then, you can generate good candidates for movie recommendation by predicting user feedback for unseen movies. This prediction can be made simply by computing the dot product of the user vector with the movie vector.

Now, let's go over the process of how to learn the latent factor matrices for users and media.

You will initialize the user and movie vectors randomly. For each known/historical user-movie feedback value f_{ij}, you will predict the movie feedback by taking the dot product of the corresponding user profile vector u_i and movie profile vector m_j. The difference between the actual (f_{ij}) and the predicted feedback ($u_i.m_j$) will be the error (e_{ij}).

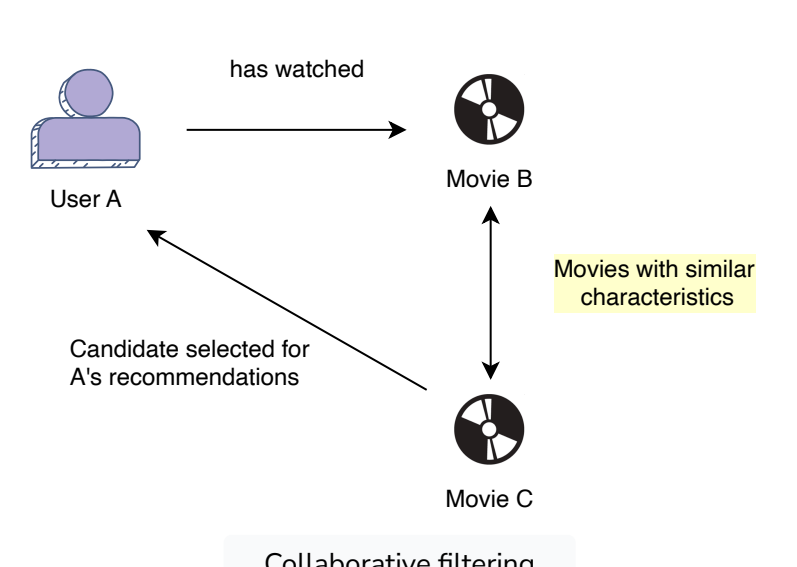
$$e_{ij} = f_{ij} - u_i.m_j$$

You will use stochastic gradient descent to update the user and movie latent vectors, based on the error value. As you continue to optimize the user and movie latent vectors, you will get a semantic representation of the users and movies, allowing us to find new recommendations that are closer in that space.

> 📝 The user profile vector will be made based on the movies that the user has given feedback on. Similarly, the media/movie profile vector will be made based on its user feedbacks. By utilizing these user and movie profile vectors, you will generate candidates for a given user based on their predicted feedback for unseen movies.

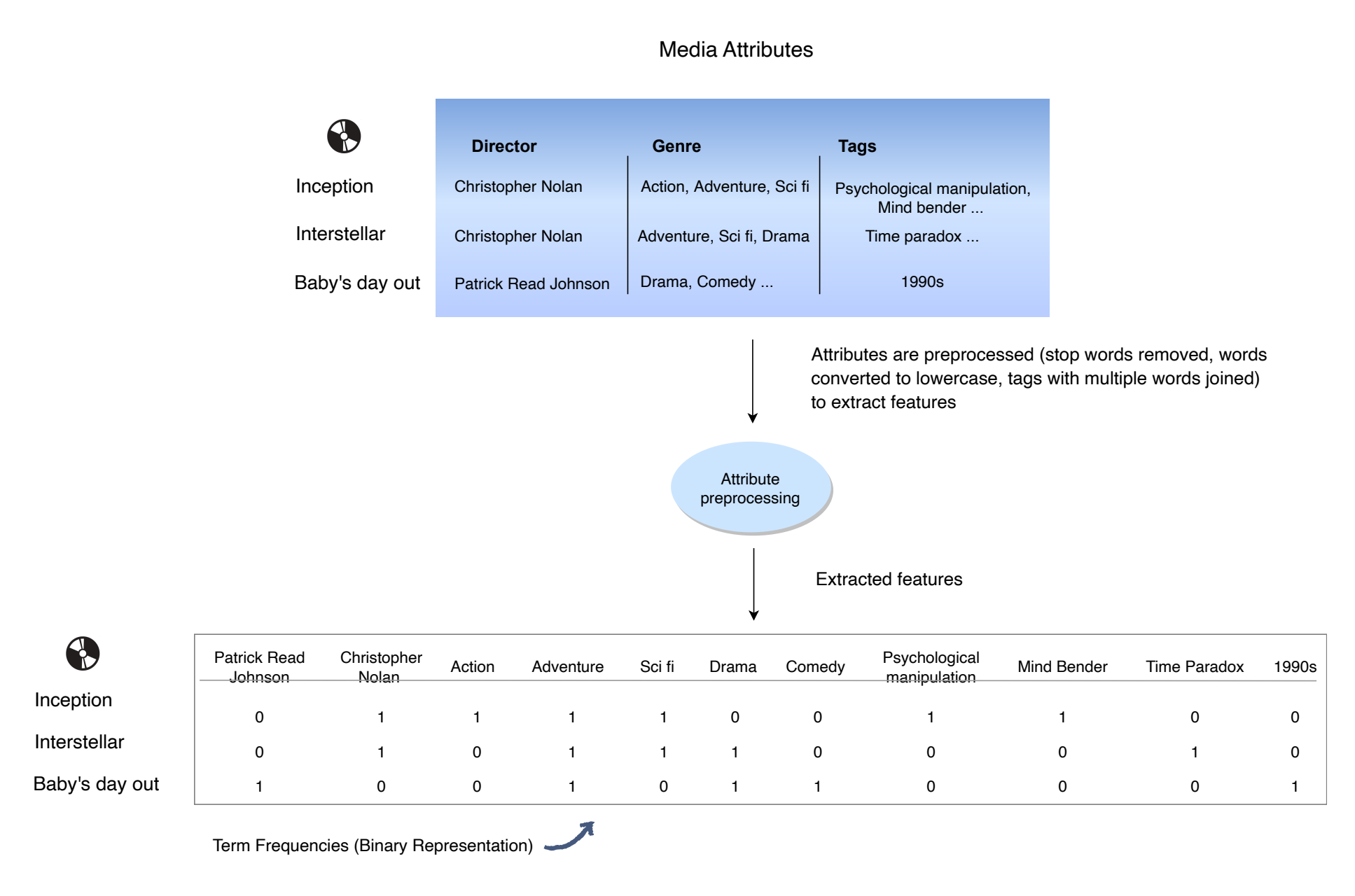
Content-based filtering

Content-based filtering allows us to make recommendations to users based on the characteristics or attributes of the media they have already interacted with.



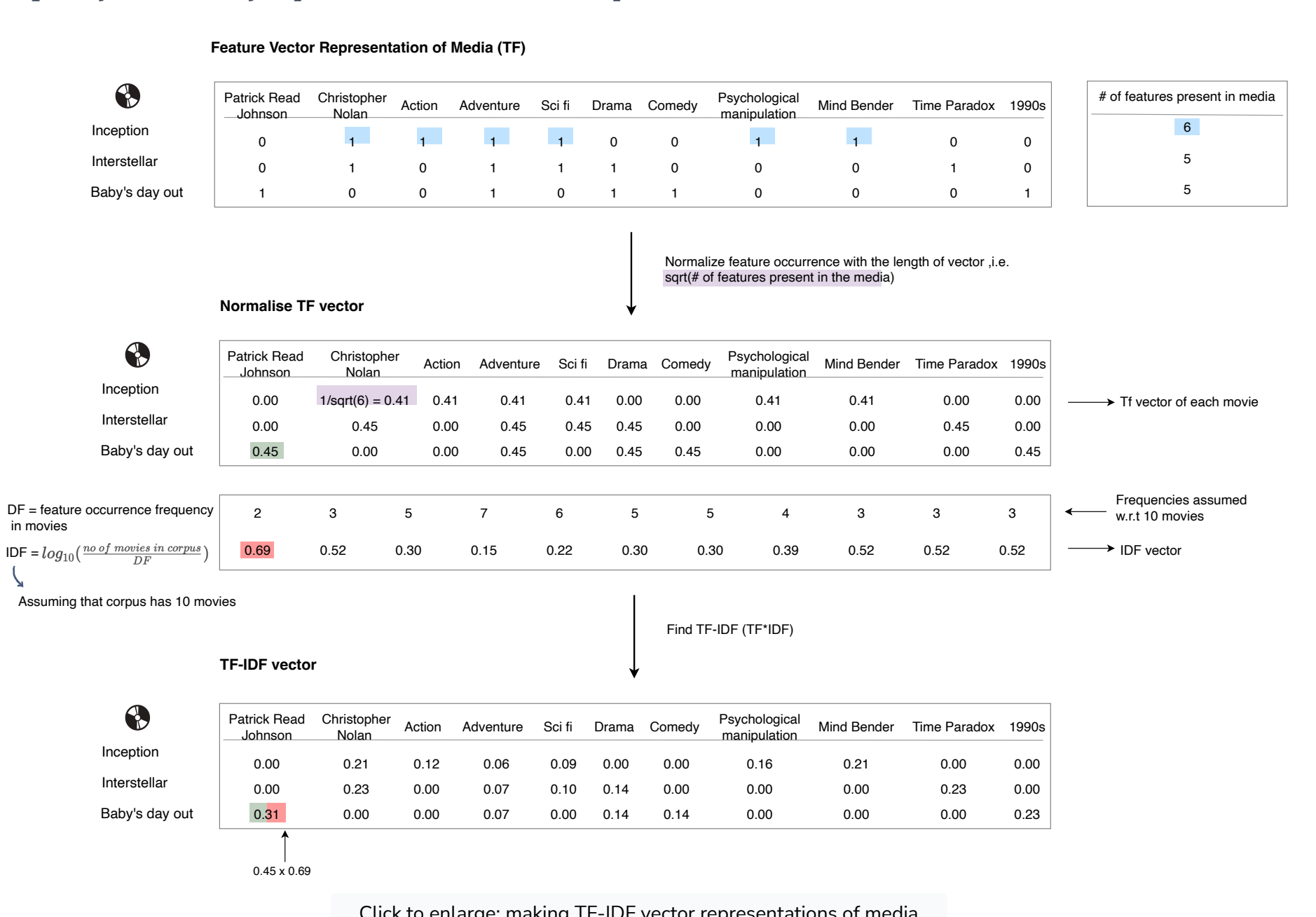
Collaborative filtering

As such the recommendations tend to be relevant to users' interest. The characteristics come from metadata (e.g., genre, movie cast, synopsis, director, etc.) information and manually assigned media-descriptive-tags (e.g., visually striking, nostalgic, magical creatures, character development, winter season, quirky indie rom-com set in Oregon, etc.) by Netflix taggers. The media is represented as a vector of its attributes. The following explanation takes a subset of the attributes for ease of understanding.



Media represented as a vector of attributes

Initially, you have the media's attributes in raw form. They need to be preprocessed accordingly to extract features. For instance, you need to remove stop words and convert attribute values to lowercase to avoid duplication. You also have to join the director's first name and last name to identify unique people since there maybe two directors with the first name Christopher. Similar preprocessing is required for the tags. After this, you are able to represent the movies as a vector of attributes with elements depicting the term frequency (TF) (binary representation of attribute's presence (1) or absence (0)).



Click to enlarge: making TF-IDF vector representations of media

Now, you have the movies represented as vectors containing the TF (term/attribute frequency) of all the attributes. This is followed by normalizing the term frequencies by the length of the vectors. Finally, you multiply the movies' normalized TF vectors and the IDF vector element-wise to make TF-IDF vectors for the movies.

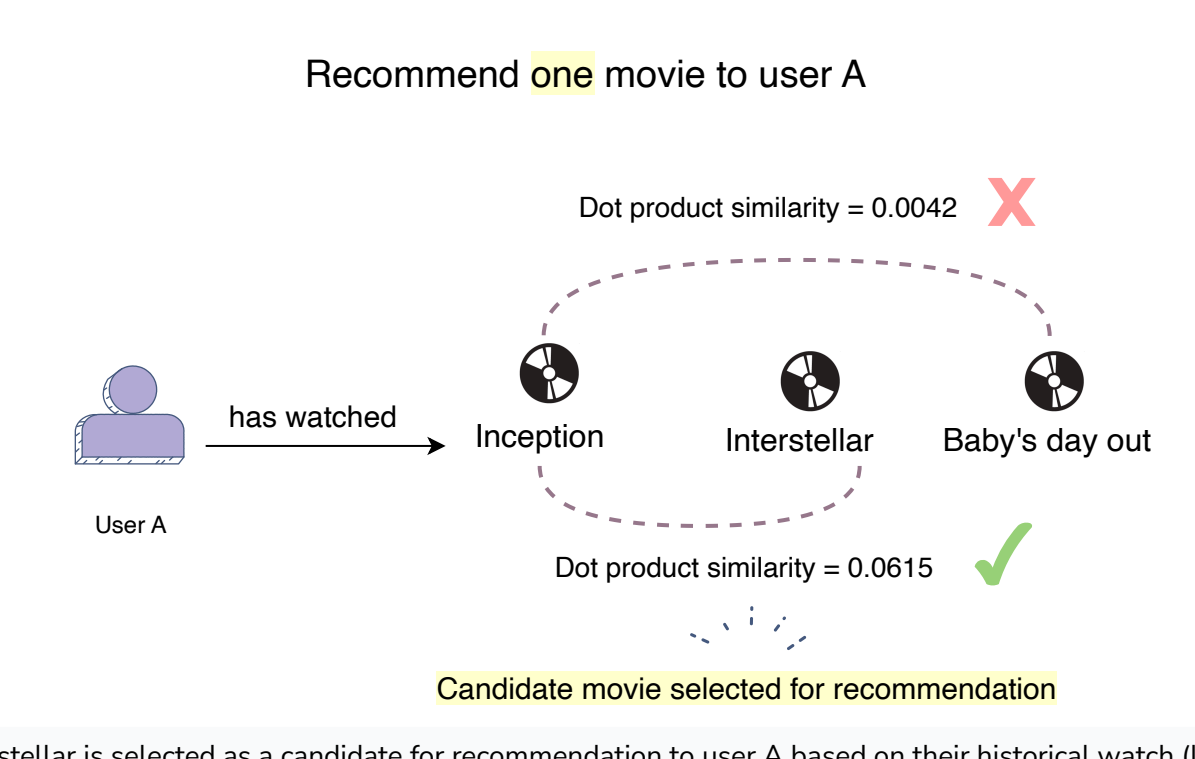
Minimize

TF-IDF puts the similarity between documents into perspective by considering the rarity of a feature throughout the corpus.

Given the TF-IDF representation of each movie/show, you have two options for recommending media to the user:

1. **Similarity with historical interactions**

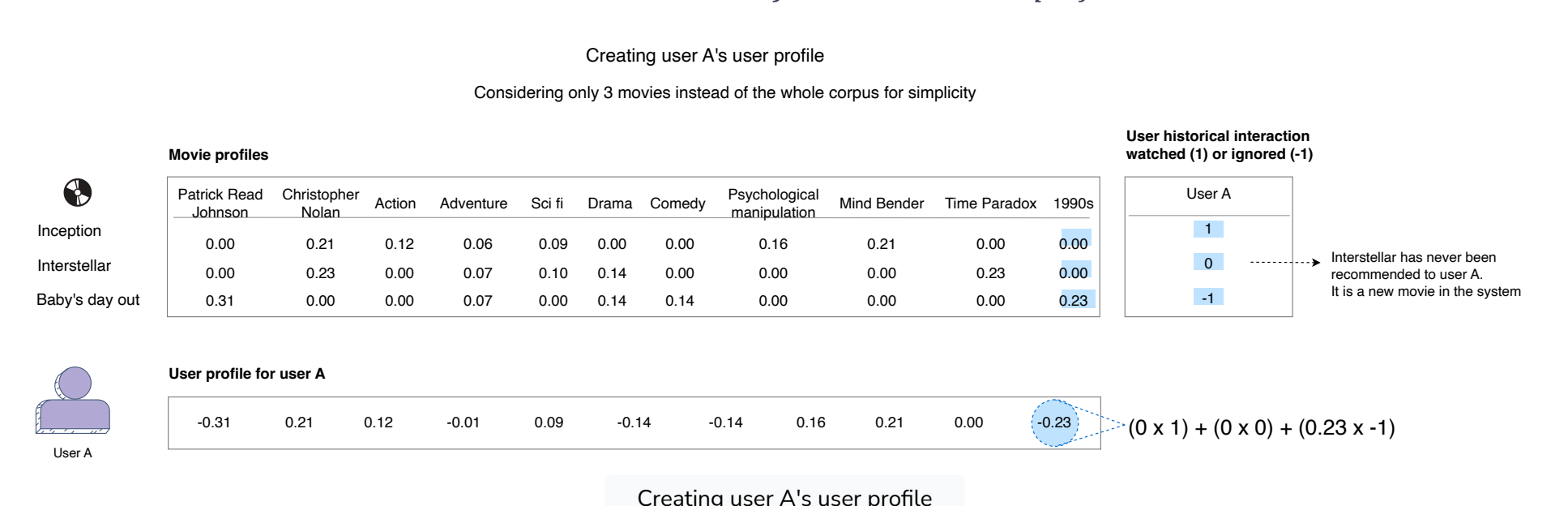
   You can recommend movies to the user similar to those they have interacted (seen) with in the past. This can be achieved by computing the dot product of movies.



Interstellar is selected as a candidate for recommendation to user A based on their historical watch (Inception)

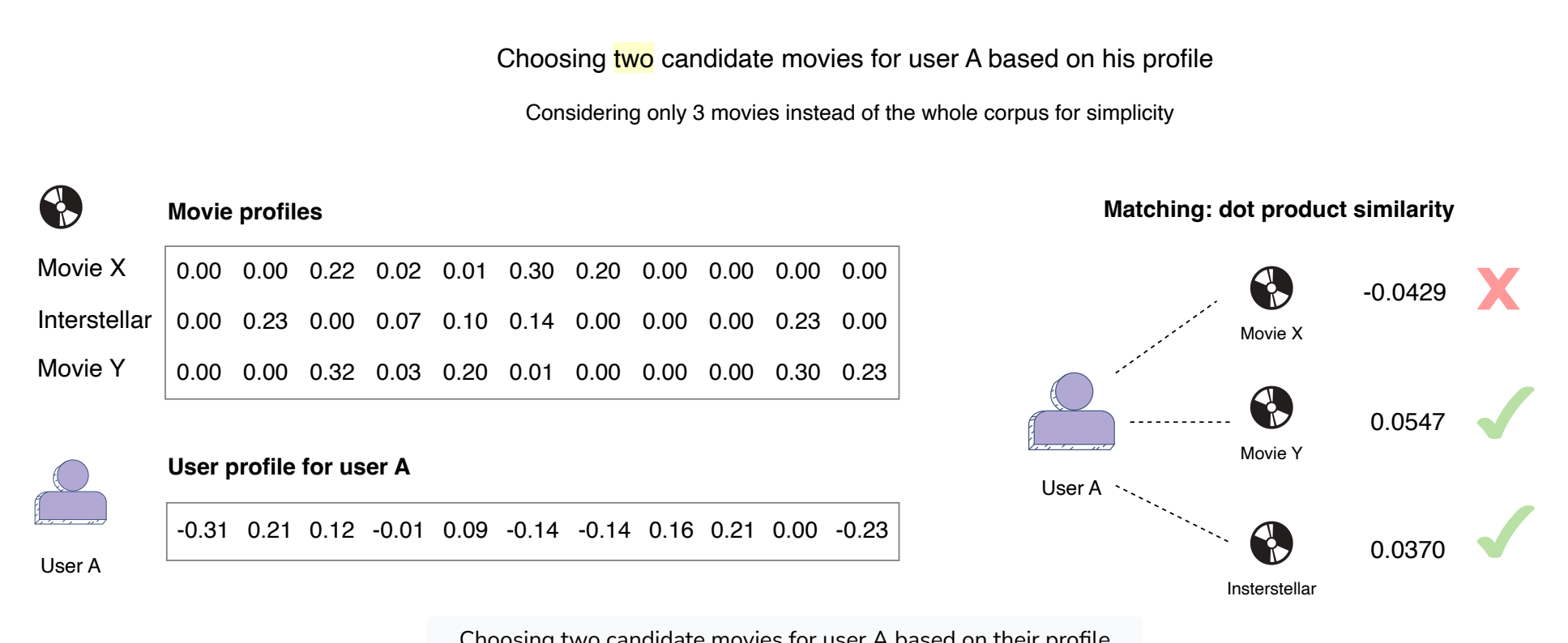
2. **Similarity between media and user profiles**

   The media's TF-IDF vectors can be viewed as their *profile*. Based on user preferences, which can be seen from its historical interactions with media, you can build *user profiles* as well as shown below.



Creating user A's user profile

Now, instead of using past watches to recommend new ones, you can just compute the similarity (dot product) between the user's profile and media profiles of unseen movies to generate relevant candidates for user A's recommendations.



Choosing two candidate movies for user A based on his profile

Generate embedding using neural networks/deep learning

Given the historical feedback (u,m), i.e., user u's feedback for movie m, you can use the power of deep learning to generate latent vectors/embeddings to represent both movies and users. Once you have generated the vectors, you will utilize KNN (k nearest neighbours) to find the movies that you would want to recommend to the user. This method is similar to generating latent vectors, that you saw in matrix factorization but is much more powerful because using NNs allows us to use all the sparse and dense features in the data to generate user and movie embedding.

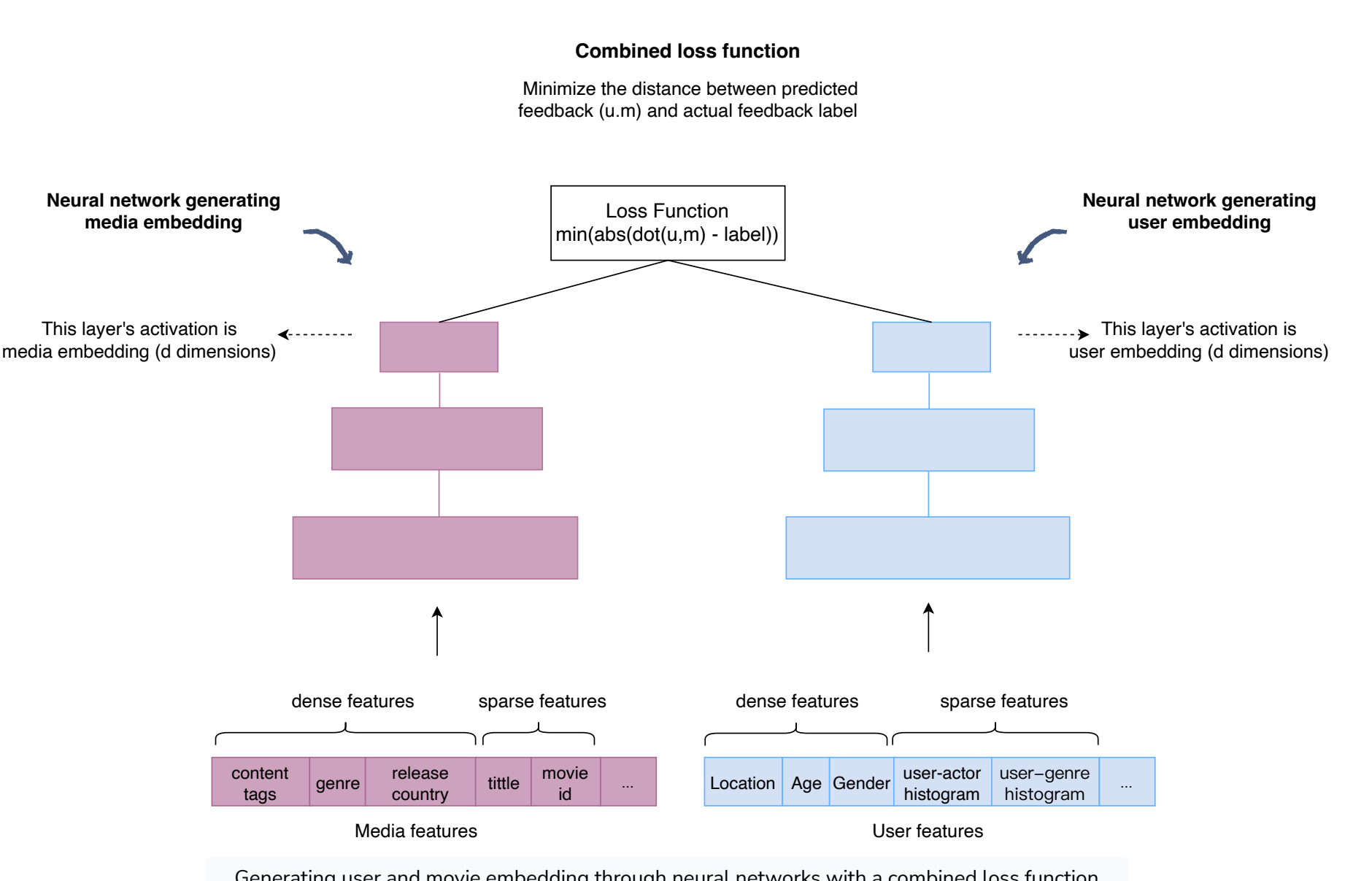
Embedding generation

You set up the network as two towers with one tower feeding in *media only* sparse and dense features and the other tower feeding in *user-only* sparse and dense features. The activation of the first tower's last layer will form the media's vector embedding *(m)*. Similarly, the activation of the second tower's last layer will form the user's vector embedding *(u)*. The combined optimization function at the top aims to minimize the distance between the dot product of *u* and *m* (predicted feedback) and the actual feedback label.

$$min(|dot(u,m) - label|) \ \ where \ u,m \in R^d, d = dimension \ of \ u \ and \ v's \ embedding$$

Let's look at the intuition behind this cost function. The actual feedback label will be positive when the media aligns with user preferences and negative when it does not. To make the predicted feedback follow the same pattern, the network learns the user and media embeddings in such a way that their distance will be minimized if the user would like this media and maximized if the user would not like the media.

> 📝 The vector representation of the user, which you would get as a result, will be nearest to the kind of movies that the user would like/watch.



Generating user and movie embedding through neural networks with a combined loss function

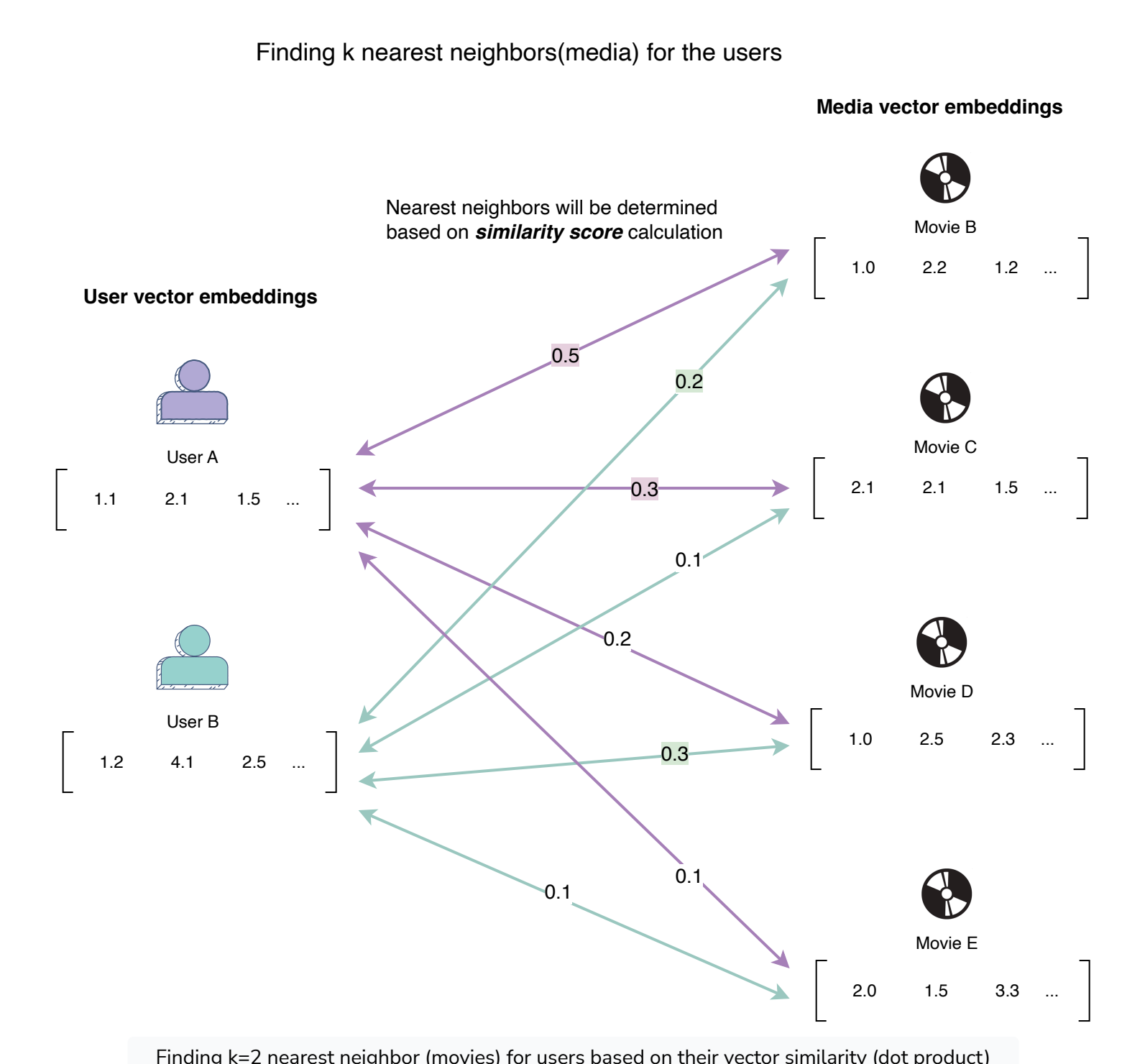
> 📝 The feature *user-actor histogram* shows the percentage of media watched by the user with particular actors (cast) in it. For instance, let's say 2% of the content viewed by the user had Brad Pitt in it, and 5% had Leonardo DiCaprio in it.

Minimize

For the above neural networks, the depth of the network and width (*d*) of the top/last layer are the hyperparameters.

Candidate selection (KNN)

Let's assume that you have to generate two candidates for each user, as shown in the diagram below. After the user and media vector embeddings have been generated, you will apply KNN and select candidates for each user. It can be observed in the diagram below, that user A's nearest neighbors are movie B and movie C, based on high *vector similarity*. Whereas, for user B, movie D and movie B are the nearest neighbors.



Finding k=2 nearest neighbor (movies) for users based on their vector similarity (dot product)

Techniques' strengths and weaknesses

Let's look at some strengths and weaknesses of the approaches for candidate generation discussed above.

Collaborative filtering can suggest candidates based solely on the historical interaction of the users. Unlike content-based filtering, it *does not require domain knowledge to create user and media profiles*. It may also be able to capture data aspects that are often elusive and difficult to profile using content-based filtering. However, collaborative filtering suffers from the *cold start problem*. It is difficult to find users similar to a new user in the system because they have less historical interaction. Also, new media can't be recommended immediately as no users have given feedback on it.

The **neural network technique** also suffers from the *cold start problem*. The embedding vectors of media and users are updated in the training process of the neural networks. However, if a movie is new or if a user is new, both would have fewer instances of feedback received and feedback given, respectively. By extension, this means there is a lack of sufficient training examples to update their embedding vectors accordingly. Hence, the cold start problem.

Content-based filtering is superior in such scenarios. It does require some initial input from the user regarding their preferences to start generating candidates, though. This input is obtained as a part of the onboarding process, where a new user is asked to share their preferences. *Once we have the initial input, it can create and then match the user's profile with media profiles*. Moreover, new medias' profiles can be built immediately as their description is provided manually.

← Back
Feature Engineering

Mark As Completed

Next →
Training Data Generation

Training Data Generation

Let's generate training data for the recommendation task with respect to implicit user feedback.

We'll cover the following

- Generating training examples
- Balancing positive and negative training examples
- Weighting training examples
- Train test split

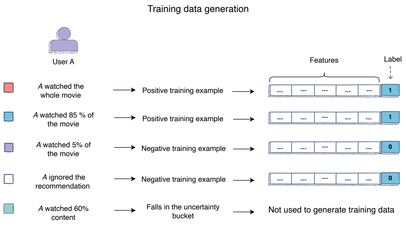
As mentioned previously, you will build your model on implicit feedback from the user. You will look at how a user interacts with media recommendations to generate positive and negative training examples.

Generating training examples

One way of interpreting *user actions* as positive and negative training examples is based on the duration for which the user watched a particular show/movie. You take positive examples as ones where the user ended up watching most of a recommended movie/show, i.e., watched 80% or more. You take negative examples, again where we are confident that the user ignored a movie/show, i.e., watched 10% or less.

If the percentage of a movie/show watched by the user falls between 10% and 80%, you will put it in the uncertainty bucket. This percentage is not clearly indicative of a user's like or dislike, so you ignore such examples. For instance, let's say a user watched 55% of a movie. This could be considered a positive example considering that they liked it enough to watch it midway. However, it could be that a lot of people had recommended it to them, so they wanted to see what all the hype was about by at least watching it halfway through. However, they, ultimately, decided that it was not according to their liking.

Hence, to avoid these kinds of misinterpretations, you label examples as positive and negative only when you are certain about it to a higher degree.

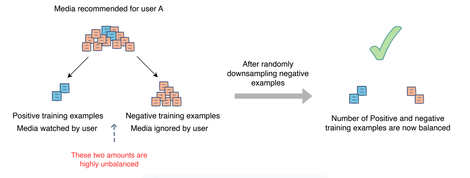


Positive and negative training examples being generated from user actions

Balancing positive and negative training examples

Each time a user logs in, Netflix provides *a lot of recommendations*. The user cannot watch all of them. Yes, people do binge-watch on Netflix, but still, this does not improve the positive to negative training examples ratio significantly. Therefore, you have a lot more negative training examples than positive ones. To balance the ratio of positive and negative training samples, you can **randomly downsample** the negative examples.

> We balance the negative and positive examples to prevent classifier from favouring the outcome that has more examples.

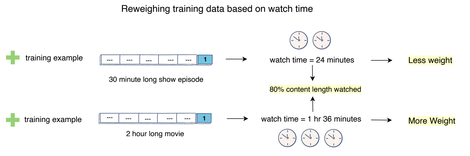


Balancing positive and negative training examples

Weighting training examples

Based on our training data discussion so far, all of the training examples are weighted equally, i.e., all have a weight of 1. According to Netflix's business objectives, the main goal could be to increase the time a user spends on the platform.

One way to incentivize your model to focus more on examples that have a higher contribution to the session watch time is to weight examples based on their contribution to session time. Here, you are assuming that your prediction model's optimization function utilizes weight per example in its objective.



In the diagram above, you have two positive training examples. The first is a thirty-minute long show episode while the second is a two-hour long movie. The user watches 80% of both media. For the show, this equals twenty-four minutes, but for the movie, it means one hour and thirty-six minutes. You would assign more weight to the second example than the first one so that your model learns which kinds of media increases the session watch time.

> One caveat of utilizing these weights is that the model might only recommend content with a longer watch time. So, it's important to choose weights such that we are not solely focused on watch time. We should find the right balance between user satisfaction and watch time, based on our online A/B experiments.

Train test split

Now, it's time to split your training data for training and then testing your models. While doing so, you need to be mindful of the fact that the user's interaction patterns may differ throughout the week. Hence, you will use the interaction with recommendations throughout a week to capture all of the patterns during model training.

You can randomly select $\frac{2}{3}^{rd}$, or 66.6%, of the training data rows generated and utilize them for training purposes. The rest of the $\frac{1}{3}^{rd}$, or 33.3%, can be used for validation and model testing.

However, this random splitting defeats the purpose of training the model on a whole week's data. Also, the data has a time dimension, i.e., you know the interaction on previous recommendations, and you want to predict the interaction with future recommendations. Hence, you will train the model on data from one time interval and validate it on the data from its succeeding time interval. This will give you a more accurate picture of how your model will perform in a real scenario.

> You are building models with the intent to forecast the future.

In the following illustration, you are training the model using data generated from the month of June and using data generated in the first and second week of July for validation and testing purposes.

Splitting data for training, validation, and testing

Even though Netflix has millions of subscribers, on average they may watch a maximum of 3 movies/1 show per week. Therefore, to generate a sufficient amount of data, you need to increase the time span over which you gather the data. Hence, we are using an entire month's data to train your model.

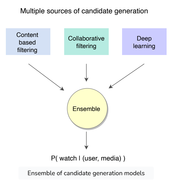
Ranking

Let's look at modeling options for the recommendation system.

We'll cover the following

The ranking model takes the top candidates from multiple sources of candidate generation that we have discussed. Then, an ensemble of all of these candidates is created, and the candidates are ranked with respect to the chance of the user watching that video content.

Here, your goal is to rank the content based on the probability of a user watching a media given a user and a candidate media, i.e., P(watch | (User, Media)).



Ensemble of candidate generation models

There are a few ways in which you can try to predict the probability of *watch*. It would make sense to first try a simplistic approach to see how far you can go and then apply complex modelling approaches to further optimize the system.

First, we will discuss some approaches using logistic regression or tree ensemble methods and then a deep learning model with dense and sparse features.

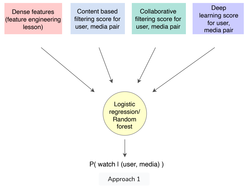
Deep learning should be able to learn through sparse features and outperform simplistic approach. Still, as we discussed in earlier problems, generalization with a deep NN model needs an order of magnitude more data (order of 100 of millions of training examples) and training capacity/time (100x more CPU cycles).

Approach 1: Logistic regression or random forest

There are multiple reasons that training a simplistic model might be the way to go. They are as follows:

- Training data is limited
- You have limited training and model evaluation capacity
- You want model explainability to really understand how the ML model is making its decision and show that to the end-user
- You require an initial baseline to see how far you can go in reducing our test set loss before you try more complex approaches

Along with other important features we discussed in the feature engineering section, output scores from different candidate selection algorithms are also a fairly important input consumed by the ranking models.



Approach 1

It is critical to minimize the test error and choose hyperparameters for training and regularization that gives us the best result on the test data.

Approach 2: Deep NN with sparse and dense features

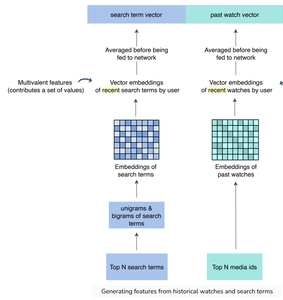
Another way to model this problem is to set up a deep NN. Some of the factors that were discussed in Approach 1 are now key requirements for training this deep NN model. They are as follows:

- Hundreds of millions of training examples should be available
- Having the capacity to evaluate these models in terms of capacity and model interpretability is not that critical.

It's important to call out that given the scale of Netflix, fulfilling the above requirements should not be a problem. Utilizing large scale data will definitely be able to outperform simplistic approaches discussed earlier.

Since the idea is that you want to predict whether the user will watch the media or not, you train a deep NN with sparse and dense features for this learning task. Two extremely powerful sparse features fed into such a network can be videos that the user has previously watched and the user's search terms. For these sparse features, you can set up the network to also learn media and search term embeddings as part of the learning task. These specialized embeddings for historical watches and search terms can be very powerful in predicting the *next watch idea* for a user. They will allow the model to personalize the recommendation ranking based on the user's recent interaction with media content on the platform.

An important aspect here is that both search terms and historical watched content are list-wise features. You need to think about how to feed them in the network given that the size of the layers is fixed. You can use an approach similar to pooling layers in CNN (convolution neural networks) and simply average the historical watch id and search text term embeddings before feeding it into the network.



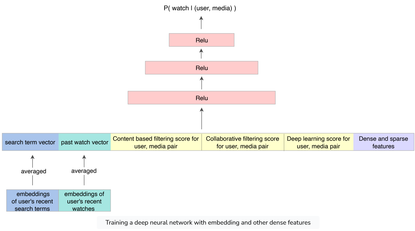
Generating features from historical watches and search terms

Minimize

User's searches or watches which are out of vocabulary are mapped as zero embeddings (a vector of all zeroes).

> vocabularies are unique ID spaces.

One way to set up the network is shown below utilizing these sparse data embeddings and feeding them into the Neural network.



Training a deep neural network with embedding and other dense features

As shown above, you can set it up as multiple RELU units layered on top of the embeddings that you learned along with other features. The top layer will predict whether the media will be watched or not using logistic loss.

Network structure

How many layers should you setup? How many activation units should be used in each layer? The best answer to these questions is that you should start with 2-3 hidden layers with a RELU based activation unit and then play around with the numbers to see how this helps us reduce the test error. Generally, adding more layers and units helps initially, but its usefulness tapers off quickly. The computation and time cost would be higher relative to the drop in error rate.

Re-ranking

The top ten recommendations on the user's page are of great importance. After your system has given the watch probabilities and you have ranked the results accordingly, you may re-rank the results.

Re-ranking is done for various reasons, such as bringing diversity to the recommendations. Consider a scenario where all the top ten recommended movies are comedy. You might decide to keep only two of each genre in the top ten recommendations. This way, you would have five different genres for the user in the top recommendations.

If you are also considering past watches for the media recommendations, then re-ranking can help you. It prevents the recommendation list from being overwhelmed by previous watches by moving some previously watched media down the list of recommendations.

Problem Statement

Let's learn how a self-driving car can "see" its surroundings using semantic image segmentation.

We'll cover the following

Introduction

The definition of a self-driving car is a vehicle that drives itself, with little or no human intervention. Its system uses several sensory receptors to perceive the environment. For instance, it identifies the drivable area, weather conditions, obstacles ahead and plans the next move for the vehicle accordingly.

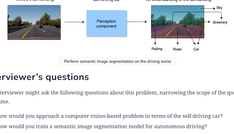
There are different levels of autonomy in such vehicles. Tesla currently implements assisted driving, where the driving is autonomous, but someone is behind the wheel. Waymo (Google's self-driving car), in contrast, is aiming for complete autonomy under all driving conditions (no driver required). Beyond human transportation, self-driving vehicles can also be utilized as a service for various purposes, e.g., Nuro is building self-driving vehicles for local goods transportation.

> 📝 Self-driving vehicles are perfect real-world systems for handling multi-sensory inputs that focus primarily on computer vision-based problems (e.g., object classification/ detection/ segmentation), using machine learning.

Now that you have some context, let's look at the problem statement.

Problem statement

The interviewer has asked you to **design a self-driving car system focusing on its perception component** (*semantic image segmentation in particular*). This component will allow the vehicle to perceive its environment and make informed driving decisions. Don't worry we will describe semantic image segmentation shortly.



Perform semantic image segmentation on the driving scene

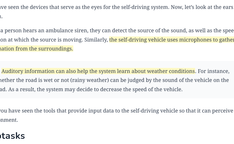
Interviewer's questions

The interviewer might ask the following questions about this problem, narrowing the scope of the question each time.

1. How would you approach a computer vision-based problem in terms of the self-driving car?
2. How would you train a semantic image segmentation model for autonomous driving?
3. How will the segmentation model fit in the overall autonomous driving system architecture?
4. How would you deal with data scarcity in imaging dataset?
5. How would you best apply data augmentation on images?
6. What are the best model architectures for image segmentation tasks?
7. Your optimized deep learning model gives a high performance on the validation set, but it fails when you take the self-driving car on the road. Why? How would you solve this issue?

> 📝 Answers
> 01 will be answered shortly when we talk about *subtasks*. 02 and 03 will be answered in the *architectural components* lesson. 04, 05 and 07 will be answered in the *training data generation* lesson. 06 will be answered in the *modelling* lesson.
>
> 📝 Any other similar (multi-sensory) computer vision problem asked during the interview can be solved in a similar manner to the self-driving car problem.

Let's have a quick look at some of the major software and hardware components of the self-driving vehicle system before you attempt to answer the first question.



Semantic image segmentation of vehicle surroundings

Hardware support

Let's look at the *sensory receptors* that allow the vehicle to "see" and "hear" its environment and plan its course of action accordingly.

- Camera

The camera provides the system with high-resolution visual information of its surrounding. The visual input from a frame-based camera can also be used to get an estimate of the depth/distance of objects from the self-driving car. However, the usefulness of the camera depends heavily on the lighting conditions and may not work optimally in the night scenarios.

- Radar

The radar uses radio waves' reflections from solid objects to detect their presence. It aids depth on top of the visual information so that the system can accurately sense the distance between various objects. It works well in varied conditions, like low light, dirt and cloudy weather, as well as long operating distances.

- Lidar

Lidar, in contrast, uses laser light reflecting off objects to create a precise 3D image of the environment, while measuring the distance at which objects are positioned. They are good at even detecting *small objects*. However, they do not work well during *foggy weather* and are quite *expensive*.

> 📝 One approach that exists in the industry is to get input for the system using a fusion of these sources to overcome their shortcomings.

The following diagram provides a good comparison of each type of hardware support.

Image source: https://www.aurolight.tech/technology/fused-sensing-vs.-sensor-fusion

- Microphones

You have seen the devices that serve as the eyes for the self-driving system. Now, let's look at the ears of the system.

When a person hears an ambulance siren, they can detect the source of the sound, as well as the speed and direction at which the source is moving. Similarly, the self-driving vehicle uses microphones to gather audio information from the surroundings.

> 📝 Auditory information can also help the system learn about weather conditions. For instance, whether the road is wet or not (rainy weather) can be judged by the sound of the vehicle on the road. As a result, the system may decide to decrease the speed of the vehicle.

Now, you have seen the tools that provide input data to the self-driving vehicle so that it can perceive its environment.

Subtasks

The pipeline from environment perception to planning the movement of the vehicle can be split into several machine learning subtasks which we will describe shortly.

In the field of computer vision, the first three subtasks represent approaches that can be adapted to deal with imaging datasets (*driving images in your case*), e.g., classification, localization, segmentation, etc. Using an example, let's break this down to understand how to approach a problem statement of this scale.

You start in the morning, and your job is to take the self-driving vehicle from Boston to New York City. Imagine a self-driving vehicle on a highway in daylight with a clear sky and less traffic. If you train an *image classifier* on the labeled data, then feeding the input camera frames in real-time would probably generate the average classification scores like this: {"car":0.1, "road":0.5, "sky":0.3, "trees":0.05, "misc":0.05}.

This score provides a decent estimate of the overall categories of objects present in the surroundings. However, it does not help locate these objects. This can be solved through an *image localizer*, which will generate bounding boxes, i.e., locations on top of the predicted objects. This leads you to your first subtask, which is *object detection*.

1. Object detection

   In object detection, we detect instances of different class objects (e.g., greenery, fences, cars, roads, and mountains) in the surroundings and localize them by drawing bounding boxes.

Object detection: Different objects are identified and localized with the help of bounding boxes, as indicated in different colors

Continuing with the example, let's complicate the problem. You enter New York City around noon and encounter medium traffic. Your image localizer will generate a lot of overlapping bounding boxes. To resolve this, you need to upgrade the system to segment the object categories to find the optimal available path for the vehicle. You build an *image segmenter* that can generate the predicted binary mask/category on top of bounding boxes and classifications through semantic segmentation. This leads to the second subtask, i.e., semantic segmentation.

2. Semantic segmentation

   Semantic segmentation can be viewed as a pixel-wise classification of an image. Objects of the same class are *assigned the same label* as shown in the following diagram.

Semantic segmentation

Let's further complicate the problem. Now, you need to navigate a self-driving vehicle in a very busy street (e.g., Times Square) in Manhattan. In this case, you need to segment individual objects in real-time with high precision: You upgrade the system from semantic to instance-based segmentation, which will generate a predictive binary mask/object on top of the bounding boxes and classifications. This leads us to the third subtask, i.e., *instance segmentation*.

3. Instance segmentation

   Semantic segmentation does not differentiate between different instances of the same class. Instance segmentation, however, combines object detection and segmentation to classify the pixels of each instance of an object. It first detects an object (a particular instance) and then classifies its pixels. Its output is as follows:

Instance segmentation: Makes a distinction between the boundary of different instances of the same class

4. Scene understanding

   Here, you try to understand what is happening in the surroundings. For instance, you may find out that a person is walking towards us, and you need to apply brakes. Or, as in the following diagram, we see that a car is going ahead of us on the highway.

Car ahead on the highway

Scene understanding: given an input image, the output is a text-based command: "Car ahead on the highway"

5. Movement plan

   After you have identified the objects, segmented unique objects in the image and understood the scenario, it's time to decide the movement plan for the self-driving vehicle. For example, you might decide to slow down (apply brake) due to a car ahead.

Instruments for executing movement plan for the self-driving car

In this chapter, the focus will be the first two subtasks, i.e., the following machine learning problem:

"Perform semantic segmentation of the self-driving vehicle's surrounding environment."

Metrics

Let's explore some metrics that will help evaluate the performance of the vision-based self-driving car system.

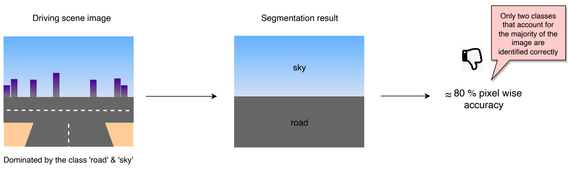
We'll cover the following

Component level metric

The component of the self-driving car system under discussion here is the semantic segmentation of objects in the input image. In order to look for a suitable metric to measure the performance of an image segmenter, the first notion that comes to mind is the *pixel-wise accuracy*. Using this metric, you can simply compare the ground truth segmentation with the model's predictive segmentation at a pixel level. However, this might not be the best idea, e.g., consider a scenario where the driving scene image has a major class imbalance, i.e., it mostly consists of sky and road.

> For this example, assume one-hundred pixels (ground truth) in the driving scene input image and the annotated distribution of these pixels by a human expert is as follows: sky=45, road=35, building=10, roadside=10.

If your model correctly classifies all the pixels of only sky and road (i.e., sky=50, road=50), it will result in high pixel-wise accuracy. However, this is not really indicative of good performance since the segmenter completely misses other classes such as building and roadside!



High pixel-wise accuracy doesn't always translate to a great performance by the segmentation model

In general, you need a higher pixel-wise accuracy for objects belonging to each class as an output from the segmenter. The following metric caters to this requirement nicely.

IoU

Intersection over Union (IoU) divides the overlapping area between the predicted segmentation and the ground truth in perspective, by the area of union between the predicted segmentation and the ground truth.

$IoU = \frac{area\ of\ overlap}{area\ of\ union}$ or $IoU = \frac{|P_{pred} \cap P_{gt}|}{|P_{pred} \cup P_{gt}|}$

This metric ranges from 0–1 (or 0%–100%). '0' indicates no overlap while '1' indicates perfectly overlapping segmentation.

The driving images contain objects of multiple classes, as shown above (e.g., building, roadside, sky, road, etc.). So, you will be performing multi-class segmentation, for which the **mean IoU** is calculated by taking the *average of the IoU for each class*.

Now, let's calculate the mean IoU and see how it differs from pixel accuracy. You will be considering the same driving scene image and its predicted segmentation, as shown above.

You will begin by calculating the IoU for each class. Here, the "area of overlap" means the number of pixels that belong to the particular class in both the prediction and ground-truth. Whereas, the "area of union" refers to the number of pixels that belong to the particular class in the prediction and in ground-truth, but not in both (the overlap is subtracted).

> Let's apply the calculations: ground truth: [sky=45, road=35, building=10, roadside=10], segmenter predictions: [sky=50, road=50, building=0, roadside=0].

$$IoU_C = \frac{P_{pred} \cap P_{gt}}{(P_{pred}+P_{gt})-(P_{pred} \cap P_{gt})} \text{ where } P_{pred} = number\ of\ pixels\ classified\ as\ class\ C\ in\ prediction \text{ and } P_{gt} = number\ of\ pixels\ classified\ as$$

$$IoU_{sky} = \frac{45}{(50+45)-45} = 90\%$$

$$IoU_{road} = \frac{35}{(50+35)-35} = 70\%$$

$$IoU_{building} = \frac{0}{(0+10)-0} = 0\%$$

$$IoU_{road\ side} = \frac{0}{(0+10)-0} = 0\%$$

$$Mean\ IoU = \frac{IoU_{road}+IoU_{sky}+IoU_{road\ side}+IoU_{building}}{4} = \frac{90+70+0+0}{4} = 40\%$$

You can see that Mean IoU (40%) is significantly lower than the pixel-wise accuracy (80%) and provides an accurate picture of how well your segmentation is performing.

> You will be using IoU as an offline metric to test the performance of the segmentation model.

End-to-end metric

You also require an online, end-to-end metric to test the overall performance of the self-driving car system as you plug in your new image segmenter to see its effect.

Manual intervention

Ideally, you want the system to be as close to self-driving as possible, where the person never has to intervene and take control of the driving. So, you can use manual intervention as a metric to judge the success of the overall system. If a person rarely has to intervene, it means that your system is performing well.

> This is a good metric for the early testing phase of the self-driving car system. During this time you have a person ready to take over in case the self-driving system makes a poor decision.

Simulation errors

Another approach is to use historical data, such as driving scene recording, where an expert driver was driving the car. You will give the historical data as input to your self-driving car system with the new segmentation model and see how its decisions align with the decisions made by an expert driver.

You will assume that the decisions made by the professional driver in that actual scenario are your ground truths. The overall objective will be to minimize the movement and planning errors with these ground truths.

As you replay the data with new segmenter, you will experiment to measure whether your segmentation is resulting in a reduction of your simulation-based errors or not. This will be a good end to end metric to track.

Architectural Components

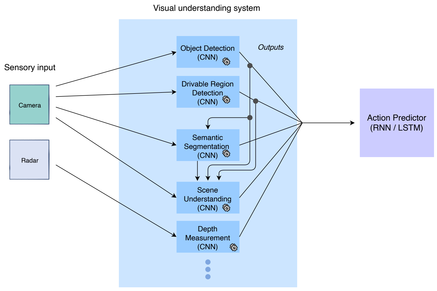
Let's take a look at the architectural components of the self-driving car system.

We'll cover the following

- Overall architecture for self-driving vehicle
- System architecture for semantic image segmentation

Overall architecture for self-driving vehicle

Let's discuss a simplified, high-level architecture for building a self-driving car. Our discussion entails the important learning problems to be solved and how different learning models can fit together, as shown below.



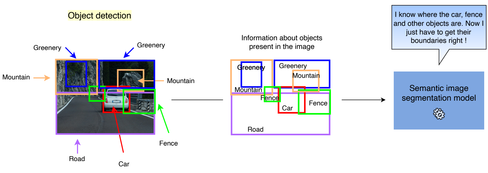
High-level architecture (CNN: Convolutional neural network, RNN: Recurrent neural network, LSTM: Long short-term memory)

The system is designed to receive sensory inputs via cameras and radars, which are fed to the *visual understanding system* consisting of different convolutional neural networks (CNN), each for a specific subtask. The output of the *visual understanding system* is used by the *action predictor RNN or LSTM*. Based on the visual understanding of the environment, this component will plan the next move of the vehicle. The next move will be a combination of outcomes, i.e., applying brakes, accelerating, and/or steering the vehicle.

> We won't be discussing input through Lidar here. However, it can also be used for scene analysis similar to a camera, especially for reconstructing a 3D view of an environment.

Now, let's have a look inside the visual understanding system. The object detection CNN detects and localizes all the obstacles and entities (e.g., humans and other vehicles) in the vehicle's environment. This is essential information because the action predictor RNN may predict to slow down the vehicle due to the impending obstacle (e.g., a person crossing the road). However, the most crucial information for the action predictor RNN is information that allows it to extract a drivable path for the vehicle. Therefore, due to the significance of this task, we will train a separate model for this purpose: the drivable region detection CNN. It will be trained to detect the road lanes to help the system decide whether the pathway for the vehicle is clear or not.

Moreover, as the object detection CNN identifies the key objects in the image (predicts bounding boxes), it is further required to share its output along with the raw pixel data for *semantic image segmentation* (i.e., draw pixel-wise boundaries around the objects). These boundaries will help in navigating the autonomous vehicle in a complex environment where overlapping objects/obstacles are presented.



Object detection model's output is fed into the segmentation model

Machine learning models used in the components

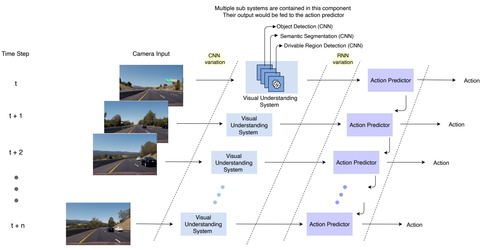
Many of the subtasks in the visual understanding component are carried out via specialized CNN models that are suited for that particular subtask.

The action predictor component, on the other hand, needs to make a movement decision based on:

1. Outputs of all the visual understanding sub-tasks
2. Track/record of the vehicle's movements based on previous scene understanding

This can be best learned through a recurrent neural network (RNN) or long short-term memory (LSTM) that can utilize the temporal features of the data, i.e., previous and current predictions from the scene segmentation as inputs.

Let's dive a little deeper into how this is happens, focusing on the input from the camera.



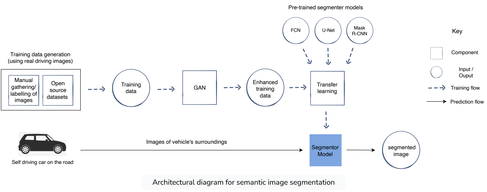
How the self-driving vehicle architecture works in a block diagram

You will receive the video frames covering the vehicle's surroundings from the camera as input. This video is nothing more than a sequence of images (frames per second). The image at each time step (t, t+1,…) will be fed to the visual understanding system. The multiple outputs of this component will form the input to the self-driving vehicle's action predictor. The action predictor will also use the previous time step information (t-1) while predicting the action for the current time step (t).

> RNN will utilize the past information to make "informed" predictions in real-time.

System architecture for semantic image segmentation

You just saw how the semantic image segmentation task fits in the overall architecture for the self-driving vehicle system architecture. Now, let's zoom in on the architecture for the semantic image segmentation task.



Architectural diagram for semantic image segmentation

The above diagram shows the training flow and prediction flow for the semantic image segmentation system architecture.

The training flow begins with training data generation, which makes use of two techniques. In the first technique, real-time driving images are captured with the help of a camera and are manually given pixel-wise labels by human annotators. The latter technique simply uses open-source datasets of self-driving vehicle images. This training data is then enhanced or augmented with the help of generative adversarial networks (GANs). Collectively all the training data is then used to train the segmenter model. Transfer learning is applied with the segmenter model to utilise powerful feature detectors from pre-trained models (FCN, U-Net, Mask R-CNN). The pre-trained models are optimized on your datasets to get the final model.

In the prediction flow, your self-driving vehicle would be on the road. You would receive real-time images of its surroundings, which would be given to the segmenter model for semantic image segmentation.

Training Data Generation

Let's generate training data for the semantic image segmentation model.

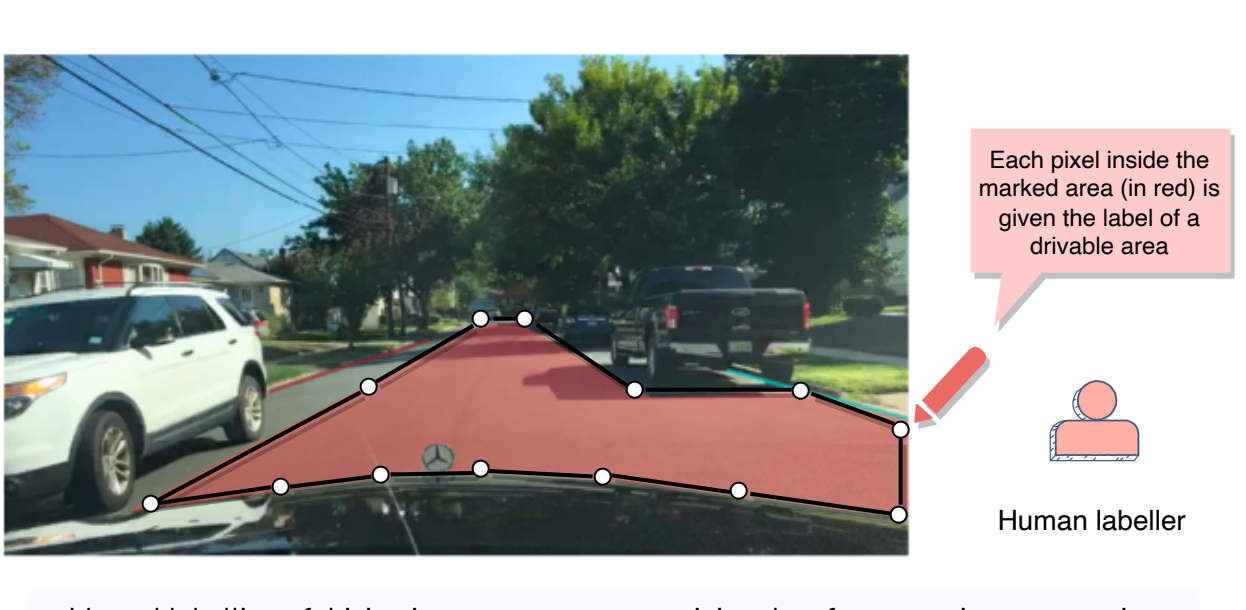
We'll cover the following

Generating training examples

Autonomous driving systems have to be extremely robust with no margin of error. This requires training each component model with all the possible scenarios that can happen in real life. Let's see how to generate such training data for performing semantic segmentation.

Human-labeled data

First, you will have to gather driving images and hire people to label the images in a pixel-wise manner. There are many tools available such as Label Box that can facilitate the pixel-wise annotation of images.



Manual labelling of driving images to generate training data for semantic segmentation

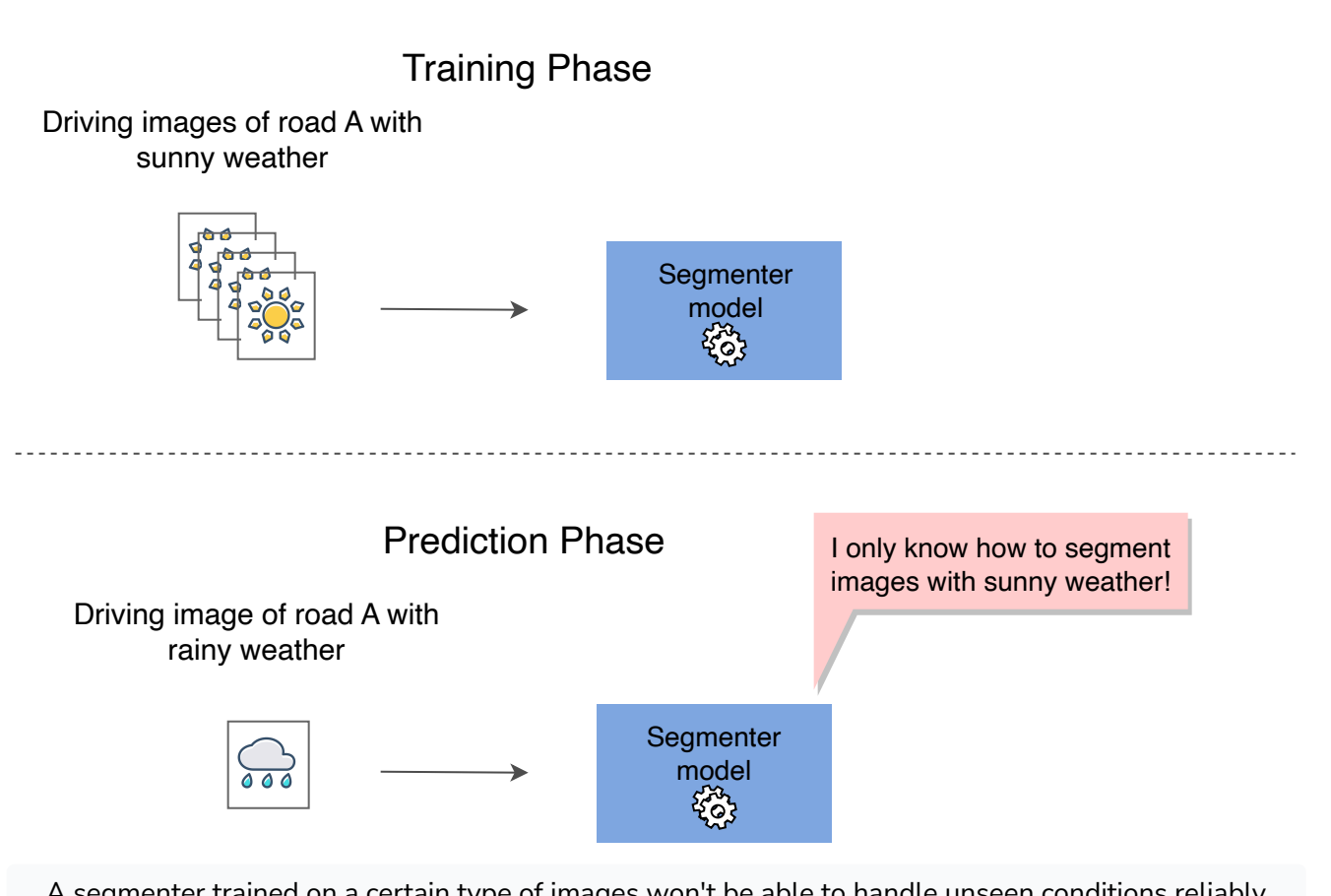
All the other parts of the image are also labelled similarly.

Open source datasets

There are quite a few open-source datasets available with segmented driving videos/images. One such example is the "BDD100K: A Large-scale Diverse Driving Video Database". It contains segmented data for full frames, as shown with an example below.

BDD100K: full frame segmentation

These two methods are effective in generating manually labeled data. In most cases, your training data distribution will match with what you observe in the *real-world scene* images. However, you may not have enough data for all the **conditions** that you would like your model to make predictions for such as snow, rain, and night. For a self-autonomous vehicle to be perfect, your segmenter should work well for all the possible weather conditions, as well as cover a variety of obstacle images in the road.



A segmenter trained on a certain type of images won't be able to handle unseen conditions reliably

The sunny vs rainy condition is just one example; there can be many such situations.

It is an important area to think about how you can give your model *training data* for all conditions. One option is to manually label a lot of examples for each scenario. The second option is to use powerful data augmentation techniques to generate new training examples given your human-labeled data, especially for conditions that are missing in your training set. Let's discuss the second approach, using generative adversarial networks (GANs).

Training data enhancement through GANs

In the big picture, your self-driving system should compete with human intelligence when it comes to making decisions and planning movements. The segmenter can play its role in creating a reliable system by being able to accurately segment the driving scene in any condition that the vehicle experiences.

Achieving this target requires a diverse set of training data that covers all the possible permutations of the driving scene. For instance, assume that your dataset only contains ten-thousand driving images in the snowy Montreal conditions and fifty-thousand images in the sunny California conditions. You need to devise a way to *enhance your training data* by converting snowy conditions of the ten-thousand Montreal images to sunny conditions and vice versa.

The target includes two parts:

1. Generating new training images
2. Ensuring generated images have different conditions (e.g., weather and lighting conditions).

For the first part, you can utilize GANs. They are deep learning models used for *generation of new content*. However, for the second part, i.e., to generate a different variation of the original segmented image, a simple GAN would not work. You would have to use **conditional generative adversarial networks (cGANs)** instead. Let's see why.

GANs

Generative adversarial networks belong to the set of *generative models* in contrast to discriminative models, i.e., they focus on learning the unknown probability distribution of the input data to **generate** similar inputs rather than just classifying inputs as belonging to a certain class.

> 📝 For instance, given panda images, a generative model will try to understand how pandas look like (learn the probability distribution) rather than trying to accurately predict if an image is of a panda or another animal. This allows you to *generate new panda images*.

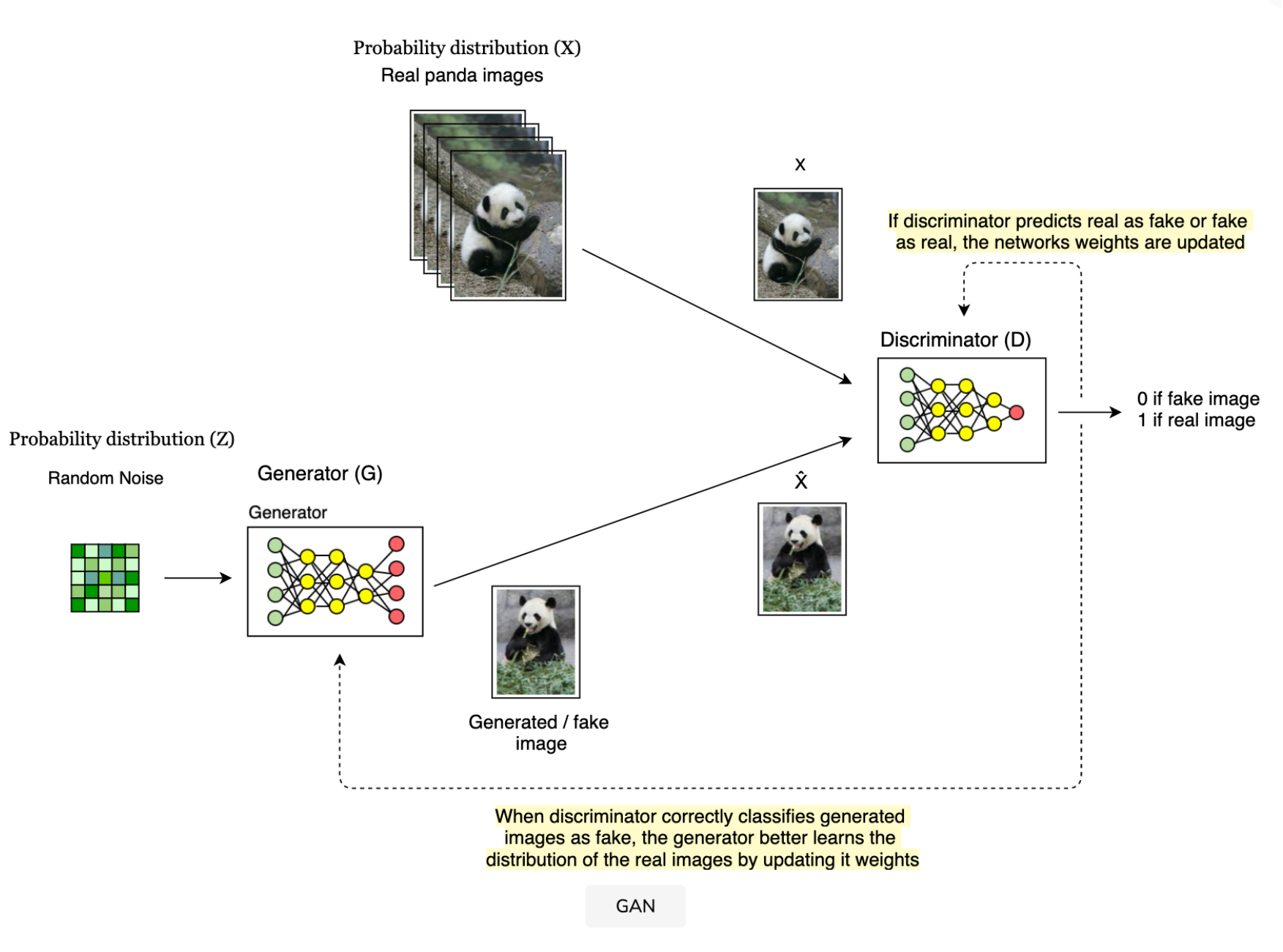
GANs are made of two deep learning models that compete against each other (hence, the term adversarial):

1. Generator (Forger)

   This neural network receives a sample from a random distribution Z (noise). It transforms it into something (a panda) that looks like it came from the target/true distribution(panda images). This is done by learning a mapping between the input samples and the images in the true distribution, during the training phase.

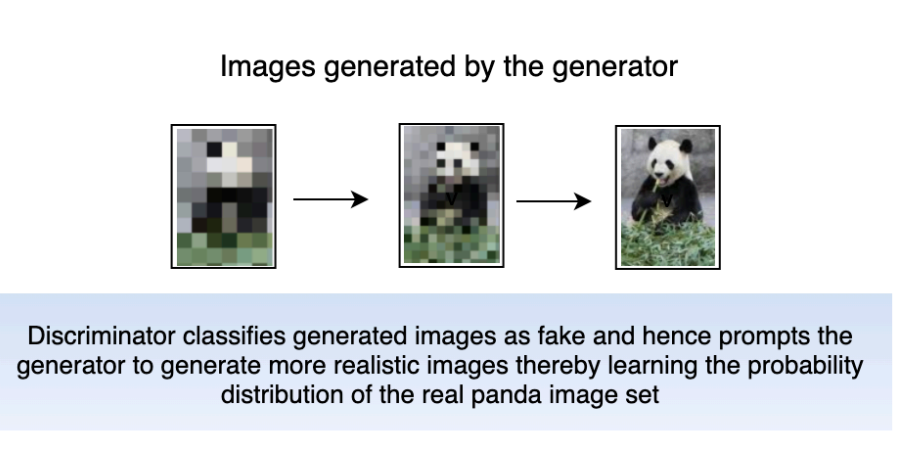
2. Discriminator (Detective)

   This neural network receives images from the true distribution X as well as the new/fake images $\hat{X}$ generated by the generator. The goal is to discriminate between real and generated images. The network is trained to output 0 for a generated image and 1 for a real image.



GAN

The generator's tries to fool the discriminator into thinking that its generated images belong to the true distribution. On the other hand, the discriminator tries to catch all the fake/generated images. This encourages the generator to learn the true probability distribution more closely and generate such real looking images that the discriminator's output converges to 0.5, i.e., it is not able to distinguish fake from real.



The generator's output improves throughout the training iterations.

Supervised or unsupervised learning?

You saw that the generator model of the GAN learns the target probability distribution (looks for patterns in the input data) to generate new content; this is known as *generative modeling*.

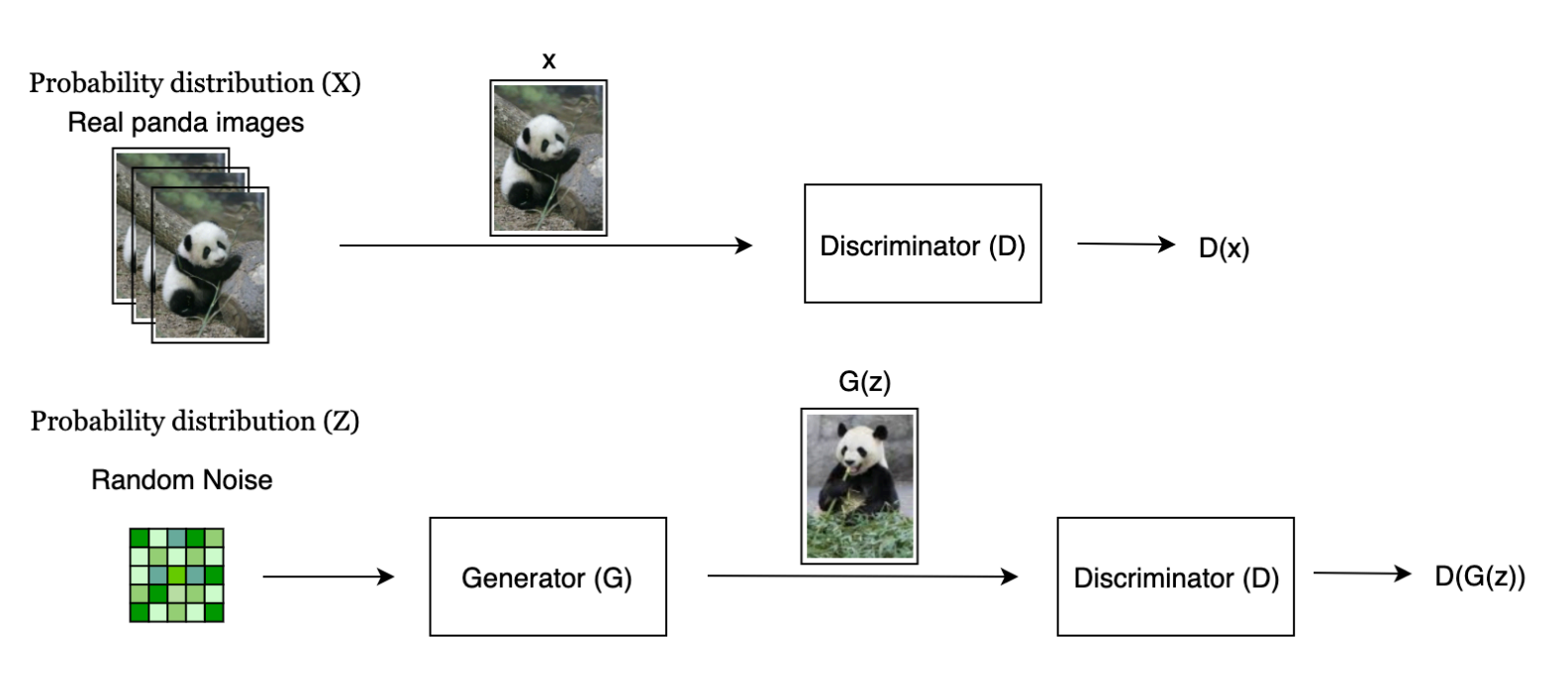
Generative modeling is an unsupervised task where the model is not told what kind of patterns to look for in the data and there is no error metric to improve the model. However, the *training process of the GAN is posed as a supervised learning problem* with the help of a discriminator. The discriminator is a supervised classifier that distinguishes between real and generated images. This allows us to devise a **supervised loss function for the GAN**.

Loss Function

The *binary cross entropy loss function* for the GAN is as follows:

$$min_G \ max_D \ \{E_{x \sim p_{data}(x)}[log\ D(x)] + E_{z \sim p_z(z)}[log(1 - D(G(z)))]\}$$

Its made up of two terms. Let's see what each part means (This is the combined loss for both discriminator and generator network).



- x is an image from the true distribution
- z is the random input to the generator
- $p_{data}(x)$ is the probability distribution of the real images.
- $p_z(z)$ is the probability distribution of the generated images.
- G(z) is the generator's output when given noise z.
- D(x) is the discriminator's probability (judgement) that the *real data instance x* is real.
- D(G(z)) is the discriminator's probability (judgement) that the *generated instance x* is real.
- E_x is the expected value over all real data instances.
- E_z is the expected value over random inputs to the generator.

> 📝 The expected value of a variable X is the average of all values of X weighted by their occurrence probability.

The GAN's objective is to replicate the p_{data} with p_z. Binary cross entropy helps to measure the distance between these two distributions. Let's see how this works.

The term *D(x)* is the probability that a data point belongs to *class real*, whereas the term *1-D(G(z))* is the probability of belonging to *class fake*. The loss function works by measuring how far away from the actual value (real 1 or fake 0) the prediction is for each class and then averages the errors (Expectation) to obtain the final loss.

Discriminator's aim

The discriminator wants to distinguish between real and fake. Earlier, you mentioned the final form of the loss. Let's see how the discriminator's loss function is derived from the original form of the binary cross-entropy loss as follows:

$$Loss(\hat{y}, y) = [y log \hat{y} + (1 - y) log(1 - \hat{y})]$$

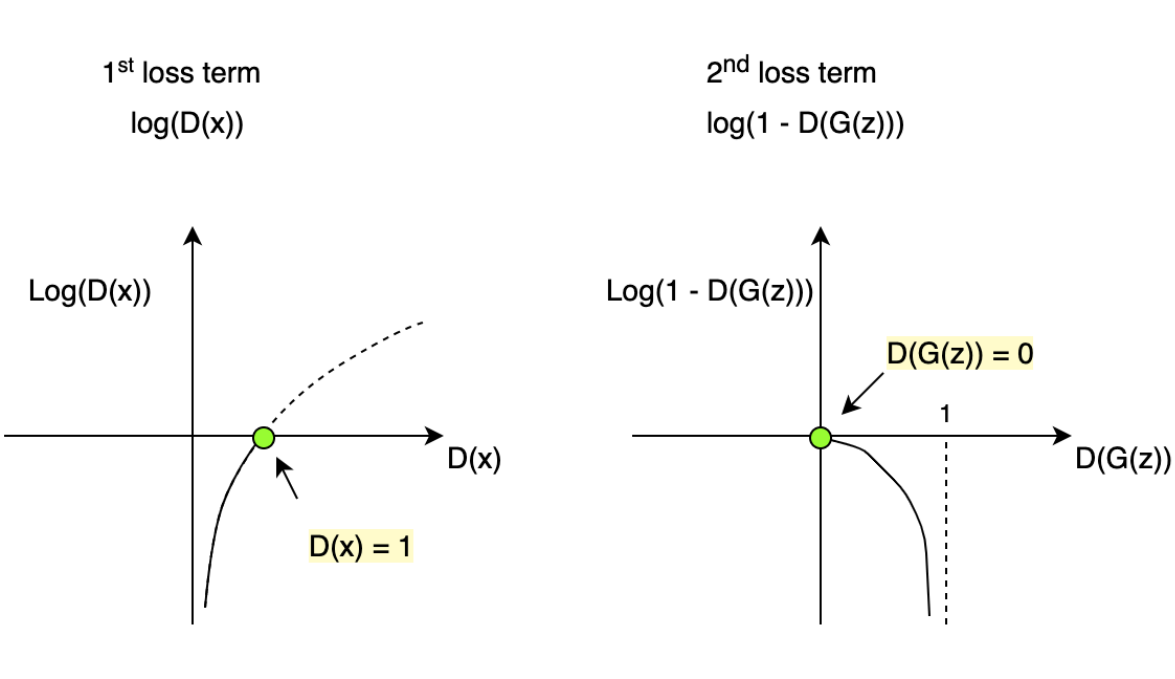
When you have the image coming from the true distribution, the function takes the form:

$$Loss(D(x), 1) = [1 * log(D(x)) + (1 - 1) * log(1 - D(x)]$$
$$= log(D(x)) - (1)$$

When you have the image coming from the generator, the function takes the form:

$$Loss(D(G(z)), 0) = [0 * log(D(G(z)) + (1 - 0) * log(1 - D(G(z))]$$
$$= log(1 - D(G(z)) - (2)$$

The terms (1) and (2) are added to achieve the final loss. To see why the discriminator maximises this loss (max_D part), you need to plot both terms.



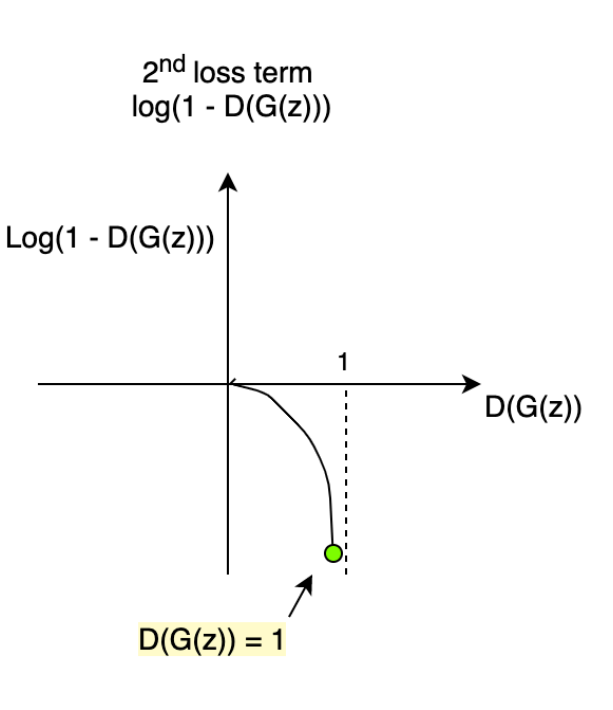
You can observe that the discriminator needs to maximize the 1^{st} term in order to correctly predict "1" for the real input image. Also, it needs to maximize the second term to correctly predict "0" for the generated input image.

Generator's Aim

The generator aims to generate such an image that the discriminator classifies it as real or 1. The generator's loss function is the same as when the discriminator receives generated image, i.e.,

$$Loss(D(G(z)), 0) = log(1 - D(G(z))$$

However, unlike the discriminator, the generator minimizes this term . Let's look at the plot again.



As you can see in the log plot of the second term, in order to fool the discriminator (D(G(z))=1), you need to minimize the term.

Training the GAN

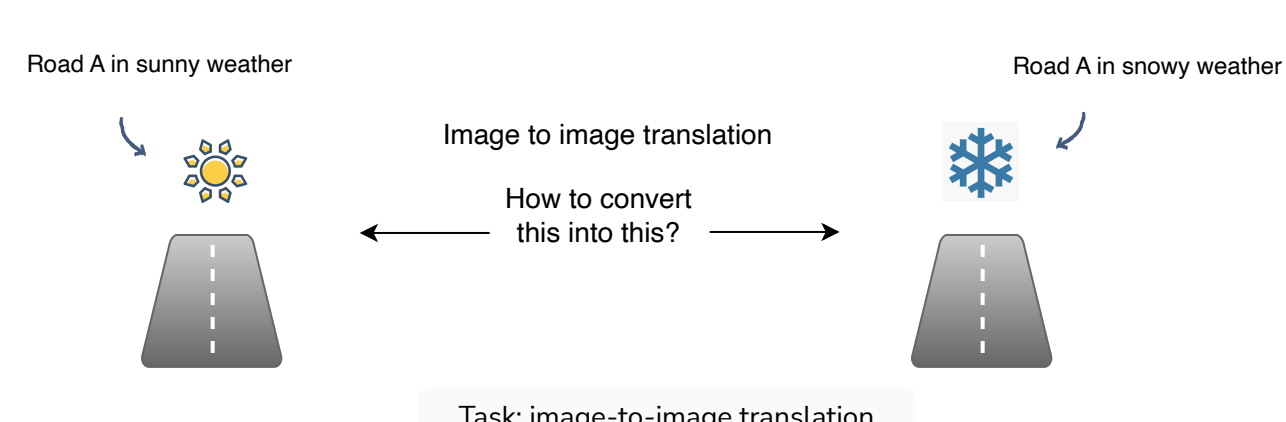
The generator and discriminator are trained alternatively.

For training the discriminator, the weights of the generator model are frozen. Two mini-batches are taken. One is from the real images data. The other consists of generated images obtained by feeding random noise samples to the generator. There are no labels associated with this data. However, the discriminator still learns due to the formulation of the loss function, which that we discussed earlier. The discriminator updates its weights to *maximize* the terms leading to fake images being classified as 0 and real as 1.

For training the generator, the weights of the discriminator model are frozen. A mini-batch is taken from the noise. Once again there are no labels, but the model updates its weights to *minimize* the loss term, leading to fake images being classified as 1.

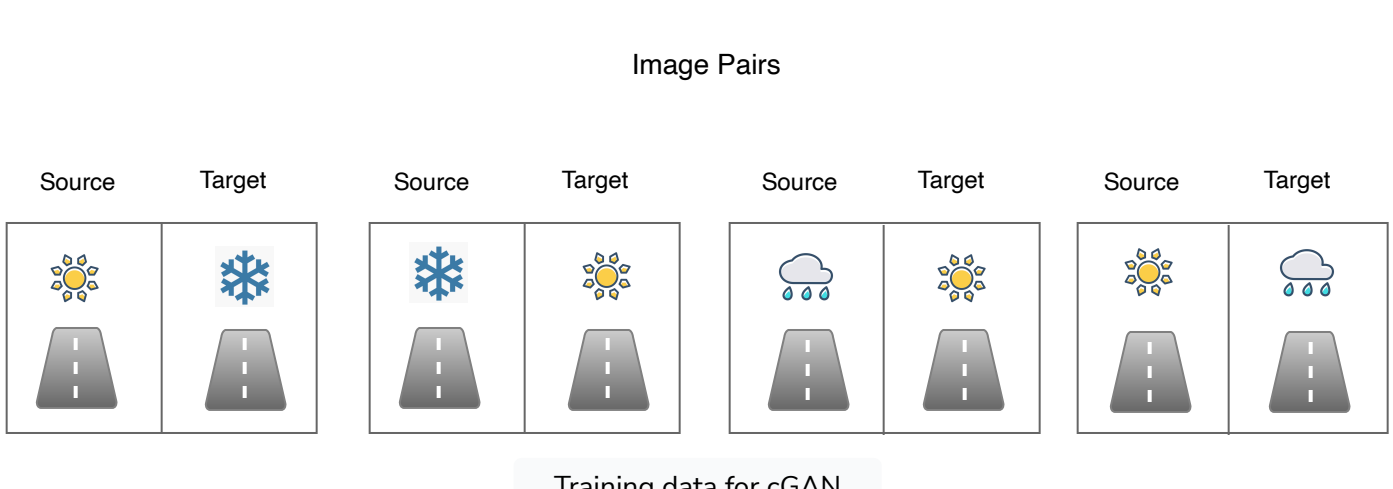
If you train a GAN on a set of driving images, you do not have any control over the generator's output, i.e., it would just generate driving images similar to the images present in the training dataset. However, that is not particularly useful in your case.

Therefore, we want to perform **image-to-image translation** (a kind of data augmentation) to enhance our training data, i.e., you want to map features from an input image to an output image. For instance, you want to learn to map driving images with one weather condition to another weather condition, or you may want to convert day time driving images to night time images and vice versa. This can be done with a cGAN which is a deep neural network that extends the concept of GAN.



Task: image-to-image translation

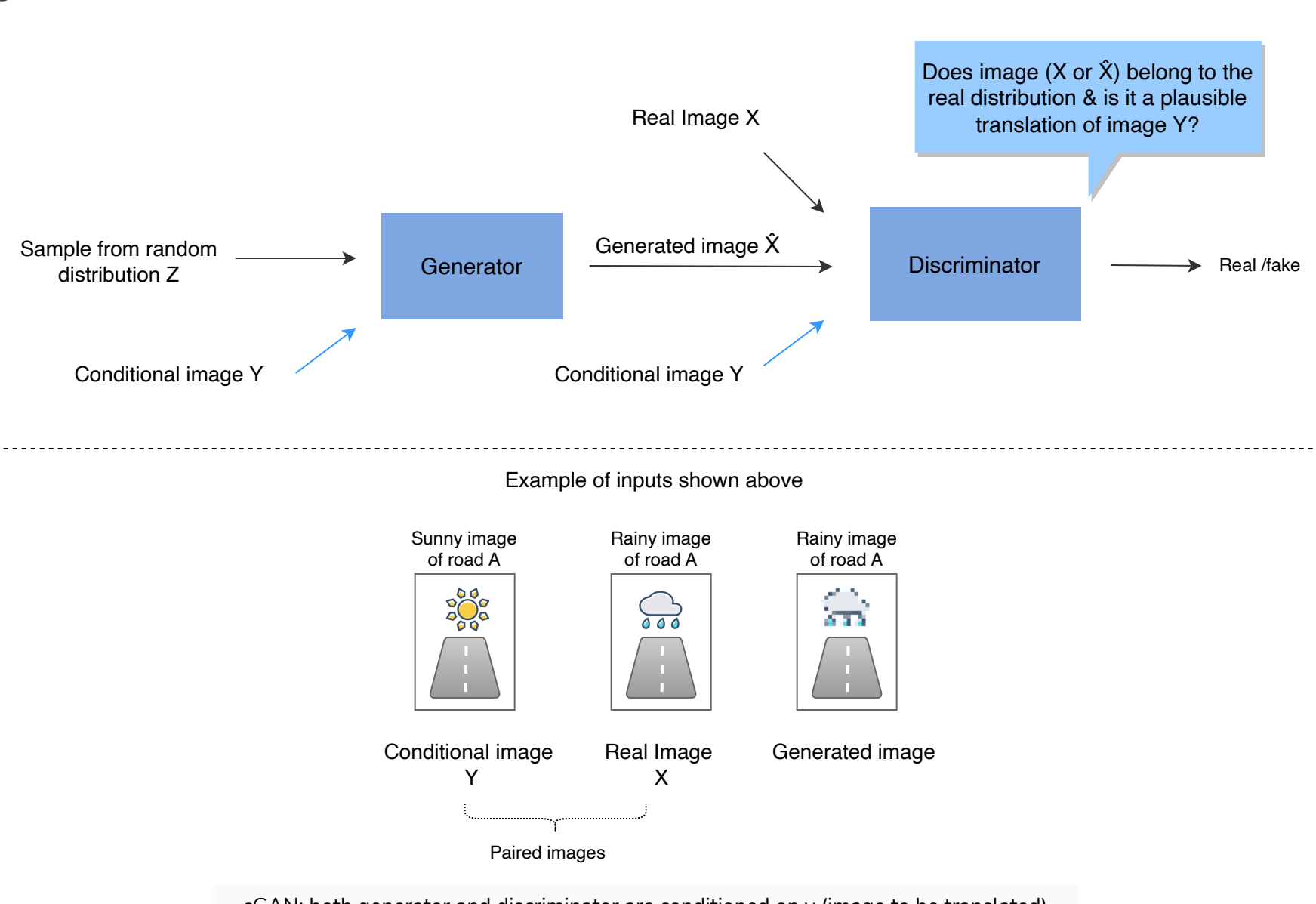
For image translation, your training data should consist of paired images. For example, image pairs could be of the same road during day and night. They could also be of the same road during winter and summer.



Training data for cGAN

In cGAN, you give the source image that you want to translate to the generator, along with an additional input (target image as a condition). This will guide the generator's output image to be of the same place but with a different weather/light condition.

You will also feed the source image to the discriminator so that apart from its usual task of discriminating between real and generated images, it will also see if the given image is a plausible translation of the source image.



cGAN: both generator and discriminator are conditioned on y (image to be translated)

The loss function for cGAN is as follows:

$$min_G \ max_D \ \{E_{x \sim p_{data}(x)}[log\ D(x|y)] + E_{z \sim p_z(z)}[log(1 - D(G(z|y)))]\}$$

To understand this loss function you can refer to the expandable section on GANs above. You see that the cGAN loss function differs from the GAN loss function as the discriminator's probability D(x) or D(G(z)) is now conditioned on y.

Once you have trained the cGAN for the image translation task, you can now feed it the driving images you manually collected through the generator model to enhance the training data.

Using a similar approach, you can build multiple cGAN models for different conditions, including generating night images, cloudy images, snow images etc., and enhance our training data set with examples of all conditions.

> 📝 This chapter does not have a feature engineering lesson, because the convolutional layers (in the CNN) extract features themselves.

Targeted data gathering

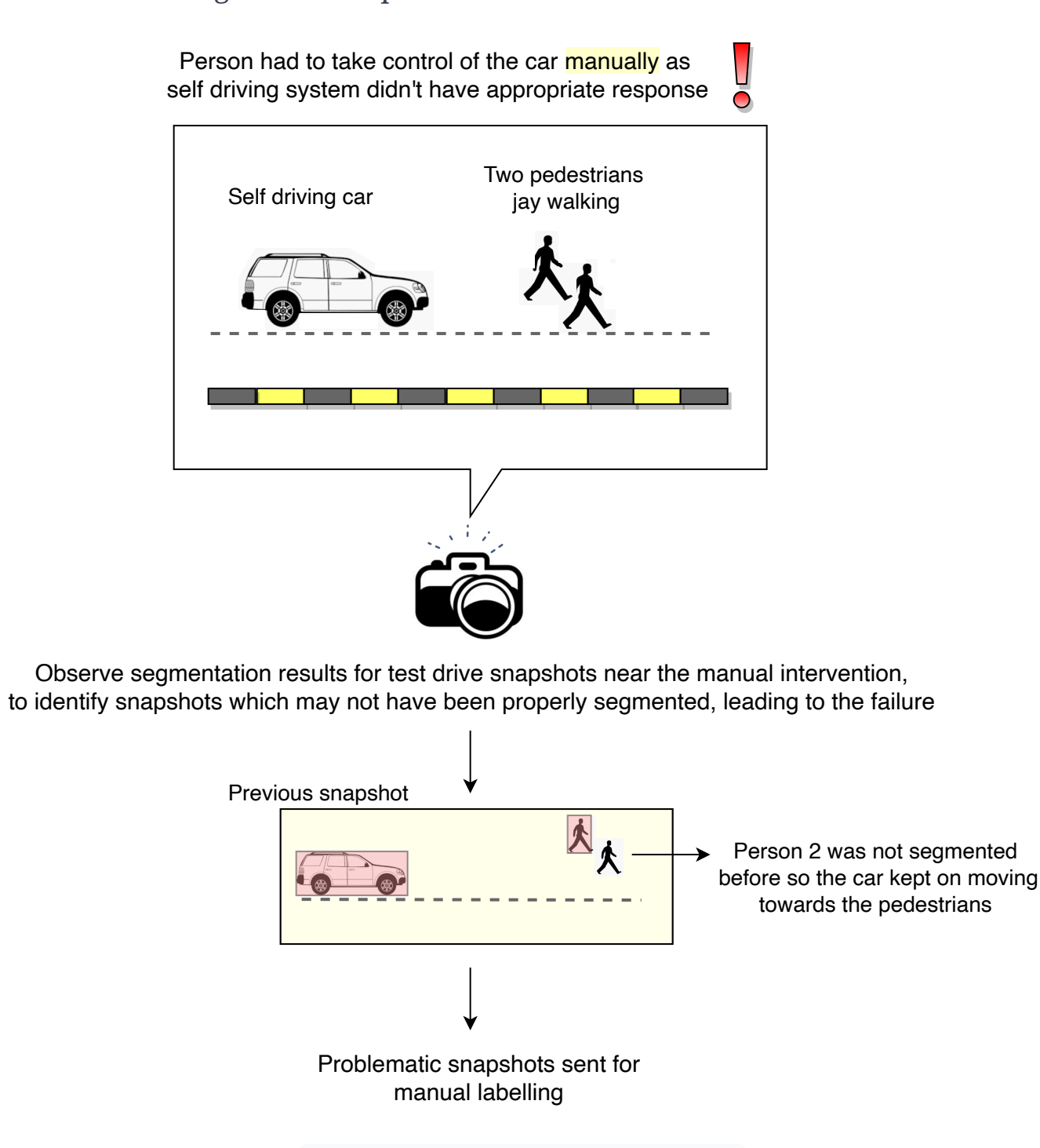
Earlier in the chapter, we discussed manual intervention as a metric for our end-to-end self-driving car system. You can use this metric to identify scenarios where your segmentation model is not performing well and learn from them.

> 📝 Here, you aim to gather training data specifically on the areas where you fail while testing the system. It will help to further enhance the training data.

You will test the self-driving car, with a person at the wheel. The vehicle has the capability to record the video of this test drive. After the test drive is over, you will observe the system and look for instances where it did not perform well and the person had to intervene.

Consider a scenario where you observe that the person had to manually intervene when two pedestrians were jaywalking. You will check the performance of the segmentation model on, let's say, the last twenty frames before the manual intervention took place. You might see that segmentation failed to predict one of the two pedestrians. You will select snapshots of those instances from the recording and ask the manual labelers to labels those snapshots since it means that the training data lacks such examples.

You will then focus on collecting such examples and ask the manual labelers to label them.



Why you need targeted data gathering

After this, you can also apply data augmentation on the newly gathered training examples to further improve your model's performance.

Modeling

In modeling discussion, we will discuss two key aspects.
1. SOTA Segmentation Models: What are the state-of-the-art segmentation models and their architectures?
2. Transfer Learning: How can you use these models to train a better segmenter for the self-driving car data?

SOTA segmentation models

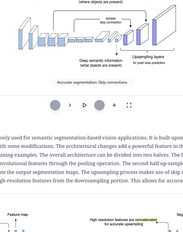
Machine learning in general and deep learning, in particular, have progressed a lot in the domain of computer vision-based applications during the last decade. The models enlisted in this section are the most commonly used deep neural networks that provide state-of-the-art (SOTA) *results for object detection and segmentation tasks*. These tasks form the basis for the self-driving car use case.

FCN

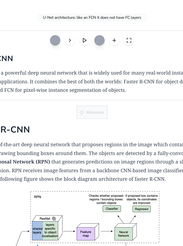
Fully convolutional networks (FCNs) are one of the top-performing networks for semantic segmentation tasks.

> Segmentation is a dense prediction task of pixel-wise classification.

A typical FCN operates by fine-tuning an image classification CNN and applying pixel-wise training. It first compresses the information using multiple layers of convolutions and pooling. Then, it up-samples these feature maps to predict each pixel's class from this compressed information.

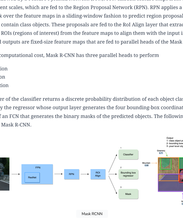


The convolutional layers at the end (instead of the fully connected layers) also allow for *dynamic input size*. The output pixel-wise classification tends to be coarse, so you make use of skip connections to achieve good edges. The initial layers capture the edge information through the deepness of the network. You use that information during our upsampling to get more refined segmentation.



U-Net

U-Net is commonly used for semantic segmentation-based vision applications. It is built upon the FCN architecture with some modifications. The architectural changes add a powerful feature in the network to require less training examples. The overall architecture can be divided into two halves. The first half down-samples the convolutional features through the pooling operation. The second half up-samples the feature maps to generate the output segmentation maps. The upsampling process makes use of skip connections to concatenate high-resolution features from the downsampling portion. This allows for accurate upsampling.



Mask R-CNN

Mask R-CNN is a powerful deep neural network that is widely used for many real-world instance segmentation applications. It combines the best of both the worlds: Faster R-CNN for object detection and localization and FCN for pixel-wise instance segmentation of objects.

Advanced

Faster R-CNN

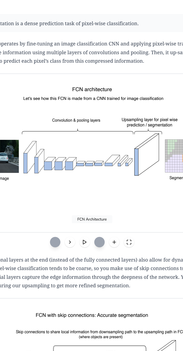
It is a state-of-the-art deep neural network that proposes regions in the image which contain different objects by drawing bounding boxes around them. The objects are detected by a fully-convolutional **Region Proposal Network (RPN)** that generates predictions on image regions through a sliding window fashion. RPN receives image features from a backbone CNN-based image classifier, e.g., ResNet. The following figure shows the block diagram architecture of faster R-CNN.

Mask R-CNN architecture is constructed by using a convolutional backbone of a powerful CNN classifier (e.g. ResNet) followed by a Feature Pyramid Network (FPN). FPN extracts feature maps from the input image at different scales, which are fed to the Region Proposal Network (RPN). RPN applies a convolutional neural network over the feature maps in a sliding-window fashion to predict region proposals as bounding boxes that will contain class objects. These proposals are fed to the RoI Align layer that extracts the corresponding ROIs (regions of interest) from the feature maps to align them with the input image properly. The ROI pooled outputs are fixed-size feature maps that are fed to parallel heads of the Mask R-CNN.

To reduce the computational cost, Mask R-CNN has three parallel heads to perform

1. Classification
2. Localization
3. Segmentation

The output layer of the classifier returns a discrete probability distribution of each object class. Localization is performed by the regressor whose output layer generates the four bounding-box coordinates. The third arm consists of an FCN that generates the binary masks of the predicted objects. The following is a block diagram of the Mask R-CNN.



> It is worth mentioning that *most* of the deep vision architectures are not inherently designed to preserve sensitivity for different scales, sizes and rotations of the objects/images. The pooling operation in these networks is only able to capture the translational variance of the labeled objects/images. Hence, it is critical to cover the variations in scale and rotation of your training images and keep the testing image size similar to the image size in your training dataset.

Transfer learning

Let's answer the question of how to utilize the discussed SOTA segmentation models to build one for the self-driving car.

Transfer learning is a widely used technique in the field of deep learning: utilizing pre-trained powerful deep neural networks (DNNs) for weak or small datasets. When you kickstart a deep learning project with a small ground-truth (human-annotated) training data, it is often challenging to scale the performance of your model on a standard-sized large training dataset. Since increasing the size of human-annotated data can be time-consuming, transfer learning helps take advantage of a powerful deep neural network that is pre-trained on a different but large human-annotated dataset that is similar in nature to the dataset in hand.

> We use transfer learning to build upon learned knowledge from one dataset to improve learning in the dataset.

> The trick of transfer learning can be explained through the functionality of the human vision system, which is the basis of convolutional and pooling layers in deep networks. There is a striking similarity between the low and high-level feature detectors (edges and complex neurons) of the primate vision system and convolutional and pooling operations in deep networks.

In a trained model, the low-level feature detectors are fine-tuned to filter high-frequency visual information (edges), which are mostly common in natural images. For example, you can transfer the learned edge detectors of a model trained to detect cats to detect dogs. For more similar objects/images, this similarity can also be correlated in high-level feature detectors. In this regard, freezing the layers of a trained deep neural network in transfer learning helps take advantage of the learned feature detectors from a large and rather similar natural imaging dataset.

Assume that, from the above-mentioned SOTA models, you want to utilize a pre-trained FCN (on ImageNet dataset) for performing segmentation for your self-driving vehicle. Let's see three different approaches where you can apply transfer learning.

Retraining topmost layer

You can take advantage of the large features' basis in the ImageNet FCN and re-train it for your smaller driving image dataset. In this case, you need to update the final pixel-wise prediction layer in the pre-trained FCN (i.e., replace the classes in ImageNet trained model with classes in driving image set) and retrain the final layer through an optimizer while freezing the remaining layers in the FCN.

This approach makes the most sense when the driving images data is limited and/or you believe that the current learned layers capture the information that you need for making a prediction.

Retraining top few layers

Similarly, in cases where you have a medium-sized driving images dataset available, you can also try retraining some deeper upsampling layers to improve the accuracy of your segmenter.

You can *experiment with how many upsampling layers you need to retrain* by looking at the *IoU metric* to see if retraining any further improves the performance of the segmentation model.

This approach makes the most sense when you have a decent amount of driving images data. Also, shallow layers generally don't need training because they are capturing the basic image features, e.g., edges, which don't need retraining.

Retraining entire model

As the size of your dataset increase, you can consider retraining even more layers or retraining the entire model. Once again, you can use the IoU metric to evaluate the optimal number of layers whose retraining can increase model performance.

Generally, retraining the entire network is laborious and time-consuming. It is usually done only if the dataset under consideration has completely different characteristics from the one on which the network was pre-trained.

On top of choosing or designing the optimal DNN architecture, one can also be creative in adding some extra features in the network to handle the data bias and variance issues. For instance, adding a regularizer (e.g., L1/L2) after a DNN architecture, making a hyper-parameter sweep for fine-tuning and experimenting with different optimizers (e.g., Adam and SGD) during training can further enhance the performance of the model.

Problem Statement

Let's get acquainted with the task of building an entity linking system.

We'll cover the following

Introduction

Named entity linking (NEL) is the process of detecting and linking entity mentions in a given text to corresponding entities in a target knowledge base.

There are two parts to entity linking:

1. *Named-entity recognition*

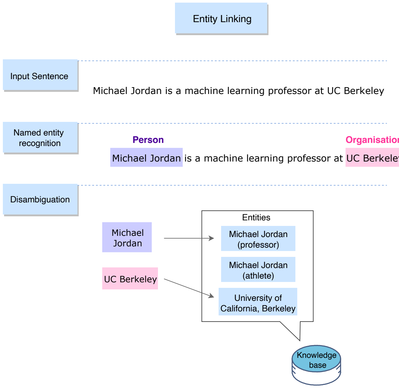
   Named-entity recognition (NER) detects and classifies potential named entities in the text into predefined categories such as a person, organization, location, medical code, time expression, etc. (multi-class prediction).

2. *Disambiguation*

   Next, disambiguation disambiguates each detected entity by linking it to its corresponding entity in the knowledge base.

> The target knowledge base depends on the application, but for generic systems, a common choice is Wikidata or DBpedia.

Let's see entity linking in action in the following example.



Entity linking

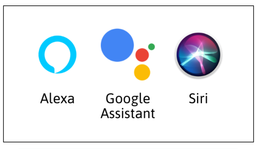
The sentence/text says, "Michael Jordan is a machine learning professor at UC Berkeley." First NER detects and classifies the named entities Michael Jordan and UC Berkeley as person and organisation.

Then disambiguation takes place. Assume that there are two 'Michael Jordan' entities in the given knowledge base, the UC Berkeley professor and the athlete. Michael Jordan in the text is linked to the *professor at the University of California, Berkeley* entity in the knowledge base (that the text is referring to). Similarly, UC Berkeley in the text is linked to the *University of California* entity in the knowledge base.

Applications

Entity linking has applications in many natural language processing tasks. The use cases can be broadly categorized as information retrieval, information extraction and building knowledge graphs, which in turn can be used in many systems, such as:

- Semantic search
- Content analysis
- Question answering systems/chatbots/virtual assistants, like Alexa, Siri, and Google assistant



NEL applications. Image source: www.alexeko.com

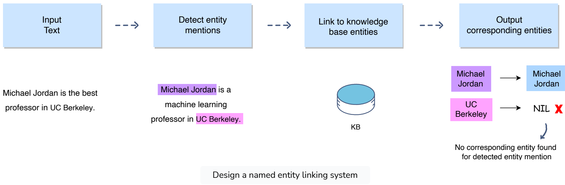
All of the above-mentioned applications require a high-level representation of the text, in which concepts relevant to the application are separated from the text and other non-meaningful data.

Now that we have introduced named entity linking and its applications, we are ready to look at the problem statement.

Problem statement

The interviewer has asked you to design an entity linking system that:

- Identifies potential named entity mentions in the text.
- Searches for possible corresponding entities in the target knowledge base for disambiguation.
- Returns either the best candidate corresponding entity or nil.



Design a named entity linking system

The problem statement translates to the following machine learning problem:

"Given a text and knowledge base, find all the entity mentions in the text(Recognize) and then link them to the corresponding correct entry in the knowledge base(Disambiguate)."

Interview questions

These are some of the questions that an interviewer can put forth.

1. How would you build an entity recognizer system?
2. How would you build a disambiguation system?
3. Given a piece of text, how would you extract all persons, countries, and businesses mentioned in it?
4. How would you measure the performance of a disambiguator/entity recognizer/entity linker?
5. Given multiple disambiguators/recognizers/linkers, how would you figure out which is the best one?

Back
Modeling

Next
Metrics