Performance and Capacity Considerations

This lesson provides a quick introduction to performance and capacity considerations and discusses why they matter when designing a solution to a machine learning problem.

We'll cover the following

As we work on a machine learning-based system, our goal is generally to improve our metrics (engagement rate, etc.) while ensuring that we meet the capacity and performance requirements.

Major performance and capacity discussions come in during the following two phases of building a machine learning system:

1. *Training time*: How much training data and capacity is needed to build our predictor?
2. *Evaluation time*: What are the Service level agreement(SLA) (https://en.wikipedia.org/wiki/Service-level_agreement) that we have to meet while serving the model and capacity needs?

We need to consider the performance and capacity along with optimization for the ML task at hand, i.e., measure the complexity of the ML system at the training and evaluation time and use it in the decision process of building our ML system architecture as well as in the selection of the ML modeling technique.

Complexities consideration for an ML system

Machine learning algorithms have three different types of complexities:

- **Training complexity**

  The training complexity of a machine learning algorithm is the time taken by it to train the model for a given task.

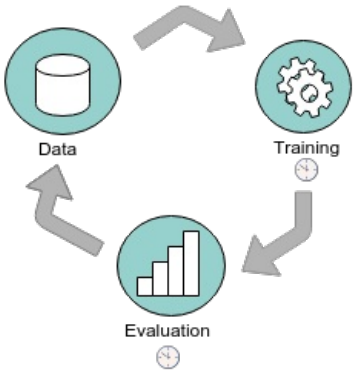
- **Evaluation complexity**

  The evaluation complexity of a machine learning algorithm is the time taken by it to evaluate the input at testing time.

- **Sample complexity**

  The sample complexity of a machine learning algorithm is the total number of training samples required to learn a target function successfully.

> Sample complexity changes if the model capacity changes. For example, for a deep neural network, the number of training examples has to be considerably larger than decision trees and linear regression.



The measure of training and evaluation time complexity on a sample data in a ML system

Comparison of training and evaluation complexities

You can see how the training and evaluation complexities can be used to evaluate which model will be best for a given task and resources.

Assume that

- n is the number of the training samples
- f is the number of features
- n_{trees} is the number of trees (for tree-based algorithms)
- n_{l_i} is the number of neurons at i^{th} layer in a neural network
- e is the number of epochs
- d is the max depth of tree

The training and prediction complexity can be approximated in terms of asymptotic analysis as follows:

Algorithm	**Training time**	**Evaluation time**
Linear/Logistic Regression (Batch)	$O(nfe)$	$O(f)$
Neural Network	Exponential (varies per implementation)	$O(fn_{l_1} + n_{l_1}n_{l_2} + \ldots)$
Multiple Additive Regression Trees (MART)	$O(ndfn_{\text{trees}})$	$O(fdn_{\text{trees}})$

Analysis

- The evaluation complexity of the *linear regression* algorithm is equal to the complexity of a single-layer neural network-based algorithm. Linear regression is the best choice if we want to save time on training and evaluation. Let's assume the model evaluates one example in 5 μs. For 100k examples, it would take 100k x 5 μs = 500 ms execution time on a single machine.

  For example, for the ad prediction system, the service level agreement(SLA) says that we need to select the relevant ads from the pool of ads in 300 ms. Given this request, we need a fast algorithm. Here linear regression would serve the purpose.

- Relatively *deep neural network* takes a lot more time in both training and evaluation. Its need for training data is also high. However, it's ability to learn complex tasks such as image segmentation and language understanding, is much higher, and it gives more accurate predictions in comparison to other models. Therefore a deep neural network is a viable choice if it is well suited for the task at hand and capacity isn't a problem.

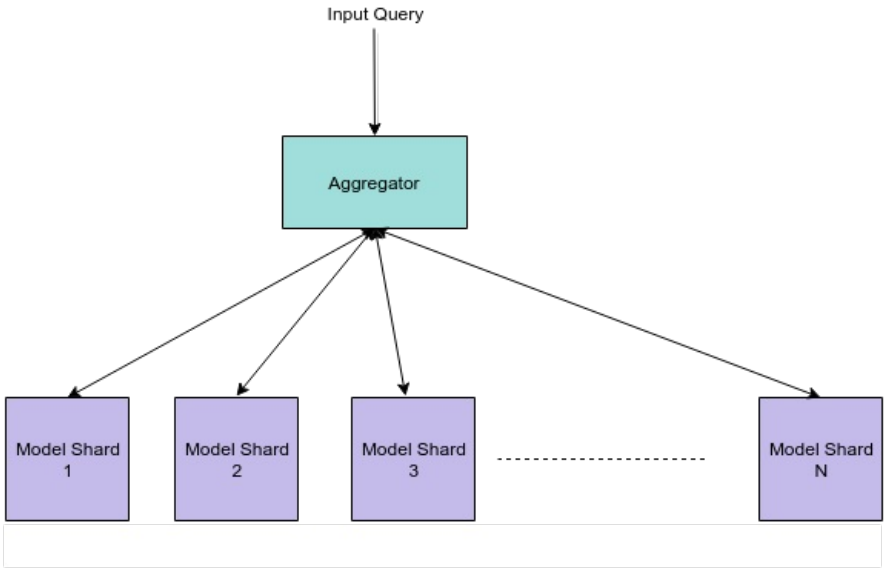
- *MART* is a tree-based algorithm that has a greater computation cost than linear models, but it is much faster than a deep neural network. Tree-based algorithms are able to generalize well using a moderately-sized training dataset. Therefore, if our training data is limited to a few million examples and capacity/performance is critical, they will be a good choice.

Performance and capacity considerations in large scale system

Consider that a search system(e.g., Google, Bing) gets a query "computer science" that matches 100 million web pages. The ML-based system wants to respond with the most relevant web pages for the searcher while meeting the system's constraints. These constraints are generally referred to as Service level agreements (SLA). There can be many SLAs around availability and fault tolerance but for our discussion of designing ML systems, *performance* and *capacity* are the most important to think about when designing the system. **Performance** based SLA ensures that we return the results back within a given time frame (e.g. 500ms) for 99% of queries. **Capacity** refers to the load that our system can handle, e.g., the system can support 1000 QPS (queries per second).

If we evaluate every document using a relatively fast model such as tree-based or linear regression and it takes 1μs, our simple model would still take 100s to run for 100 million documents that matched the query "computer science".

This is where distributed systems come in handy; we will distribute the load of a single query among multiple shards, e.g., we can divide the load among 1000 machines and can still execute our fast model on 100 million documents in 100ms (100s/1000).



Input query sent to multiple shards to distribute load

Let's now consider the scenarios in which we decided that a deep learning model for search ranking is a much better choice and helps improve our search metrics. However, deep learning is significantly slow, assuming that it needs 1ms to evaluate an example. Even now with our 1000 shards, it would still take 100s to rank all the results using this model. Clearly, we are far from our performance SLAs.

If we had unlimited capacity, one way to solve this problem would be to continue to add more shards and bring the number down. However, we cannot have unlimited capacity, and it's important to have the system find the most optimal result given a fixed capacity.

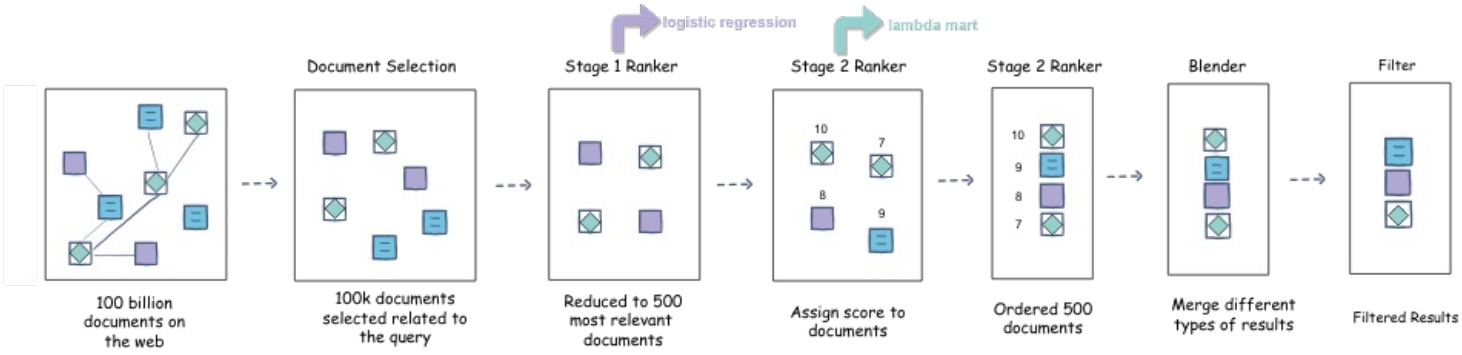
This prompts another important discussion on using a funnel- based approach when designing ML systems for performance and limited capacity. Let's discuss it next.

Layered / funnel based modeling approach

To manage both the performance and capacity of a system, one reasonable approach that's commonly used is to start with a relatively fast model when you have the most number of documents e.g. 100 million documents in case of the query "computer science" for search. In every later stage, we continue to increase the complexity (i.e. more optimized model in prediction) and execution time but now the model needs to run on a reduce number of documents e.g. our first stage could use a linear model and final stage can use a deep neural network. If we apply deep neural network for only top 500 documents, with 1ms evaluation time per document, we would need 500ms on a single machine. With five shards we can do it in around 100ms.

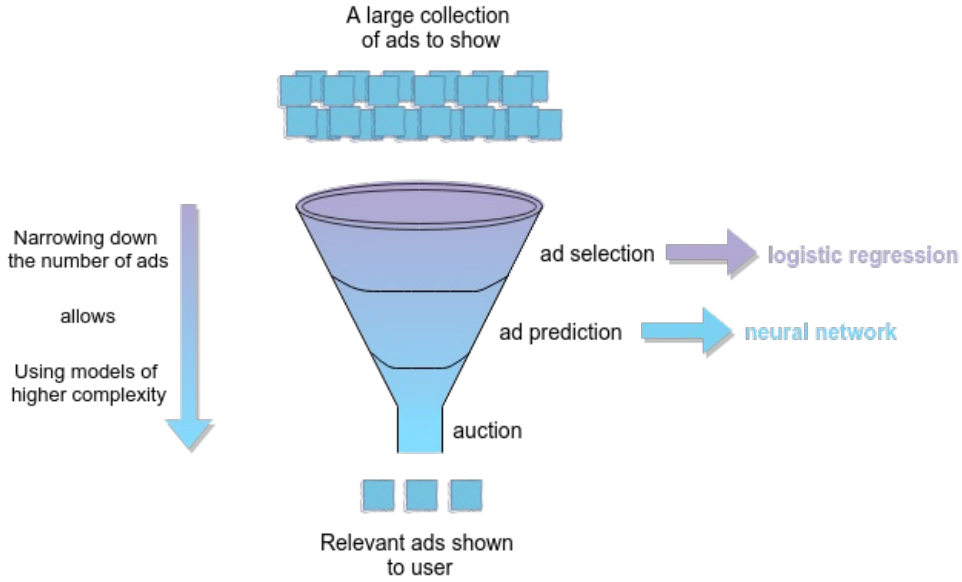
> In ML systems like search ranking, recommendation, and ad prediction, the layered/funnel approach to modeling is the right way to solve for scale and relevance while keeping performance high and capacity in check.

The following figure shows an illustration of this multi-layer funnel approach that we will explain in detail in the search ranking system design problem (https://www.educative.io/collection/page/10370001/6237869033127936/5429316609376256).



Layered model approach for search ranking system

Similarly, the following is an illustration of how an ads relevance system would look in this funnel based approach. We will discuss this in more detail in the ad prediction system chapter (https://www.educative.io/collection/page/10370001/6237869033127936/5310143676809216).



Funnel approach for an ad prediction system

Back

Setting up a Machine Learning System

Next

Training Data Collection Strategies

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/performance-and-capacity-considerations__practical-ml-techniquesconcepts__grokking-the-machine-learning-interview)

Performance and Capacity Considerations

This lesson provides a quick introduction to performance and capacity considerations and discusses why they matter when designing a solution to a machine learning problem.

We'll cover the following

As we work on a machine learning-based system, our goal is generally to improve our metrics (engagement rate, etc.) while ensuring that we meet the capacity and performance requirements.

Major performance and capacity discussions come in during the following two phases of building a machine learning system:

1. *Training time*: How much training data and capacity is needed to build our predictor?
2. *Evaluation time*: What are the Service level agreement(SLA) (https://en.wikipedia.org/wiki/Service-level_agreement) that we have to meet while serving the model and capacity needs?

We need to consider the performance and capacity along with optimization for the ML task at hand, i.e., measure the complexity of the ML system at the training and evaluation time and use it in the decision process of building our ML system architecture as well as in the selection of the ML modeling technique.

Complexities consideration for an ML system

Machine learning algorithms have three different types of complexities:

- **Training complexity**

  The training complexity of a machine learning algorithm is the time taken by it to train the model for a given task.

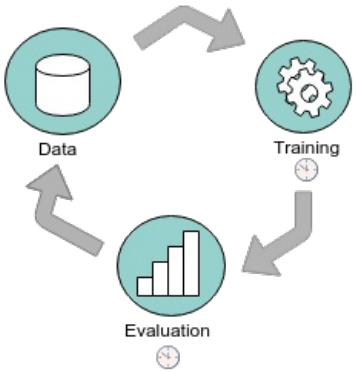
- **Evaluation complexity**

  The evaluation complexity of a machine learning algorithm is the time taken by it to evaluate the input at testing time.

- **Sample complexity**

  The sample complexity of a machine learning algorithm is the total number of training samples required to learn a target function successfully.

> Sample complexity changes if the model capacity changes. For example, for a deep neural network, the number of training examples has to be considerably larger than decision trees and linear regression.



The measure of training and evaluation time complexity on a sample data in a ML system

Comparison of training and evaluation complexities

You can see how the training and evaluation complexities can be used to evaluate which model will be best for a given task and resources.

Assume that

- n is the number of the training samples
- f is the number of features
- n_{trees} is the number of trees (for tree-based algorithms)
- n_{l_i} is the number of neurons at i^{th} layer in a neural network
- e is the number of epochs
- d is the max depth of tree

The training and prediction complexity can be approximated in terms of asymptotic analysis as follows:

Algorithm	**Training time**	**Evaluation time**
Linear/Logistic Regression (Batch)	$O(nfe)$	$O(f)$
Neural Network	Exponential (varies per implementation)	$O(fn_{l_1} + n_{l_1}n_{l_2} + \ldots)$
Multiple Additive Regression Trees (MART)	$O(ndfn_{\text{trees}})$	$O(fdn_{\text{trees}})$

Analysis

- The evaluation complexity of the *linear regression* algorithm is equal to the complexity of a single-layer neural network-based algorithm. Linear regression is the best choice if we want to save time on training and evaluation. Let's assume the model evaluates one example in 5 μs. For 100k examples, it would take 100k x 5 μs = 500 ms execution time on a single machine.

  For example, for the ad prediction system, the service level agreement(SLA) says that we need to select the relevant ads from the pool of ads in 300 ms. Given this request, we need a fast algorithm. Here linear regression would serve the purpose.

- Relatively *deep neural network* takes a lot more time in both training and evaluation. Its need for training data is also high. However, it's ability to learn complex tasks such as image segmentation and language understanding, is much higher, and it gives more accurate predictions in comparison to other models. Therefore a deep neural network is a viable choice if it is well suited for the task at hand and capacity isn't a problem.

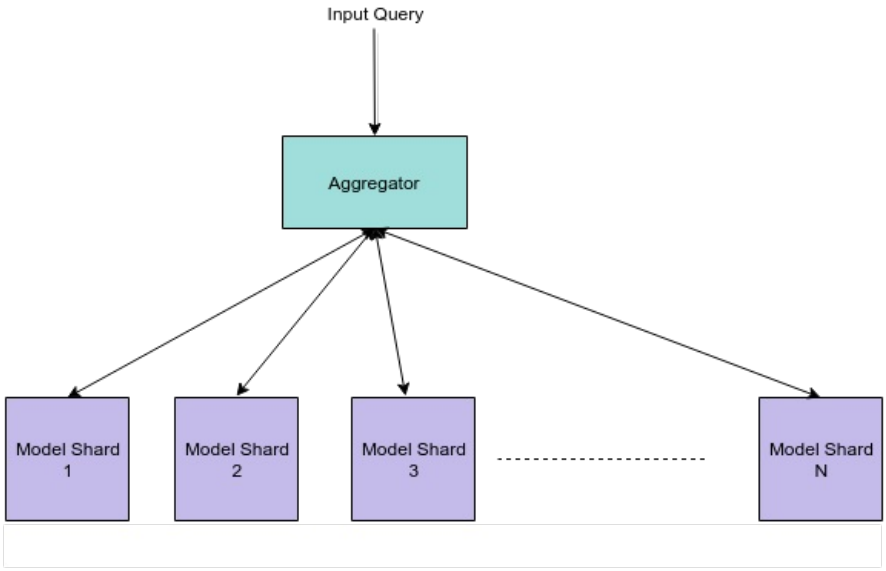
- *MART* is a tree-based algorithm that has a greater computation cost than linear models, but it is much faster than a deep neural network. Tree-based algorithms are able to generalize well using a moderately-sized training dataset. Therefore, if our training data is limited to a few million examples and capacity/performance is critical, they will be a good choice.

Performance and capacity considerations in large scale system

Consider that a search system(e.g., Google, Bing) gets a query "computer science" that matches 100 million web pages. The ML-based system wants to respond with the most relevant web pages for the searcher while meeting the system's constraints. These constraints are generally referred to as Service level agreements (SLA). There can be many SLAs around availability and fault tolerance but for our discussion of designing ML systems, *performance* and *capacity* are the most important to think about when designing the system. **Performance** based SLA ensures that we return the results back within a given time frame (e.g. 500ms) for 99% of queries. **Capacity** refers to the load that our system can handle, e.g., the system can support 1000 QPS (queries per second).

If we evaluate every document using a relatively fast model such as tree-based or linear regression and it takes 1μs, our simple model would still take 100s to run for 100 million documents that matched the query "computer science".

This is where distributed systems come in handy; we will distribute the load of a single query among multiple shards, e.g., we can divide the load among 1000 machines and can still execute our fast model on 100 million documents in 100ms (100s/1000).



Input query sent to multiple shards to distribute load

Let's now consider the scenarios in which we decided that a deep learning model for search ranking is a much better choice and helps improve our search metrics. However, deep learning is significantly slow, assuming that it needs 1ms to evaluate an example. Even now with our 1000 shards, it would still take 100s to rank all the results using this model. Clearly, we are far from our performance SLAs.

If we had unlimited capacity, one way to solve this problem would be to continue to add more shards and bring the number down. However, we cannot have unlimited capacity, and it's important to have the system find the most optimal result given a fixed capacity.

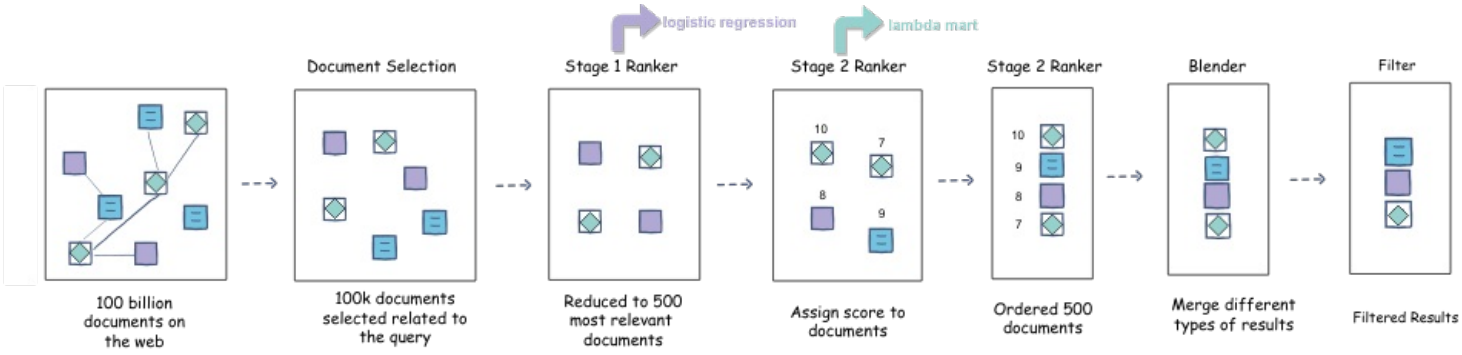
This prompts another important discussion on using a funnel- based approach when designing ML systems for performance and limited capacity. Let's discuss it next.

Layered / funnel based modeling approach

To manage both the performance and capacity of a system, one reasonable approach that's commonly used is to start with a relatively fast model when you have the most number of documents e.g. 100 million documents in case of the query "computer science" for search. In every later stage, we continue to increase the complexity (i.e. more optimized model in prediction) and execution time but now the model needs to run on a reduce number of documents e.g. our first stage could use a linear model and final stage can use a deep neural network. If we apply deep neural network for only top 500 documents, with 1ms evaluation time per document, we would need 500ms on a single machine. With five shards we can do it in around 100ms.

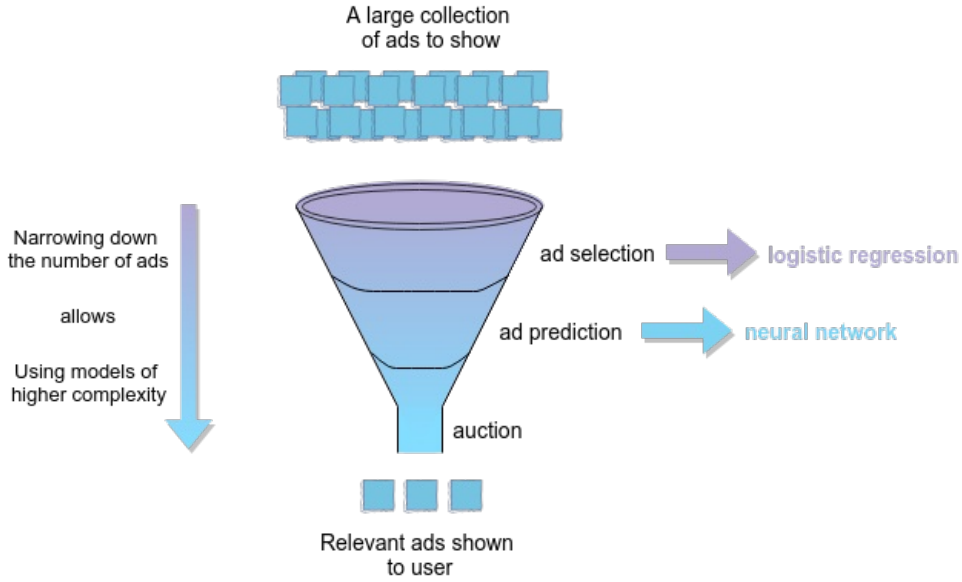
> In ML systems like search ranking, recommendation, and ad prediction, the layered/funnel approach to modeling is the right way to solve for scale and relevance while keeping performance high and capacity in check.

The following figure shows an illustration of this multi-layer funnel approach that we will explain in detail in the search ranking system design problem (https://www.educative.io/collection/page/10370001/6237869033127936/5429316609376256).



Layered model approach for search ranking system

Similarly, the following is an illustration of how an ads relevance system would look in this funnel based approach. We will discuss this in more detail in the ad prediction system chapter (https://www.educative.io/collection/page/10370001/6237869033127936/5310143676809216).



Funnel approach for an ad prediction system

Back

Setting up a Machine Learning System

Next

Training Data Collection Strategies

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/performance-and-capacity-considerations__practical-ml-techniquesconcepts__grokking-the-machine-learning-interview)

Training Data Collection Strategies

Learn the training data collection strategies for the machine learning systems you are going to build.

We'll cover the following

Significance of training data

A machine-learning system consists of three main components. They are the training algorithm (*e.g., neural network, decision trees, etc.*), training data and features. The training data is of paramount importance. The model learns directly from the data to create and refine its rules on a given task. Therefore, inadequate, inaccurate, or irrelevant data will render even the most performant algorithms useless. The quality and quantity of training data is a big factor in determining how far you can go in our machine learning optimization task.

> *A lot of the progress in machine learning - and this is an unpopular opinion in academia - is driven by an increase in both computing power and* ***data****. An analogy is to building a space rocket: You need a huge rocket engine, and you need a lot of fuel.*
>
> — Andrew Ng

Most real-world problems fall under the category of supervised learning problems which require labeled training data. This means that it is necessary to strategically think about the collection of labeled data to feed into your learning system.

Let's explore strategies that will help in collecting labeled training data for our learning tasks.

Collection techniques

We will begin by looking at the training data collection techniques.

User's interaction with pre-existing system (online)

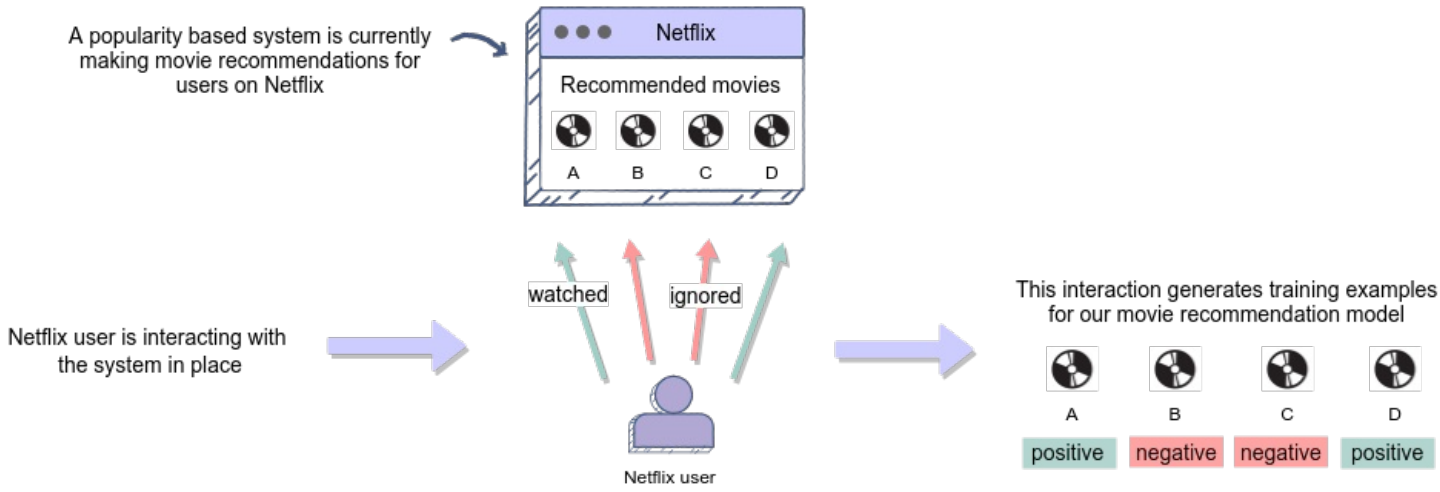
In some cases, the user's interaction with the pre-existing system can generate good quality training data.

> We will refer to this technique as online data collection in the course.

For many cases, the early version built for solving relevance (https://en.wikipedia.org/wiki/Relevance_(information_retrieval)) or ranking (https://en.wikipedia.org/wiki/Learning_to_rank) problem is a rule-based system. With the rule-based system in place, you build an ML system for the task (which is then iteratively improved). So when you build the ML system, you can utilize the user's interaction with the in-place/pre-existing system to generate training data for model training. Let's get a better idea with an example: building a movie recommender system.

Assume that you are asked to recommend movies to a Netflix user. You will be training a model that will predict which movies are more likely to be enjoyed/viewed by the user. You need to collect positive examples *(cases where user liked a particular movie)* as well as negative examples *(cases where the user didn't like a particular movie)* to feed into your model.

Here, the consumer of the system (the Netflix user) can generate training data through their *interactions with the current system* that is being used for recommending movies.



Online data collection through a user's interaction with the system in place

The early version for movie recommendation might be popularity-based, localization-based, rating-based, hand created model or ML-based. The important point here is that we can get training data from the user's interaction with this system. If a user likes/watches a movie recommendation, it will count as a positive training example, but if a user dislikes/ignores a movie recommendation, it will be seen as a negative training example.

> The above discussion was one example, but many machine learning systems utilize the current system in place for the generation of training data.
>
> We will discuss training data generation strategies from the current system in multiple problems in this course, such as search ranking, ads relevance and recommendation systems.

Human labelers (offline)

In other cases, the user of the system would not be able to generate training data. Here, you will utilize labelers to generate good quality training data.

> We will refer to this technique as offline data collection in the course.

Let's take an example of one such case: an image segmentation system for a self-driving car.

Assume that you are asked to perform image segmentation of the surroundings of a self-driving vehicle. The self-driving car will have a camera to take pictures of its surroundings. You will be training a model that will segment these pictures/images into various objects such as the road, pedestrians, building, sidewalk, signal and so on. For this, you will need to provide accurately segmented images of the self-driving car's surroundings as training data to the model.

Here, the consumer of the system (the person sitting in the self-driving car) can't generate training data for us. They are not interacting with the system in a way that would give segmentation labels for the images captured by the camera.

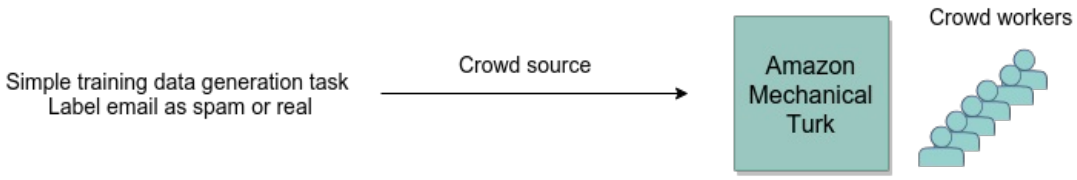
In such a scenario, we need to figure out the person/resource that can generate labeled training data for us.

Three such resources are:

1. **Crowdsourcing**

   As the name implies, in crowdsourcing, we outsource a task to a large group of people. Several crowdsourcing websites, such as Amazon Mechanical Turk (https://en.wikipedia.org/wiki/Amazon_Mechanical_Turk), where we can get quick results by hiring a lot of people for on-demand tasks, are available for this purpose.

   Crowdsourcing can be used to collect training data for relatively simpler tasks. For instance, if you are building an email spam detection system, you would need to label emails as spam or real. This is a simple task that *crowd workers* can easily do without requiring any special training.

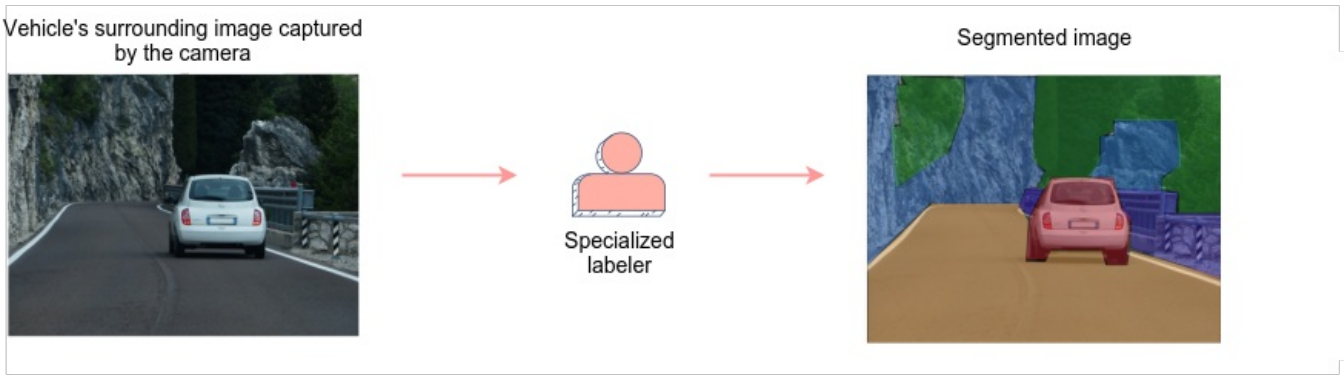


Crowdsourcing the task of generating labeled training data

However, there are cases, like when we have privacy concerns, where we cannot utilize crowdsourcing. If we do not want to show user emails to outsiders, we won't be using crowdsourcing. Another case where crowdsourcing falls short is when the task at hand requires specialized training. In these cases, we need to have specialized labelers to do the job.

2. **Specialized labelers**

   We can hire specialized/trained labelers who can label data for us according to the given ML task. Let's say, you have to build a system for the segmentation of driving images for an autonomous vehicle. The trained labelers will use software, such as Label box, to mark the boundaries of different objects in the driving images. One caveat of using specialized labelers is that training them for a specialized task may be time-consuming and costly. The tasks would be delayed until enough labelers have received training.



Specialized labellers generating training data

Targeted data gathering

Offline training data collection is expensive. So, you need to identify what kind of training data is more important and then target its collection more. To do this, you should see where the system is failing, i.e., areas where the system is unable to predict accurately. Your focus should be to collect training data for these areas.
Continuing with the autonomous vehicle example, you would see where your segmentation system is failing and collect data for those scenarios. For instance, you may find that it performs poorly for night time images and where multiple pedestrians are present. Therefore, you will focus more on gathering and labeling night time images and those with multiple pedestrians.
Have a look at the model debugging and testing lesson (https://www.educative.io/collection/page/10370001/6237869033127936/4824081099653120) to find out how to identify areas where the system is performing poorly.

3. **Open-source datasets**

   Generating training data through manual labelers is an expensive and time-consuming way to gather data. So, we need to supplement it with *open-source datasets* where possible.

For instance, “BDD100K: A Large-scale Diverse Driving Video Database” is an example of an open-source dataset that can be used as training data for this segmentation task. It contains labeled segmented data for driving images.

BDD100K: open source data set for segmentation

We may use more than one of these methods for ways to generating training data. The search ranking chapter (https://www.educative.io/collection/page/10370001/6237869033127936/5102082685140992) is a good example of this.

Additional creative collection techniques

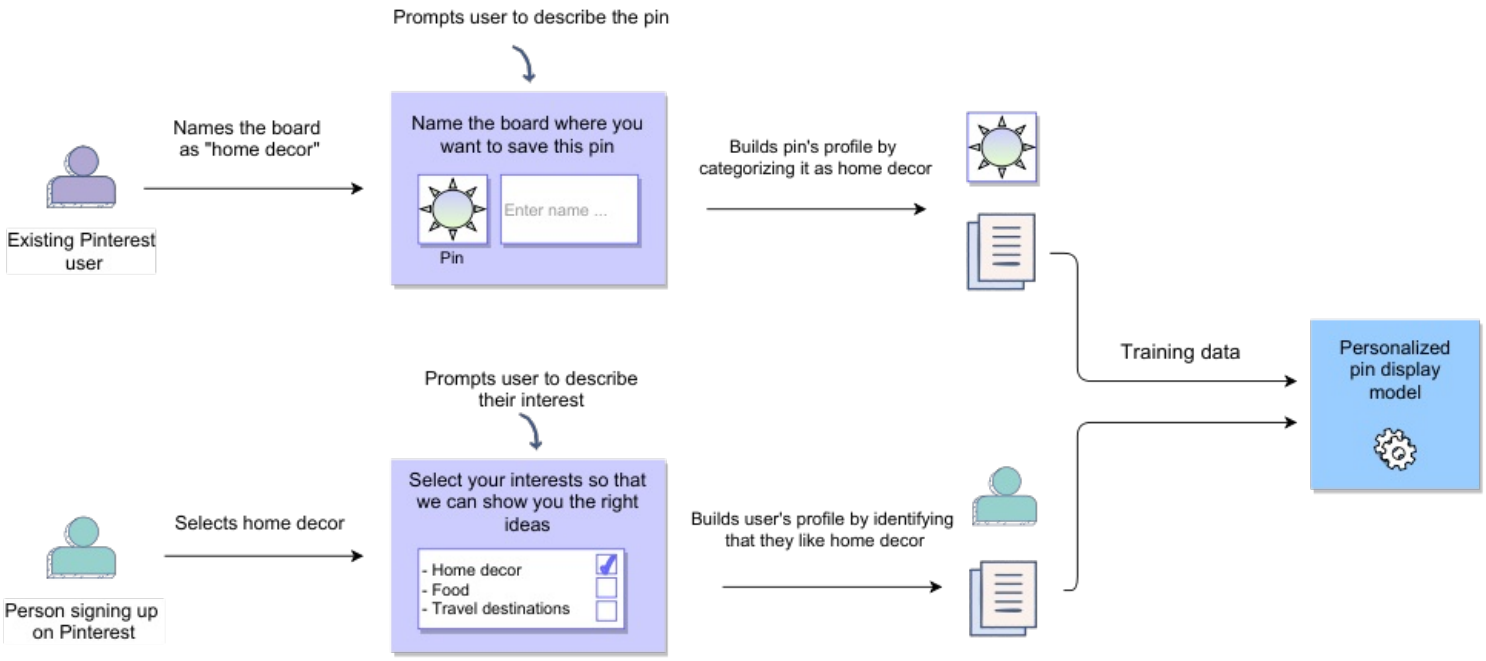
Let’s discuss a few creative ways to *collect and expand* training data.

- **Build the product in a way that it collects data from user**

  We can tweak the functionality of our product in a way that it starts generating training data for our model. Let’s consider an example where people go to explore their interests on Pinterest. You want to show a personalized selection of pins to the new users to kickstart their experience. This requires data that would give you a semantic understanding of the user and the pin. This can be done by tweaking the system in the following way:

  - Ask users to name the board (collection) to which they save each pin. The name of the board will help to categorize the pins according to their content.
  - Ask new users to choose their interests in terms of the board names specified by existing users.

  The first step will help you to build content profiles. Whereas, the second step will help you build user profiles. The model can utilize these to show pins that would interest the user, personalizing the experience.



Designing the product in a way that generates training data

- **Creative manual expansion**

  We can expand training data using creative ways. Assume that we are building a system that detects logos in images (object detection) and we have some images containing the logos we want to detect. We can expand/enhance the training data by manually placing those logos on a different set of images as well. This logo placement can be done in different positions and sizes. The enhanced training data will enable us to build a more robust logo detector model, which will be able to identify logos of all sizes at different positions and from various kinds of images.

Creative manual expansion of training data for logo detection

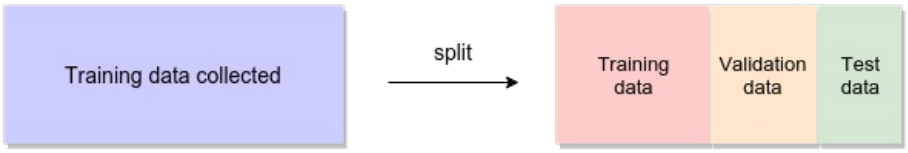
- **Data expansion using GANs**

  When working with systems that use visual data, such as object detectors or image segmenters, we can use GANs (generative adversarial networks) to enhance the training data. For instance, consider that we are building an object detector. There are a lot of training images os sunny weather conditions but less of rainy weather. A model trained on this training data may not work well for images with rainy conditions. Here, we can utilize GANs to convert images with sunny weather conditions to rainy weather conditions. This will increase our training data for rainy conditions, resulting in a more robust model.

Training data expansion through GANs: converting images in sunny weather condition to rainy weather condition

Train, test, & validation splits

We have looked at various methods to collect training data. Now it is time to split it into three parts and look at the importance of each split.



The collected training data is split into training data, validation data, and test data

The three parts are as follows:

1. **Training data**

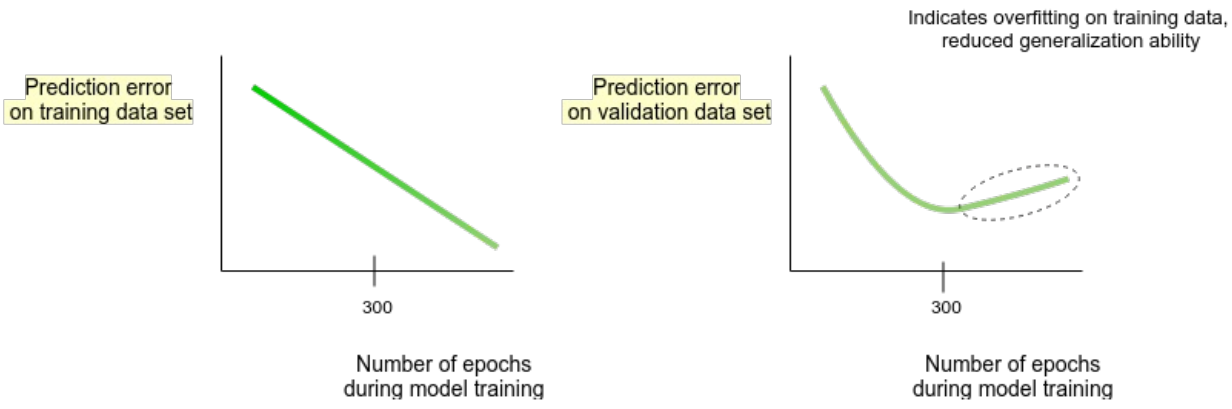
   It helps in training the ML model (fit model parameters).

   Click to expand: model parameters vs. model hyperparameters

2. **Validation data**

   After training the model, we need to tune its hyperparameters (try different model configurations). This process requires *testing the model's performance* on various hyperparameter combinations to select the best one. A simple idea could be to use the training data itself for this task. However, testing the model's performance on the same data that it was trained on would not give an accurate picture of the model's generalization ability. A good performance score may be the result of the model overfitting the training data. So we use the validation set while tuning the hyperparameters.

   Consider a scenario where you are training a neural network. You want to know the optimal number of epochs. The prediction error on the training set will continue to decrease as you keep on experimenting with a higher number of epochs. However, if you compute the error on the validation set, you would find out that maybe after 300 epochs the prediction error starts to increase, indicating that an epoch number higher than 300 will overfit the training data.



The validation set gives true picture of model's generalization ability

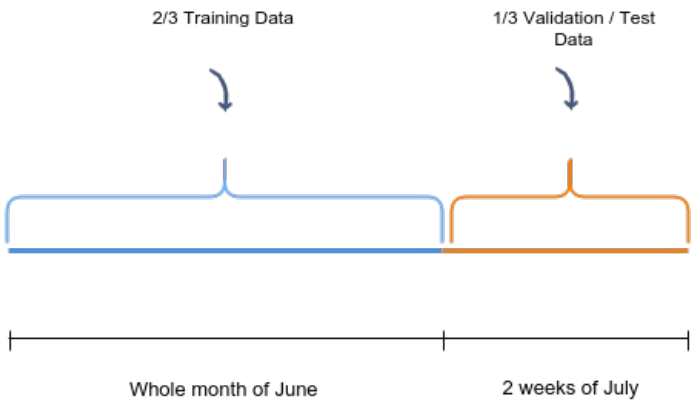
3. **Test data**

Now that we have trained and tuned the model, the final step is to test its performance on data that it has not seen before. In other words, we will be testing the model's generalization ability. The outcome of this test will allow us to make the final choice for model selection. If we had trained several models, we can decide the best ones amongst them and then further see if their performance boost is significant enough to call for an online A/B test.

We pointed out earlier that *we need a data set, which the model has not directly learned from, for this test.*. As such, training data is out of the question. Validation data is also not a suitable choice. It indirectly impacts the model as it's used to tune the model hyperparameters. This is where the test data set comes in. Test data is a completely held out data set that provides an unbiased evaluation of the final model, which has been fit on the training dataset.

Points to consider during splitting

- The size of each split will depend on your particular scenario. The training data will generally be the largest portion, especially if you are training a model like a deep neural network that requires a lot of training data. Common ratios used for training, validation and test splits are 60%, 20%, 20% or 70%, 15%, 15% respectively.
- While splitting training data, you need to ensure that you are capturing all kinds of patterns in each split. For example, if you are building a movie recommendation system like Netflix, your training data would consist of users' interactions with the movies on the platform. After analysing the data, you may conclude that, generally, the user's interaction patterns differ throughout the week. Different genres and movie lengths are preferred on different days. Hence, you will use the interaction with recommendations throughout a week to capture all patterns in each split.
- Most of the time, we are building models with the intent to forecast the future. Therefore, you need your splits to reflect this intent as well. For instance, in the movie recommendation system example, your data has a time dimension, i.e., you know the users' interactions with previous movie recommendations, and you want to predict their interactions with future recommendations ahead of time. Hence, you will train the model on data from one time interval and validate/test it on the data from its succeeding time interval, as shown in the diagram below. This will provide a more accurate picture of how our model will perform in a real scenario.

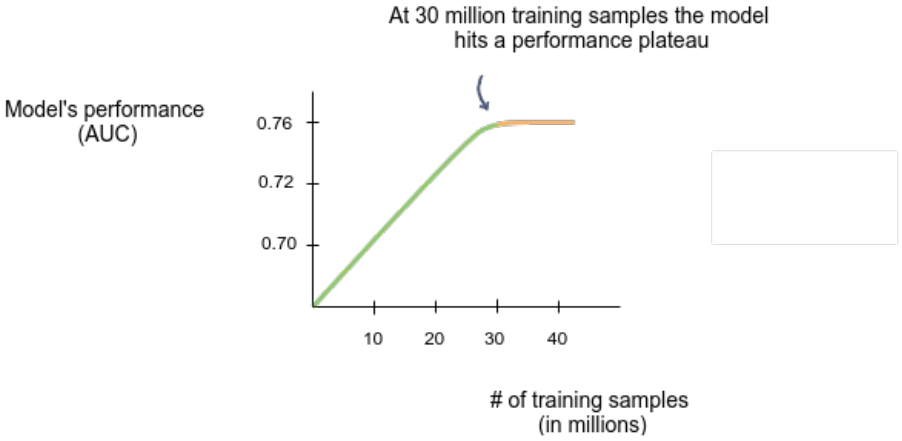


Splitting data for training, validation, and testing

Quantity of training data

As a general guideline, the quantity of the training data required depends on the modeling technique you are using. If you are training a simple linear model, like linear regression, the amount of training data required would be less in comparison to more complex models. If you are training complex models, such as a neural network, the magnitude of data required would be much greater.

Gathering a large amount of training data requires time and effort. Moreover, the model training time and cost also increase as we increase the quantity of training data. To see the optimal amount of training data, you can plot the model's performance against the number of training data samples, as shown below. After a certain quantity of training data, you can observe that there isn't any gain in the model's performance.



Plot of the model's performance on the validation set against the number of training data samples

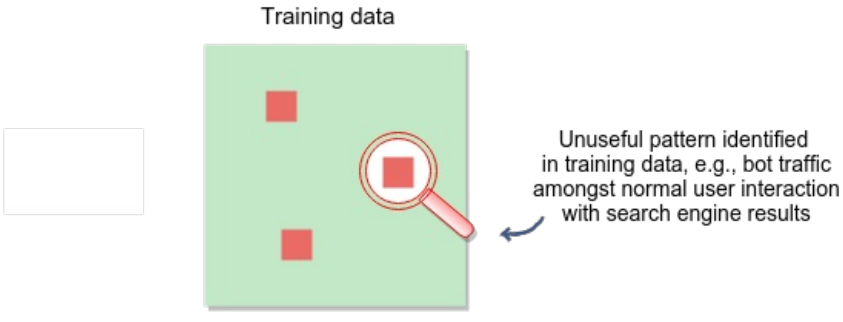
Training data filtering

It is essential to filter your training data since the model is going to learn directly from it. Any discrepancies in the data will affect the learning of the model.

- **Cleaning up data**

  General guidelines are available for data cleaning such as handling missing data, outliers, duplicates and dropping out irrelevant features.

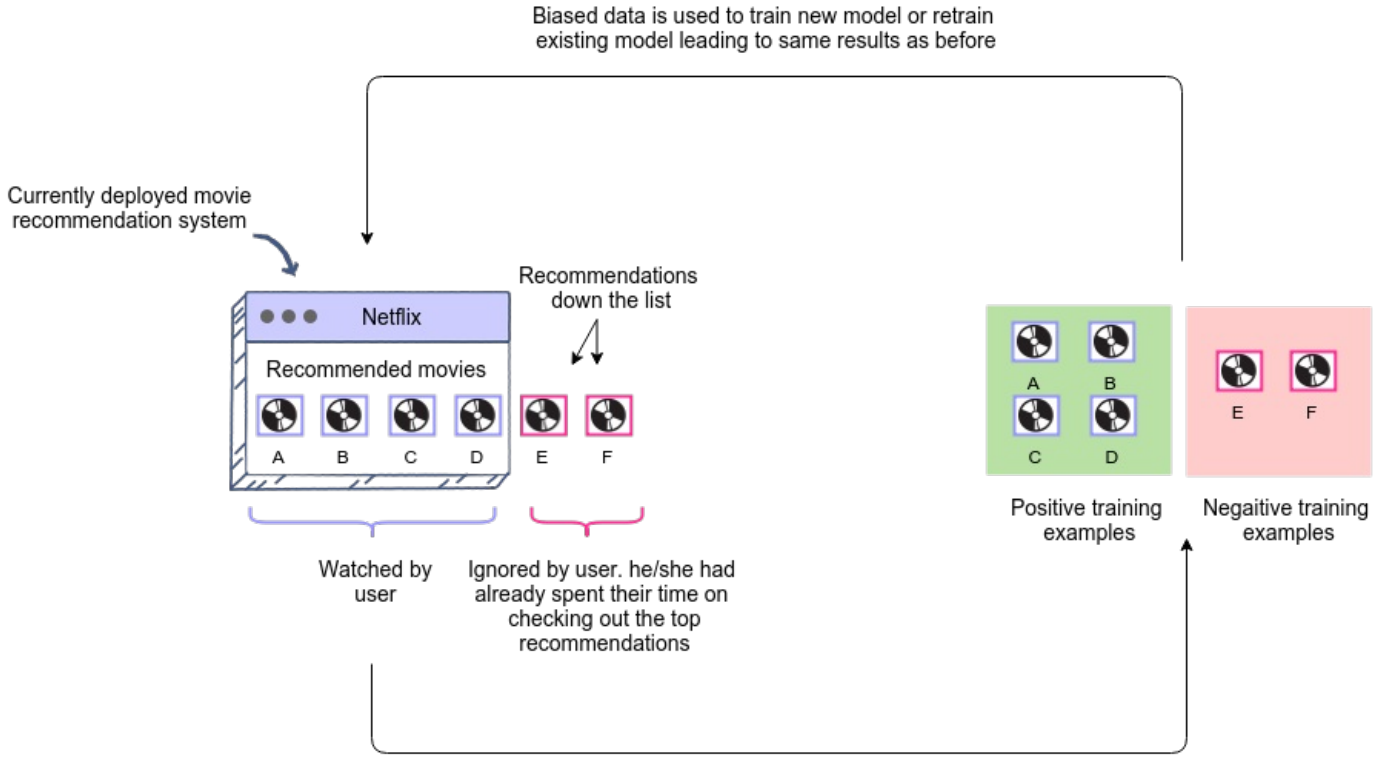
  Apart from this, you need to analyze the data with regards to the given task to identify patterns that are not useful. For instance, consider that you are building a search engine's result ranking system. Cases where the user clicks on a search result are considered positive examples. In comparison, those with just an impression are considered negative examples. You might see that the training data consist of a lot of bot traffic apart from the real user interactions. Bot traffic would just contain impressions and no clicks. This would introduce a lot of wrong negative examples. So we need to exclude them from the training data so that our model doesn't learn from wrong examples.



- **Removing bias**

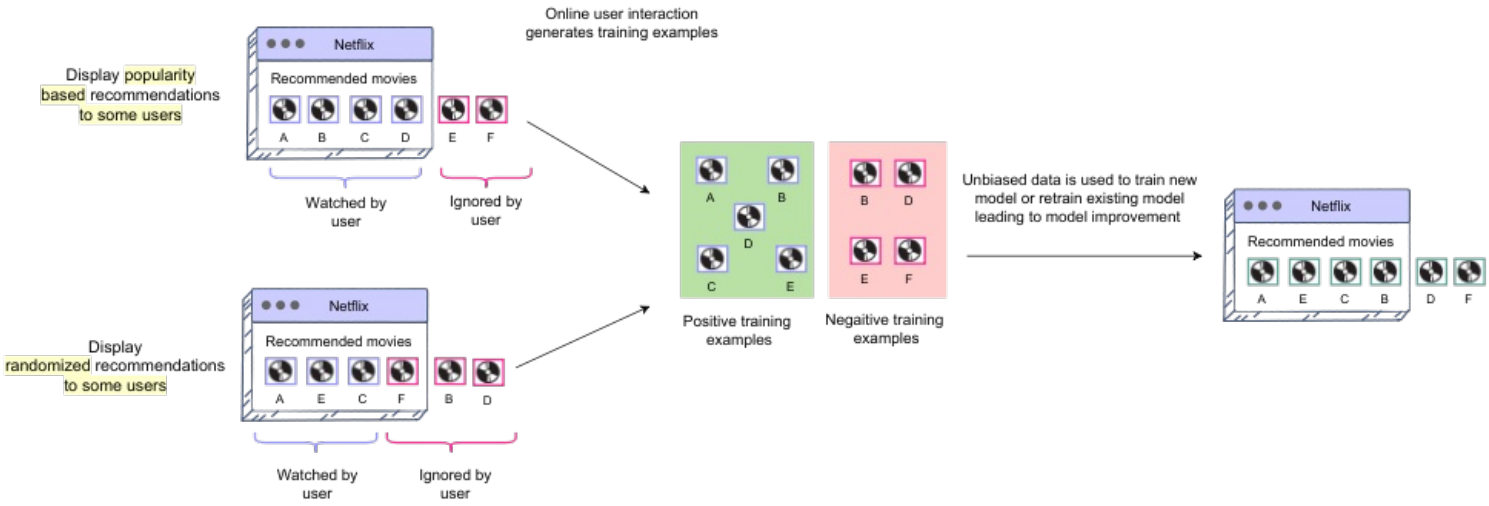
  When we are generating training data through online user engagement, it may become biased. Removing this bias is critical. Let's see why by taking the example of a movie recommender system like Netflix.

  The pre-existing recommender is showing recommendations based on popularity. As such, the popular movies always appear first and new movies, although they are good, always appear later on as they have less user engagement. Ideally, the user should go over the whole recommendation list so that he/she can discover the good, new movies as well. This would allow us to classify them as positive training examples so that the model can learn to put them on top, once re-trained. However, due to the user's time constraints, he/she would only interact with the topmost recommendations resulting in the generation of biased training data. The model hence trained, will continue considering the previous top recommendation to be the top recommendation this time too. Hence, the "rich getting richer" cycle will continue.



Biased training data generated through online user engagement

In order to break this cycle, we need to employ an exploration technique that explores the whole content pool (all movies available on Netflix). Therefore, we show "randomized" recommendations instead of "popular first" for a small portion of traffic for gathering training data. The users' engagement with the randomized recommendations provides us with unbiased training data. This data really helps in removing the current positional and engagement bias of the system.



Removing bias through exploration technique

- **Bootstrapping new items**

  Sometimes we are dealing with systems in which new items are added frequently. The new items may not garner a lot of attention, so we need to boost them to increase their visibility. For example, in the movie recommendation system, new movies face the cold start problem (https://en.wikipedia.org/wiki/Cold_start_(recommender_systems)). We can boost new movies by recommending them based on their similarity with the user's already watched movies, instead of waiting for the new movies to catch the attention of a user by themselves. Similarly, we may be building a system to display ads, and the new ads face the cold start problem. We can boost them by increasing their relevance scores a little, thereby artificially increasing their chance of being viewed by a person.

← Back

Performance and Capacity Considerati...

Next →

Online Experimentation

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/training-data-collection-strategies__practical-ml-techniquesconcepts__grokking-the-machine-learning-interview)

Online Experimentation

Irrespective of the problem you're working on, model experimentation and evaluation flow are always critical. In this lesson, we will go over the key steps and concepts in model experimentation and evaluation.

We'll cover the following

A successful machine learning system should be able to gauge its performance by testing different scenarios. This can lead to more innovations in the model design. For an ML system, “success” can be measured in numerous ways. Let’s take an example of an advertising platform that uses a machine-learning algorithm to display relevant ads to the user. The success of this system can be measured using the users’ engagement rate with the advertisement and the overall revenue generated by the system. Similarly, a search ranking system might take into account correctly ranked search results on SERP as a metric to claim to be a successful search engine. Let’s assume that the first version of the system (v0.1) has been created and deployed.

The initial version of the system is created

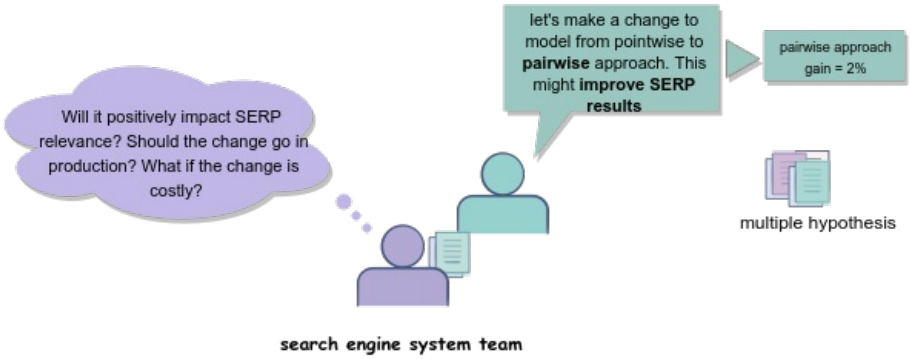
Hypothesis and metrics intuition

At any point in time, the team can have multiple hypotheses that need to be validated via experimentation.

Imagine, for instance, that the team designing an ad prediction system wants to test the hypothesis that increase in the neural network model depth (increase in hidden layers) or width (increase in activation units) will increase latency and capacity but will still have an overall positive effect on user engagement and net ad revenue.

Team desires to test multiple hypotheses to see the impact on the system

Similarly, a team working on designing a search engine wants to test the hypothesis that the pointwise algorithm instead of the pairwise algorithm would positively impact search relevance.



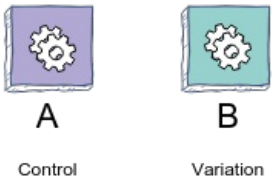
Team desires to test the hypotheses to see the impact on the system

So, to test the hypotheses, should the ML system v0.2 be created and deployed in the production environment? What if the hypothesis intuition is wrong and the mistake becomes costly?

This is where online experimentation comes in handy. It allows us to conduct controlled experiments that provide a valuable way to assess the impact of new features on customer behavior.

Running an online experiment

A/B testing is very beneficial for gauging the impact of new features or changes in the system on the user experience. It is a method of comparing two versions of a webpage or app against each other simultaneously to determine which one performs better. In an A/B experiment, a webpage or app screen is modified to create a second version of the same page. The original version of the page is known as the control and the modified version of the page is known as the variation.

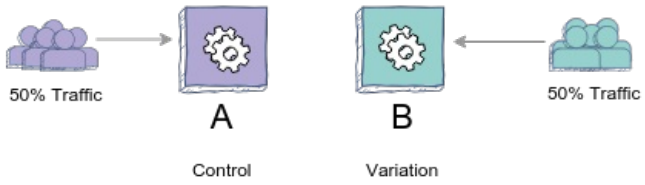


A/B testing

We can formulate the following two hypotheses for the A/B test:

- **The null hypothesis**, H0 is when the design change will not have an effect on variation. If we fail to reject the null hypothesis, we should not launch the new feature.
- **The alternative hypothesis**, H1 is alternate to the null hypothesis whereby the design change will have an effect on the variation. If the null hypothesis is rejected, then we accept the alternative hypothesis and we should launch the new feature. Simply put, the variation will go in production.

Now the task is to *determine if the number of successes in the variant is significantly better from the control*, i.e., if the conversion caused a positive impact on the system performance. This requires confidently making statements (using statistical analysis) about the difference in the variant sample, even if that difference is small. Before statistically analyzing the results, a power analysis test (https://en.wikipedia.org/wiki/Power_of_a_test) is conducted to determine how much overall traffic should be given to the system, i.e., the minimum sample size required to see the impact of conversion. Half of the traffic is sent to the control, and the other half is diverted towards the variation.



Traffic split evenly between version A and B

Measuring results

As visitors are served with either the control or variation/test version of the app, and their engagement with each experience is measured and analyzed through statistical analysis testing. Note that unless the tests are statistically significant, we cannot back up the claims of one version winning over another.

Computing statistical significance

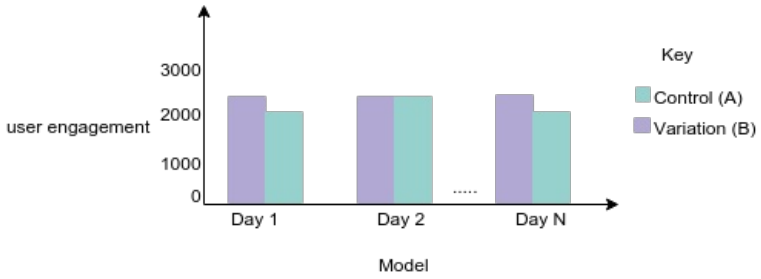
P-value (https://en.wikipedia.org/wiki/P-value) is used to help determine the statistical significance of the results. In interpreting the p-value of a significance test, a significance level (alpha) must be specified.

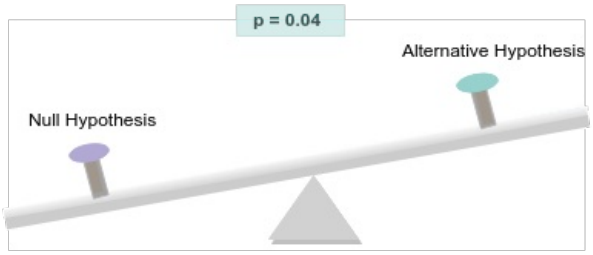
> The significance level is a boundary for specifying a statistically significant finding when interpreting the p-value. A commonly used value for the significance level is 5% written as 0.05.

The result of a significance test is claimed to be “statistically significant” if the p-value is less than the significance level.

- p <= alpha: reject H0 - launch the new feature
- p > alpha: fail to reject H0 - do not launch the new feature

If an A/B test is run with the outcome of a significance level of 95% (p-value ≤ 0.05), there is a 5% probability that the variation that we see is by chance.





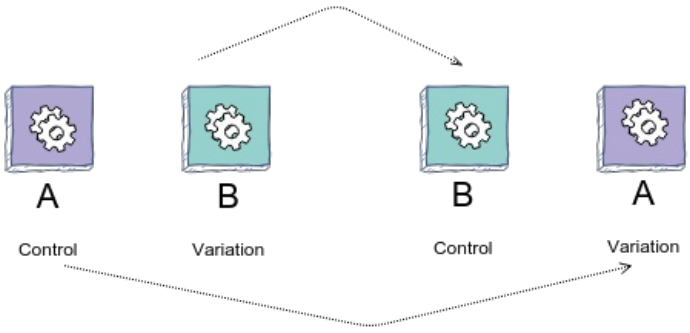
Statistical test analysis shows system B(variation) outperforms system A(control).

Measuring long term effects

In some cases, we need to be more confident about the result of an A/B experiment when it is overly optimistic.

Back Testing

Let's assume that variation improved the overall system performance by 5% when the expected gain was 2%. In the case of the ads prediction system, we can say that the rate of user engagement with the ad increased by 5% in variation (system B). This surprising change puts forth a question. Is the result overly optimistic? To confirm the hypothesis and be more confident about the results, we can perform a backtest (https://en.wikipedia.org/wiki/Backtesting). Now we change criteria, system A is the previous system B, and vice versa.



Backtest

We will check all potential scenarios while backtesting:

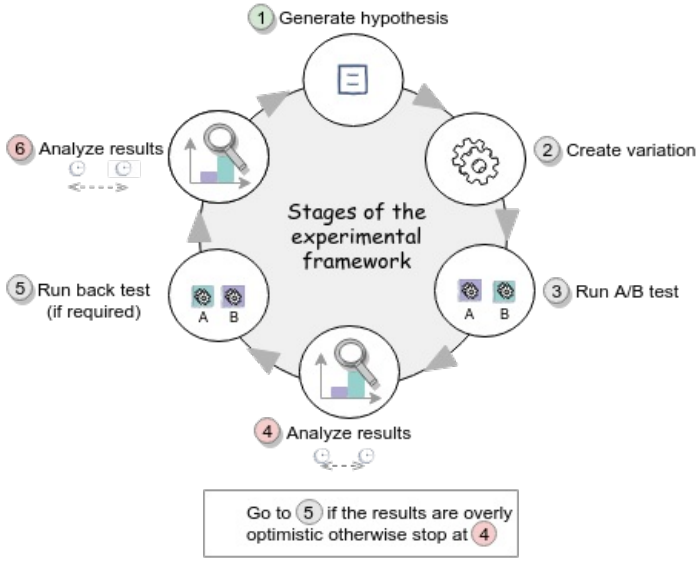
Do we lose gains? Is the gain caused by an A/B experiment equal to the loss by B/A experiment? Assume that the A/B experiment gave a gain of 5% and B/A experiment gave a loss of 5%. This will ensure that the changes made in the system improved performance.

Long-running A/B tests

In a few experiments, one key concern could be that the experiment can have a negative long term impact since we do A/B testing for only a short period of time. Will any negative effects start to appear if we do a long term assessment of the system subject to variation?

For example, suppose that for the ad prediction system, the revenue went up by 5% when we started showing more ads to users but this had no effect on user retention. Will users start leaving the platform if we show them significantly more ads over a longer period of time? To answer this question, we might want to have a long-running A/B experiment to understand the impact.

The long-running experiment, which measures long-term behaviors, can also be done via a backtest. We can launch the experiment based on initial positive results while continuing to run a long-running backtest to measure any potential long term effects. If we can notice any significant negative behavior, we can revert the changes from the launched experiment.



Experimental framework stages

← Back

Training Data Collection Strategies

Next →

Embeddings

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/online-experimentation__practical-ml-techniquesconcepts__grokking-the-machine-learning-interview)

Embeddings

Let's learn what embeddings are and how do we generate them using machine learning techniques.

We'll cover the following

Embeddings

Embeddings enable the encoding of entities (e.g., words, docs, images, person, ad, etc.) in a low dimensional vector space such that it captures their semantic information. Capturing semantic information helps to identify related entities that occur close to each other in the vector space.

This representation of entities in a lower-dimensional vector space has been of massive help in various ML-based systems. The use of embeddings has seen a major increase because of the recent surge in the use of neural networks and transfer learning.

Usually, they are generated using neural networks. A **neural network** architectures can be set up easily to learn a dense representation of entities. We will go over a few of such architectures later in this lesson.

Transfer learning (https://www.educative.io/collection/page/10370001/6237869033127936/5927951246819328) refers to transferring information from one ML task to another. Embeddings easily enable us to do that for common entities among different tasks. For example, Twitter can build an embedding for their users based on their organic feed interactions and then use the embeddings for ads serving. Organic interactions are generally much greater in volume compared to ads interactions. This allows Twitter to learn user interests by organic feed interaction, capture it as embedding, and use it to serve more relevant ads.

Another simple example is training word embeddings (like Word2vec) from Wiki data and using them as spam-filtering models.

In this lesson, we will go through some general ways of training neural networks to learn embeddings, using real-world example scenarios of their usage.

Text embeddings

We will go over two popular text term embeddings generation models and examples of their utilization in different ML systems.

Word2vec

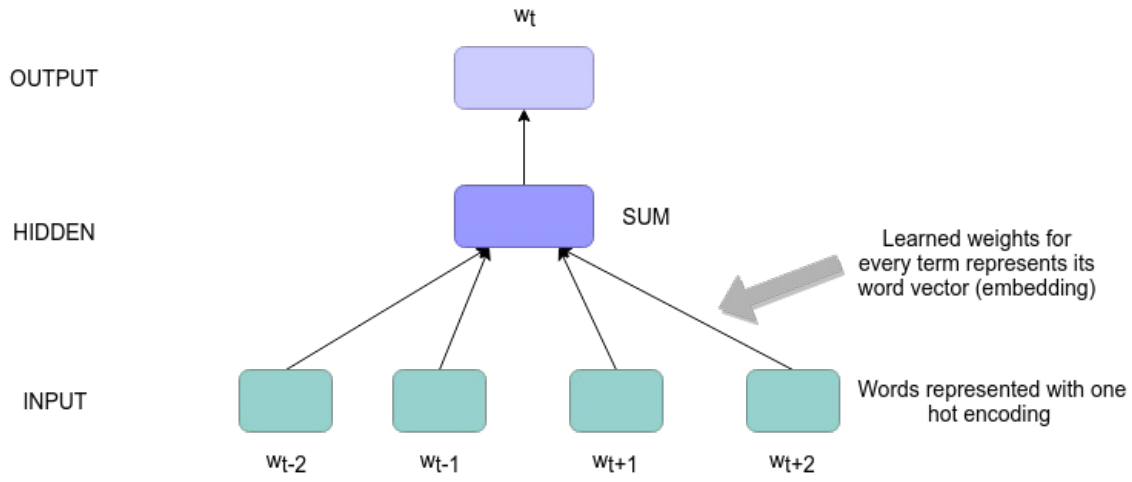
Word2vec produces word embeddings by using shallow neural networks (having a single hidden layer) and self-supervised learning from a large corpus of text data. Word2vec is self-supervised as it trains a model by predicting words from other words that appear in the sentence(context). So, it can utilize tons of text data available in books, Wikipedia, blogs, etc. to learn term representation.

Representing words with a dense vector is critical for the majority of Natural language processing (NLP) tasks. Word2vec uses a simple but powerful idea to use neighboring words to predict the current word and in the process, generates word embeddings. Two networks to generate these embeddings are:

1. **CBOW:** Continuous bag of words (CBOW) tries to predict the current word from its surrounding words by optimizing for following loss function:

$$Loss = -log\ (\text{p}(w_t \,|\, w_{t-n},...,w_{t-1}, w_{t+1}, w_{t+n}))$$

where n is the size of our window to look for the corresponding word. It uses the entire contextual information as one observation while training the network. Utilizing the overall context information to predict one term helps generate embeddings with the smaller training dataset. The architecture would look like the following:

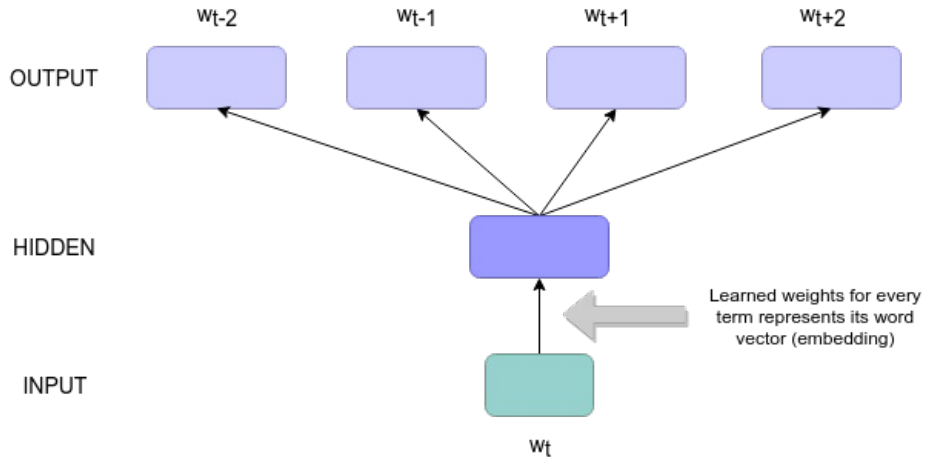


Word2vec using CBOW of window size two, i.e., predicting current word using surrounding words

2. **Skipgram:** In this architecture, we try to predict surrounding words from the current word. The loss function will now look like:

$$Loss = -log\ (\text{p}(w_{t-n},...,w_{t-1}, w_{t+1}, w_{t+n} \,|\, w_t))$$

where n is now the size of surrounding words that we are trying to predict using the current word. In training this network, each context pair will result in a different training example that the model tries to predict. This architecture is most helpful when we have a large training set. The model architecture now looks like:



Word2vec using Skip-gram to predict two surrounding words to current word

Example

Any machine learning task that wants to utilize text terms can benefit from this dense embedding vector, which captures word semantic meanings.

Let's assume that we want to predict whether a user is interested in a particular document given the documents that they have previously read. One simple way of doing this is to represent the user by taking the mean of the Word2vec embeddings of document titles that they haved engaged with. Similarly, we can represent the document by the mean of its title term embeddings. We can simply take the dot product of these two vectors and use that in our ML model.

Another way to accomplish this task is to simply pass the user and the document embedding vector to a neural network to help with the learning task.

Context-based embeddings

Once trained, Word2vec embeddings have a fixed vector for every term. So, a Word2vec embedding doesn't consider the context in which the word appears to generate its embedding. However, words in a different context can have very different meanings. For example, consider these two sentences:

1. I'd like to eat an apple.
2. Apple makes great products.

Word2vec will give us the same embedding for the term "apple" although it points to completely different objects in the above two sentences.

So, contextualized information can result in different meanings of the same word, and context-based embeddings look at neighboring terms at embedding generation time. This means that we have to provide contextual information (neighboring terms) to fetch embeddings for a term. In a Word2vec case, we don't need any context information at the embeddings fetch time as embedding for each term was fixed.

Two popular architectures used to generate word context-based embedding are:

1. Embeddings from Language Models (ELMo)
2. Bidirectional Encoder Representations from Transformers (BERT)

The idea behind **ELMO** is to use the bi-directional LSTM model to capture the words that appear before and after the current word.

BERT uses an attention mechanism and is able to see all the words in the context, utilizing only the ones (i.e., pay more attention) which help with the prediction.

We will have an in-depth explanation of these models and architectures in the entity linking chapter (https://www.educative.io/collection/page/10370001/6237869033127936/5575618289729536). Entity recognition and linking are also a great examples of how contextual embeddings can be very effective; we will discuss this in detail in that chapter.

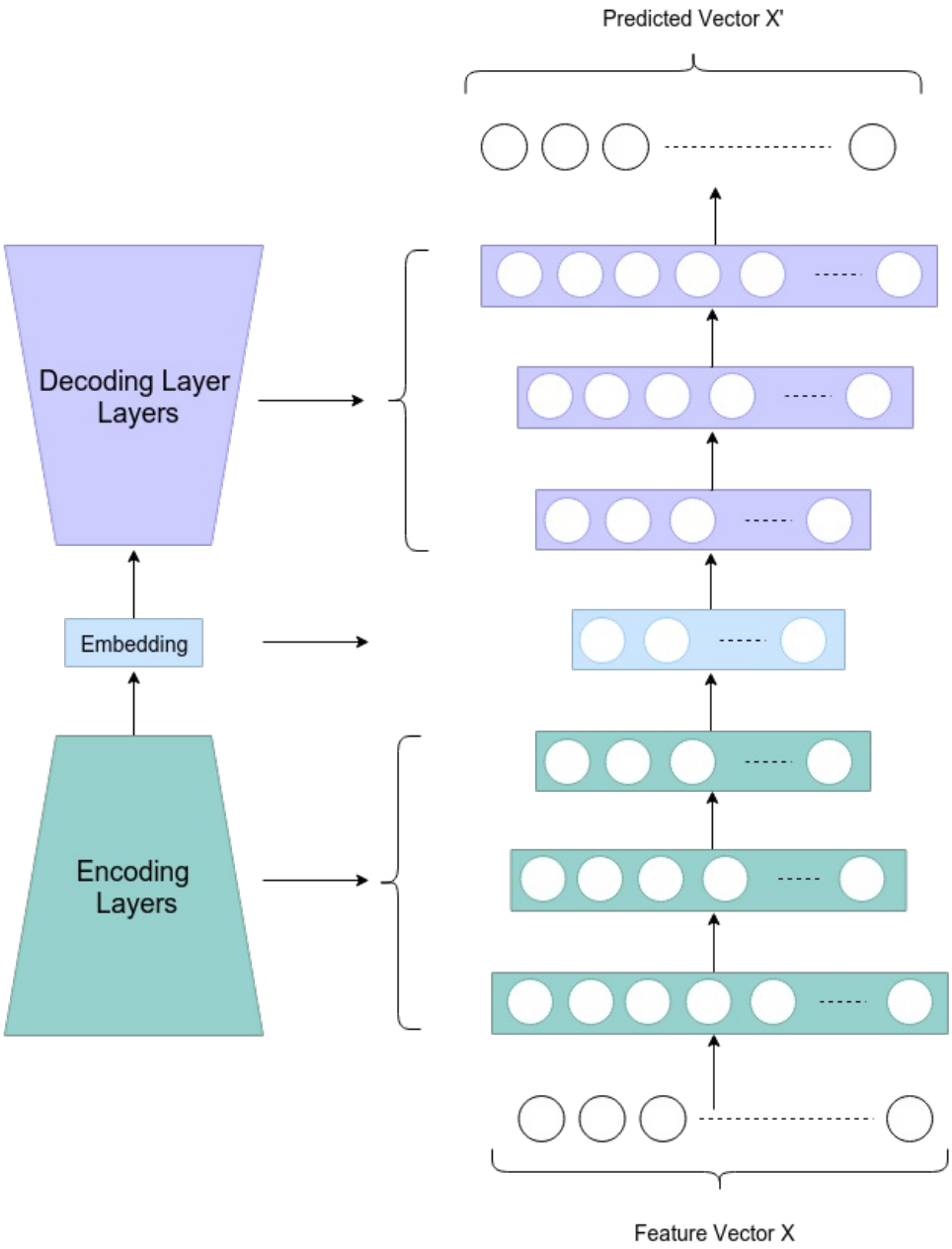
Visual embedding

Let's discuss a couple of interesting ways to generate image embedding.

Auto-encoders

Auto-encoders use neural networks consisting of both an encoder and a decoder. They first learn to compress the raw image pixel data to a small dimension via an encoder model and then try to de-compress it via a decoder to re-generate the same input image. The last layer of encoder determines the dimension of the embedding, which should be sufficiently large to capture enough information about the image so that the decoder can decode it.

The combined encoder and decoder tries to minimize the difference between original and generated pixels, using backpropagation to train the network. The network will look like the following:



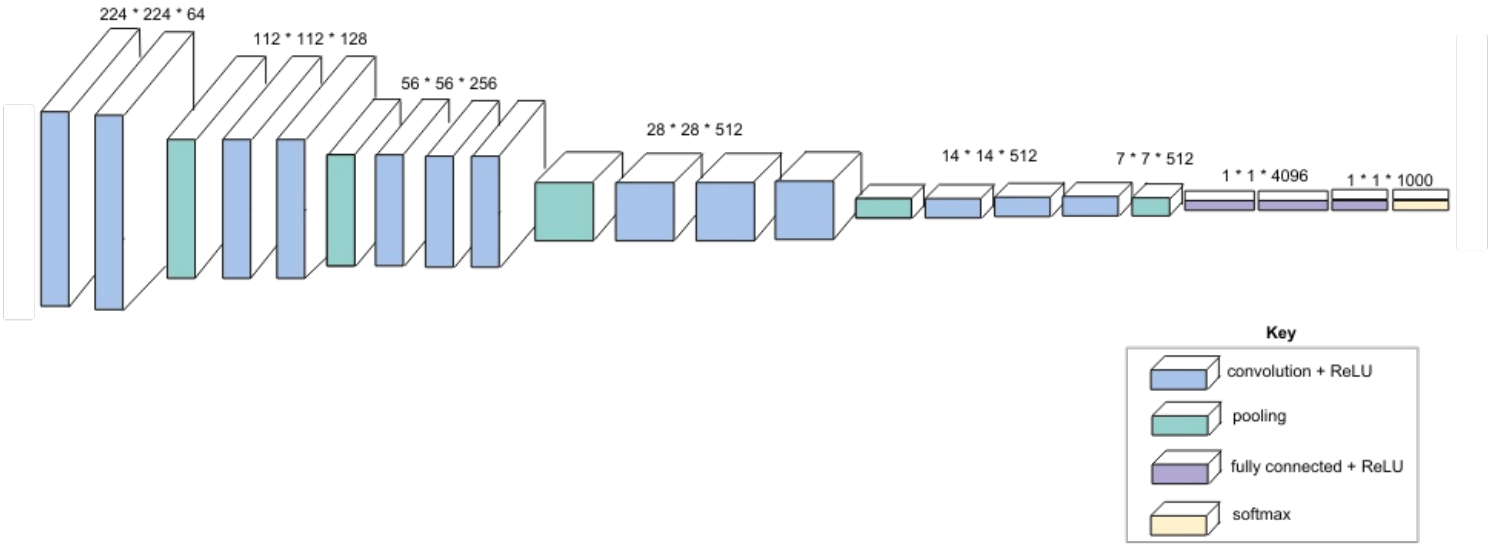
Auto-encoder model architecture for generating visual embeddings

Once we have trained the model, we only use the encoder (first N network layers) to generate embeddings for images.

Auto-encoders are also an example of self-supervised learning, like Word2vec, as we can use an image data set without any label to train the model and generate image embeddings.

Visual supervised learning tasks

Visual supervised learning tasks such as image classification or object detection, are generally set up as convolution layers, pooling layers, and fully connected network layers, followed by final classification(softmax) layers. Let's consider the example of the *ImageNet VGG16 model* that is shown in the figure below. The input passes through a set of convolution, pooling, and fully connected layers to the last softmax layer for the final classification task. The penultimate layer before softmax captures all image information in a vector such that it can be used to classify the image correctly. So, we can use the penultimate layer value of a pre-trained model as our image embedding.



VGG16 architecture

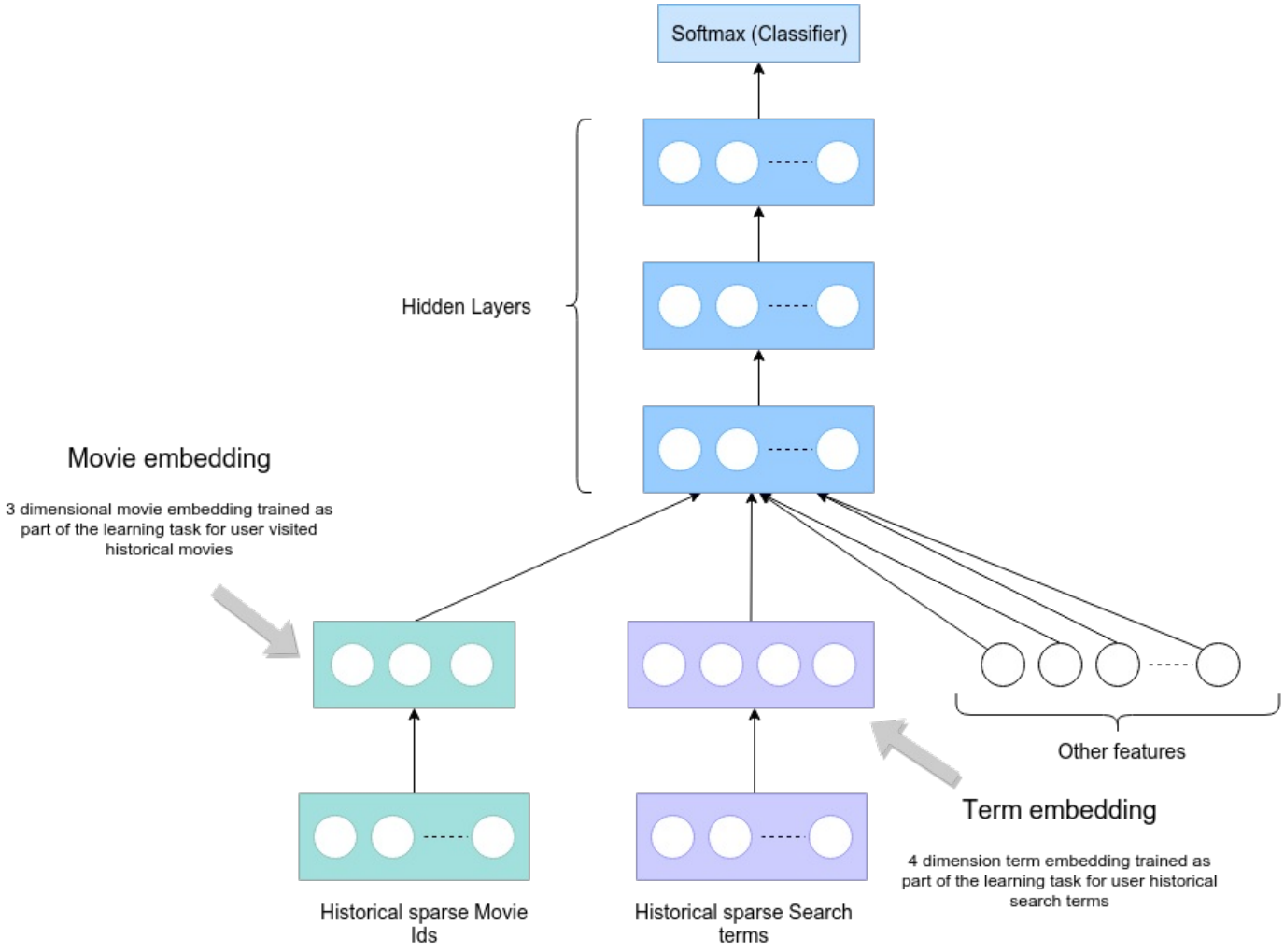
An **example** of image embedding usage could be to find images similar to a given image.

Another example is an image search problem where we want to find the best images for given text terms, e.g. query “cat images”. In this case, image embedding along with query term embedding can help refine search relevance models.

Learning embeddings for a particular learning task

Most of our discussion so far has been about training a general entity embedding that can be used for any learning task. However, we can also embed an entity as part of our learning task. The advantage of this embedding is a specialized one for the given prediction task. One important assumption here is that we have enough training data to be able to learn such representation during model training. Another consideration is that training time for learning the embedding as part of the task will be much higher compared to utilizing a pre-trained embedding.

Let’s consider an example where we are trying to predict whether a user will watch a particular movie based on their historical interactions. Here, utilizing movies that the user has previously watched as well as their prior search terms can be very beneficial to the learning task. We can do this by embedding sparse vector of movies and terms in the network itself, as shown in the image below. We will also discuss this specialized embedding usage in detail in the recommendation system chapter (https://www.educative.io/collection/page/10370001/6237869033127936/5559503958310912).



Network/Relationship-based embedding

Most of the systems have multiple entities, and these entities interact with each other. For example, Pinterest has users that interact with Pins, YouTube has users that interact with videos, Twitter has users that interact with tweets, and Google search has both queries and users that interact with web results.

We can think of these interactions as relationships in a graph or resulting in interaction pairs. For the above example, these pairs would look like:

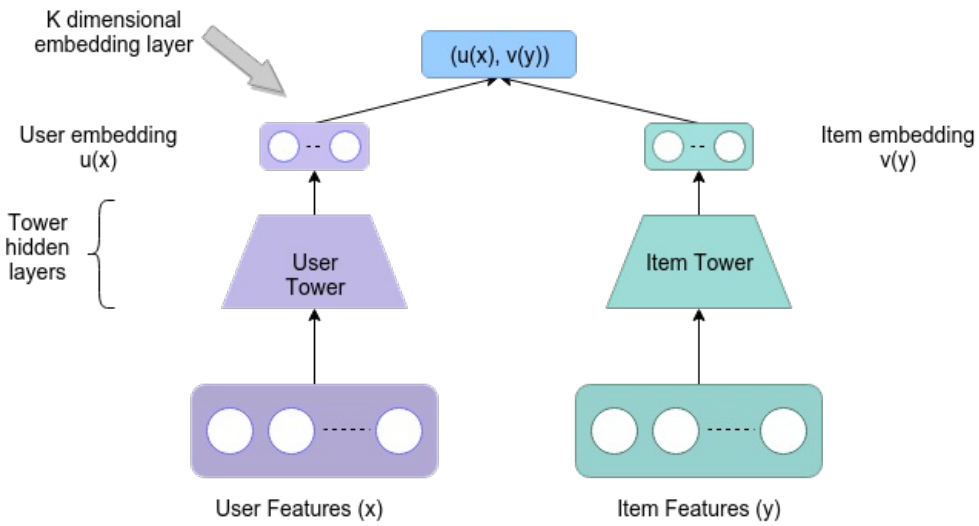
1. (User, Pin) for Pinterest
2. (User, Video) for YouTube
3. (User, Tweet) for Twitter
4. (Query, Webpage) for Search
5. (Searcher, Webpage) for Search

In all the above scenarios, the retrieval and ranking of results for a particular user (or query) are mostly about predicting how close they are. Therefore, having an embedding model that projects these documents in the same embedding space can vastly help in the retrieval and ranking tasks of recommendation, search, feed-based, and many other ML systems.

We can generate embeddings for both the above-discussed pairs of entities in the same space by creating a two-tower neural network model that tries to encode each item using their raw features. The model optimizes the inner product loss such that positive pairs from entity interactions have a higher score and random pairs have a

lower score. Let's say the selected pairs of entities (from a graph or based on interactions) belong to set A. We then select random pairs for negative examples. The loss function will look like

$$Loss = max(\sum_{(u,v)\in A} dot(u,v) - \sum_{(u,v)\notin A} dot(u,v))$$



Two tower model to optimize inner product loss for user and item embeddings

---

Back

Online Experimentation

Next

Transfer Learning

Mark as
Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/embeddings__practical-ml-techniquesconcepts__grokking-the-machine-learning-interview)

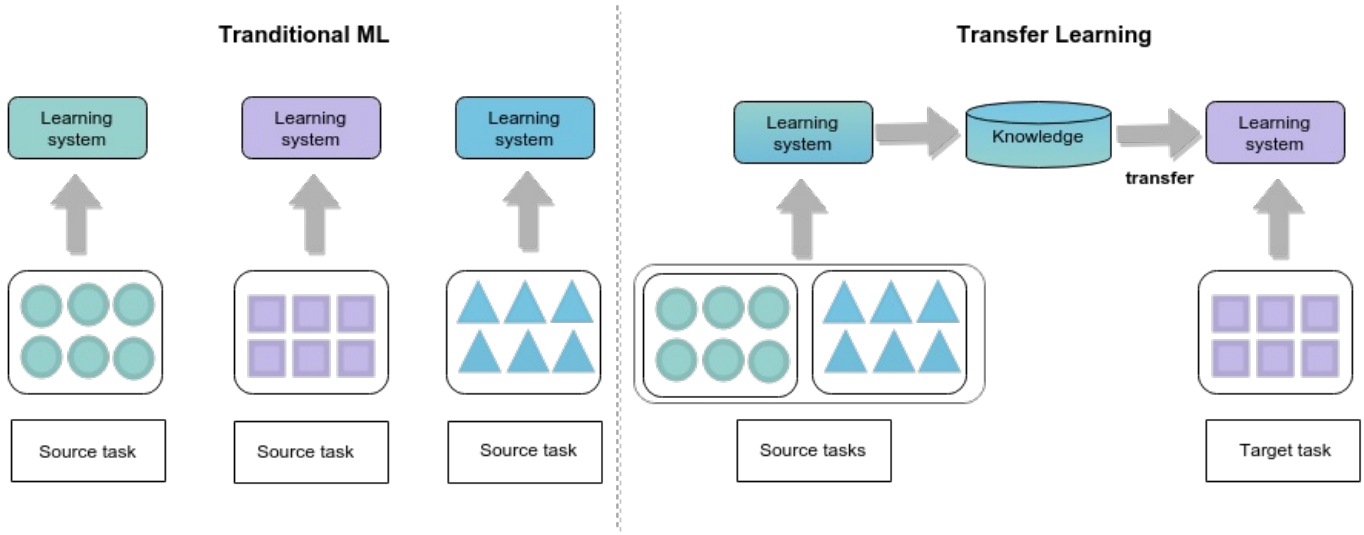
Transfer Learning

An introduction to transfer learning, its importance, and applications.

We'll cover the following

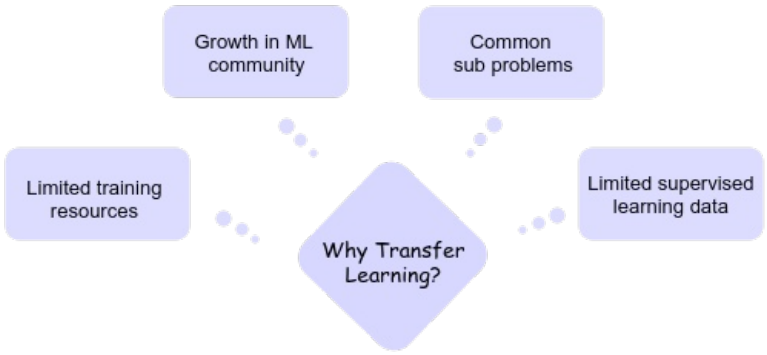
What is transfer learning?

Transfer learning is the task of using a pre-trained model and applying it to a new task, i.e., transferring the knowledge learned from one task to another. This is useful because the model doesn't have to learn from scratch and can achieve higher accuracy in less time as compared to models that don't use transfer learning.



Motivation

The use of transfer learning in the machine learning domain has surged in the last few years. The following are the top reasons:



Motivation for transfer learning

1. **Growth in the ML community and knowledge sharing**: The research and investments by top universities and tech companies have grown exponentially in the last few years and there is also a strong desire to share state-of-the-art models and datasets with the community. This allows people to utilize pre-trained models in a specific area bootstrap quickly.

2. **Common sub-problems**: Another key motivator is that many problems share common sub-problems, e.g., in all visual understanding and prediction areas, tasks such as finding edges, boundaries, and background are common sub-problems. Similarly, in the text domain, the semantic understanding of textual terms can be helpful in almost all problems where the user is represented by text terms, including search, recommendation systems, ads, etc.

3. **Limited supervised learning data and training resources**: Many real-world applications are still mapped onto supervised learning problems where the model is asked to predict a label. One key problem is the limited amount of training data available for models to generalize well. One key advantage of doing transfer learning is that we have the ability to start learning from pre-trained models, and hence, we can utilize the knowledge from similar domains.

   Self-supervised learning models are able to utilize massive available datasets for text and image representation, e.g., Word2vec embedding models don't need any manual labels and can use the books and Wikipedia data to build a semantic understanding of terms effectively. Once we train a model for a certain representation, it can be utilized and help in many other supervised learning tasks.

   Transfer learning also optimizes training resources, and it helps teams that don't have massive computing resources available. For instance, Google can train a BERT model on billions of examples with its massive computing power, but others are going to find it challenging to train similar optimized models. With transfer learning, we don't have to reinvest those resources and can just plug in the output of the BERT model or use it as a sub-model in our training process. We discussed this concept in detail earlier in our embedding (https://www.educative.io/collection/page/10370001/6237869033127936/6130870193750016) lesson discussion.

Techniques for transfer learning utilization

The transfer learning technique can be utilized in the following ways:

- **Extract features from useful layers**

  Keep the initial layers of the pre-trained model and remove the final layers. Add the new layer to the remaining chunk and train them for final classification.

- **Fine-tuning**

  Change or tune the existing parameters in a pre-trained network, i.e., optimizing the model parameters during training for the supervised prediction task. A key question with fine-tuning the model is to see how many layers can we freeze and how many final layers we want to fine-tune. This requires understanding the network structure of the model and role of each layer, e.g., for the image classification model we used in the Image data example, once we understand the convolution, pooling, and fully connected layers, we can decide how many final layers we need to fine-tune for our model training process.

Transfer learning technique can be utilized in one or both of the above ways depending on the following two factors:

1. **Size of our supervised training dataset**
   How much labeled data do we possess to optimize the model? Do we have 100k examples, 1 million examples, 10 million examples? This is an important question for deciding on the approach that we want to use in utilizing the pre-trained model.

   Training data is limited: In case of a limited amount of specialized training data, we can either go with the approach of freezing all the layers and using the pre-trained model for feature generation or fine-tuning only the final layers.

   Training data is plenty: If we have a significant amount of training data (e.g. one million+ examples), we have the choice to play around with multiple ideas. We can start with just freezing the model, fine-tuning only final layers, or we can retrain the whole model to adjust weights for our specialized task.

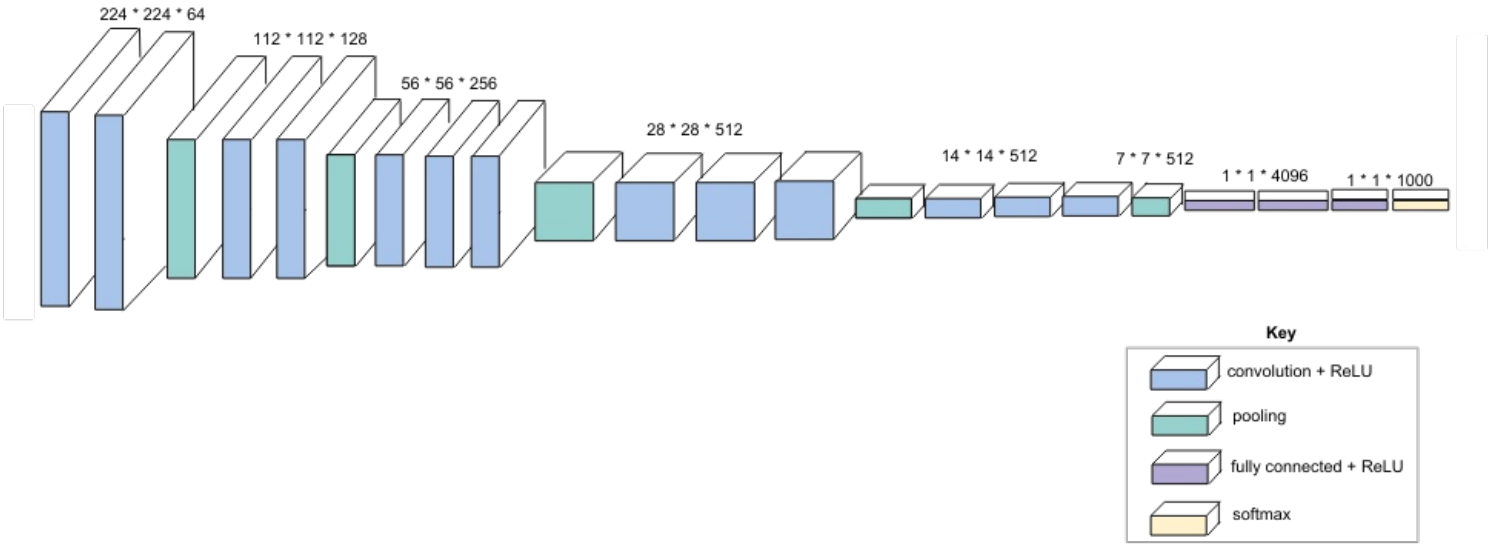
2. **Similarity of prediction tasks**

   The similarity of learning tasks can also guide us on whether we can simply use the model as it is or need to fine-tune the model for our new prediction task. For example, if we built a classifier for cars and now we want to use it for trucks, there is a good chance that many of the features are going to be common and we don't have to fine-tune much. Here, we can utilize the pre-trained model as it is and build our models on top of it (i.e., utilizing the output of pre-trained models as features).

Applications

Computer vision problems

Let's go over an example problem where we are trying to build a classifier for medical imaging data and we have 100k manual labelled examples for training our model. Given significant amount of research done in ImageNet data classifier, we can pick one pre-trained ImageNet classifier and start building on top of it.



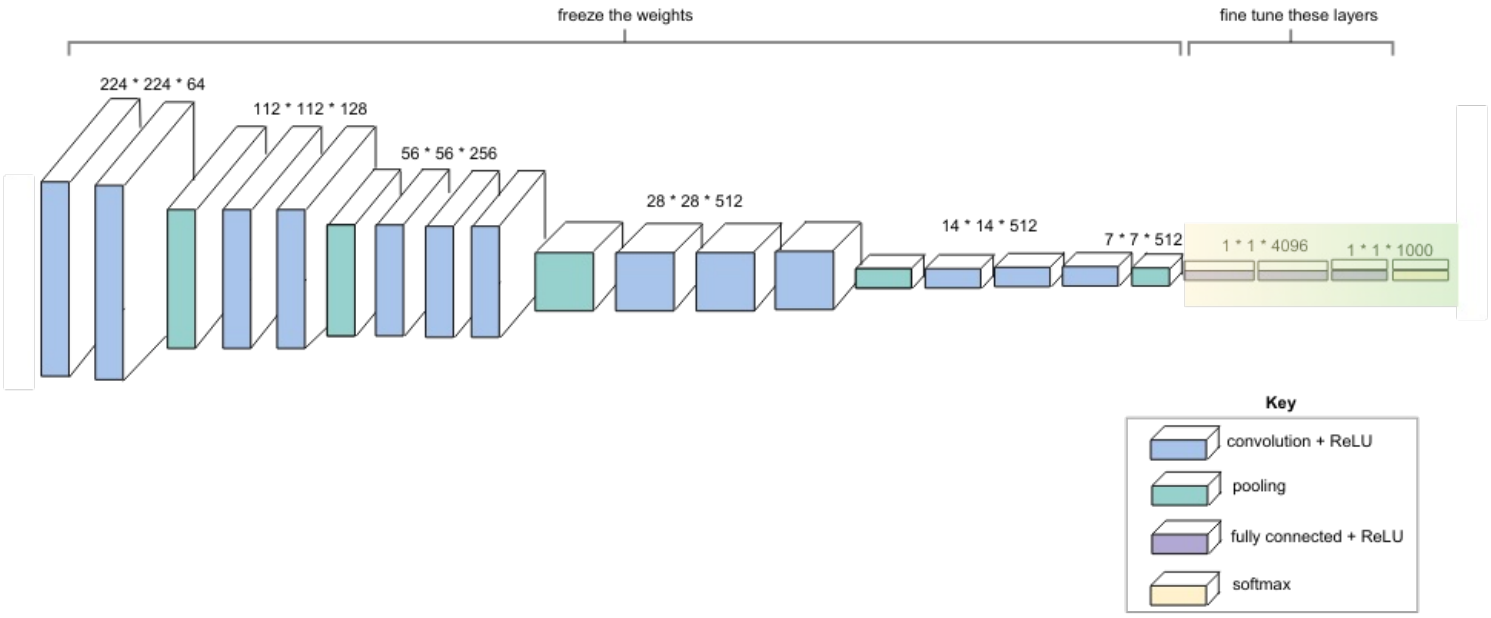
VGG16 architecture

The convolutional filters in a trained convolutional neural network (CNN) are arranged in a kind of hierarchy. The filters in the first layer often detect edges or blocks of color. The second layer's filters can detect features like shapes. All of them are very general features that are useful in analyzing any image in any dataset. The filters in the last layers are more specific. Let's go over all of the freezing layers, fine-tuning a few layers, and fine-tuning the whole model scenarios:

- **Case 1: Fine-tuning a few layers**

  If our prediction task is similar, there will be similar higher-level features or layers output. Therefore most or all of the pre-trained model layers already have relevant information about the new data set and should be kept. We will *freeze the weight of most of the starting layers* of the pre-trained model and fine-tune only the end layers.

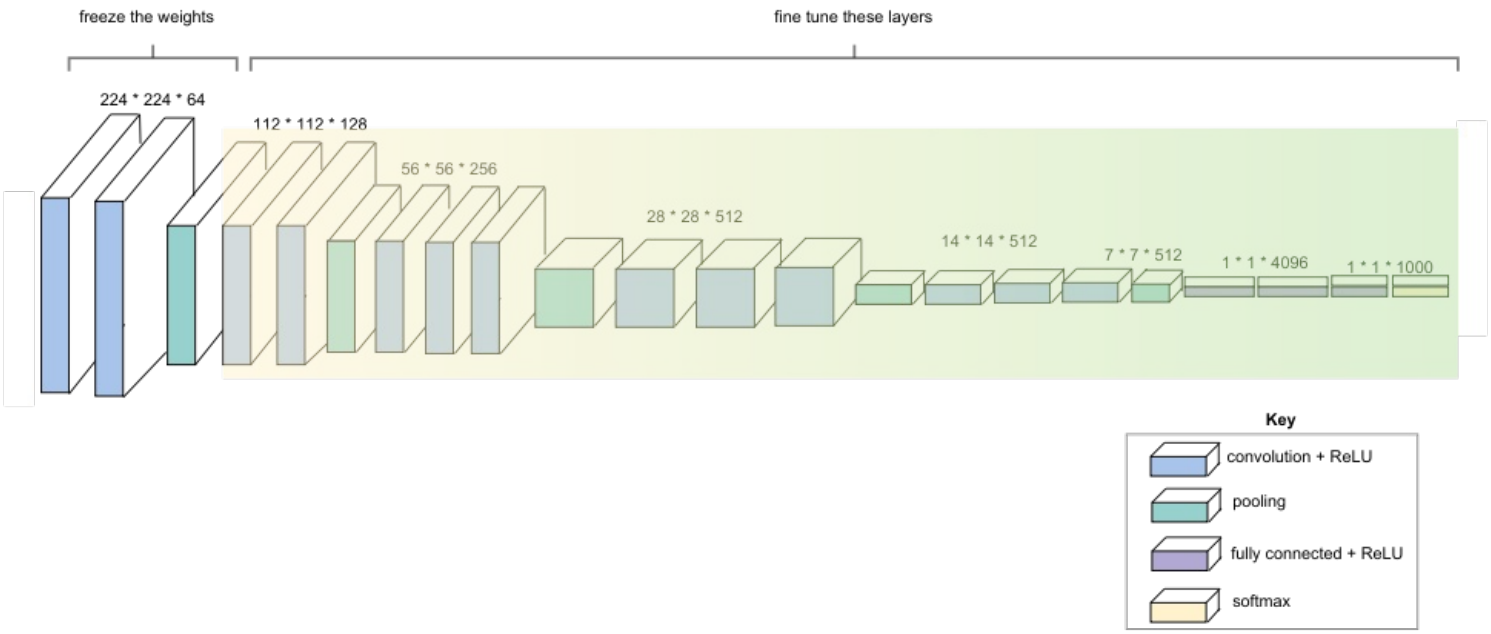
  This approach will also be most viable if our labeled training data is limited as it would be hard to re-tune all layers based on that limited data set.



Fine-tuning a few layers

- **Case 2: Fine tuning more layers**

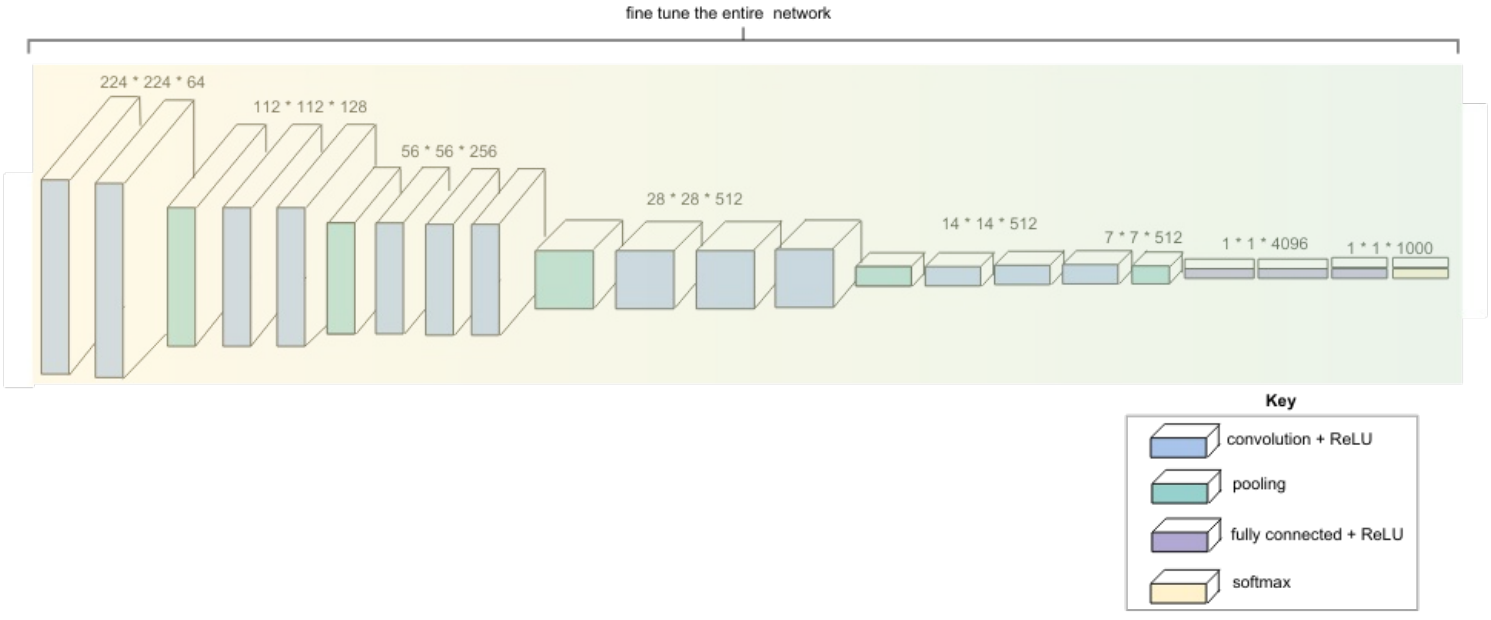
If we have significant amount of labelled examples and our learning tasks have commonalities but few differences as well, it would make sense to go deeper in fine tuning our pre-trained model. We will *freeze the weights of the first few layers* and fine-tune the weights of the remaining end layers to optimize the model for our new learning task on medical image dataset.



Fine-tuning most of the end layers

- **Case 3: Fine tuning the entire model**

  If the new data set is larger, then we will load the weights of the pre-trained model and *fine-tune the entire network.* This will definitely increase our training time as well but should help us optimize our learning task when we have significant training data.



Fine tuning the entire model

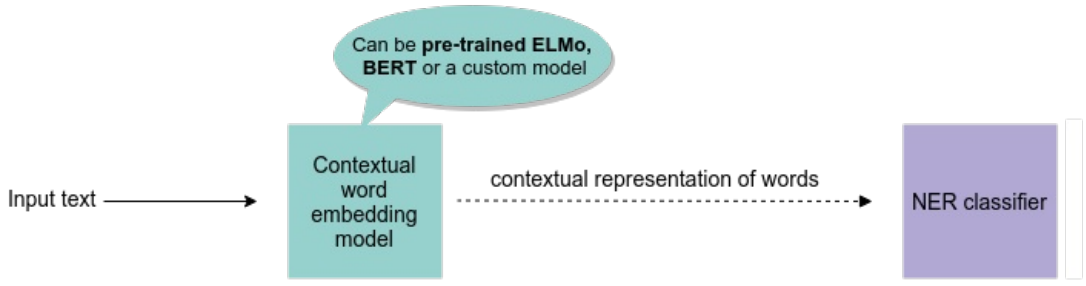
More details of transfer learning using image data are explained while modeling the solution for the image segmentation problem (https://www.educative.io/courses/grokking-the-machine-learning-interview/g77j6mXv7R9).

Natural language processing(NLP)

In many of NLP learning tasks such as language understanding, speech recognition, entity recognition, language generation, semantic understanding, etc. as well as other problems that are based on search, one major need is to represent our text terms in a way that they capture the semantic meaning.

For this, we need to generate the *dense representation* of textual terms. A few of the popular term representation models that use a self-supervised learning approach, trained on massive datasets are word2vec, BERT, and ELMO. The term representation based on these models capture their semantic meanings. Hence, we can transfer knowledge from this learned task in many of the NLP tasks.

Through the transfer learning approach, we can now utilize these embeddings in a NER classifier, spam detector classifier, search ranking, language understanding, etc. and can significantly improve the quality of these ML models.



BERT/ELMo can be as an embedding generator

More details can be learned about these approaches while modeling the solution for the entity linking problem (https://www.educative.io/courses/grokking-the-machine-learning-interview/YVmpGv0D01O#elmo).

← Back

Embeddings

Next →

Model Debugging and Testing

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/transfer-learning__practical-ml-techniquesconcepts__grokking-the-machine-learning-interview)

Model Debugging and Testing

An introduction to model debugging and how it is conducted for a machine learning system.

We'll cover the following

Let's go over different phases in the development of a machine learning system, potential issues that we can face, and how to debug and fix them.

There are two main phases in terms of the development of a model that we will go over:

- Building the first version of the model and the ML system.
- Iterative improvements on top of the first version as well as debugging issues in large scale ML systems.

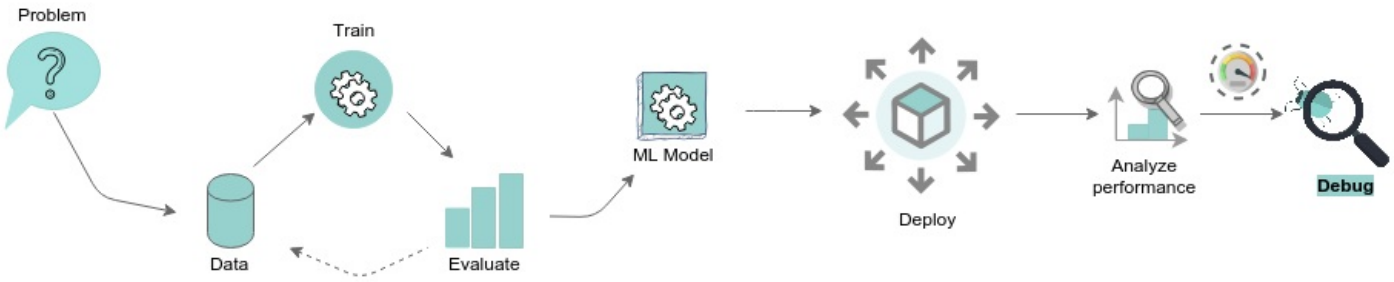
Building model v1

The goal in this phase is to build the 1st version of the model. Few important steps in this stage are:

- We begin by identifying a business problem in the first phase and mapping it to a machine learning problem.
- We then go onto explore the training data and machine learning techniques that will work best on this problem.
- Then we train the model given the available data and features, play around with hyper-parameters.
- Once the model has been set up and we have early offline metrics like accuracy, precision/recall, AUC, etc., we continue to play around with the various features and training data strategies to improve our offline metrics.
- If there is already a heuristics or rule-based system in place, our objective from the offline model would be to perform at least as good as the current system, e.g., for ads prediction problem, we would want our ML model AUC to be better than the current rule-based ads prediction based on only historical engagement rate.

It's important to get version 1 launched to the real system quickly rather than spending too much time trying to optimize it. For example, if our AUC is 0.7 and it's better than the current system with AUC 0.68, it's generally a better idea to take model online and then continue to iterate to improve the quality. The reason is primarily that

model improvement is an iterative process and we want validation from real traffic and data along with offline validation. We will look at various ideas that can help in that iterative development in the following sections.



ML Model Workflow

Deploying and debugging v1 model

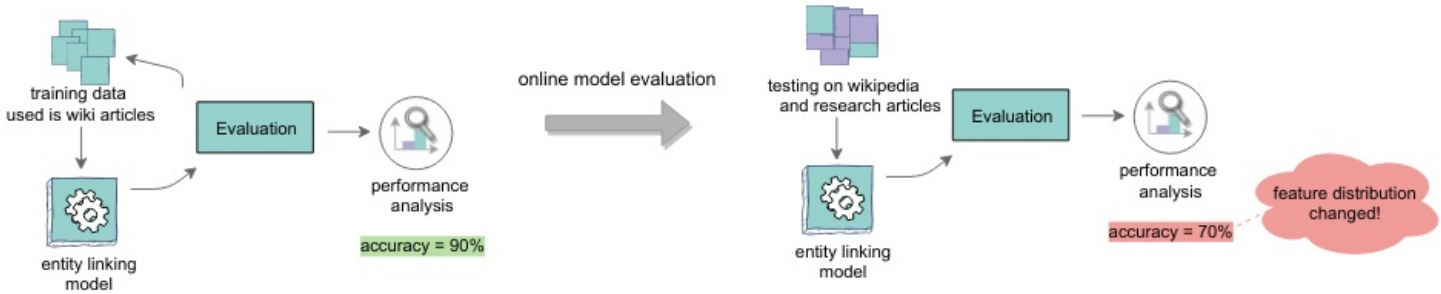
Some ML-based systems only operate in an offline setting, for example, detect objects from a large set of images. But, most systems have an online component as well, for example, building a search ranking ML model will have to run online to service incoming queries and ranking the documents that match the query.

In our first attempt to take the model online, i.e., enable live traffic, might not work as expected and results don't look as good as we anticipated offline. Let's look at a few failure scenarios that can happen at this stage and how to debug them.

Change in feature distribution

The change in the feature distribution of training and evaluation set can negatively affect the model performance. Let's consider an example of an Entity linking system that is trained using a readily available Wikipedia dataset. As we start using the system for real traffic, the traffic that we are now getting for finding entities is a mix of Wikipedia articles as well as research papers. Given the model wasn't trained on that data, its feature distribution would be a lot different than what it was trained on. Hence it is not performing as well on the research articles entity detection.

Another scenario could be a significant change in incoming traffic because of seasonality. Let's consider an example of a search system trained using data for the last 2 weeks of December, i.e., mostly holiday traffic. If we deploy this system in January, the queries that it will see will be vastly different than what it was trained on and hence not performing as well as we observed in our offline validation.

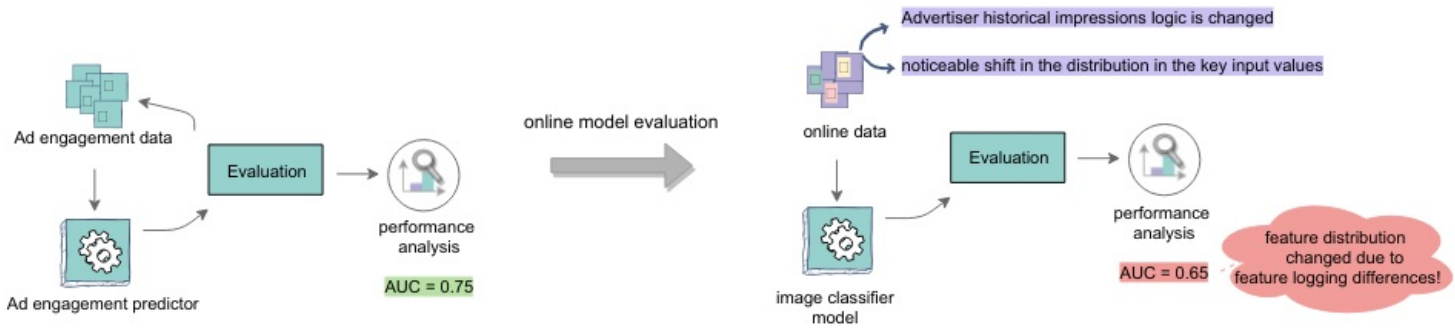


Entity linking model could not work online due to change in feature distribution

Feature logging issues

When the model is trained offline, there is an assumption that features of the model generated offline would exactly be the same when the model is taken online. However, this might not be true as the way we generated features for our online system might not exactly be the same. It's a common practice to append features offline to our training data for offline training and then add them later to the online model serving part. So, if the model doesn't perform as well as we anticipated online, it would be good to see if feature generation logic is the same for offline training as well as online serving part of model evaluation.

Suppose we build an ads click prediction model. The model is trained on historical ad engagement. We then deploy it to predict the ads engagement rate. Now the assumption is that the features generated for the model offline would exactly be the same for run-time evaluation. Let's assume that we have one important feature/signal for our model that's based on historical advertiser ad impressions. During training, we compute this feature by using the last 7 days' impression. But, the logic to compute this feature at model evaluation time uses the last 30 days of data to compute advertiser impressions. Because of this feature computed differently at training time and evaluation time, it will result in the model not performing well during online serving. It would be worth comparing the features used for training and evaluation to see if there is any such discrepancy.



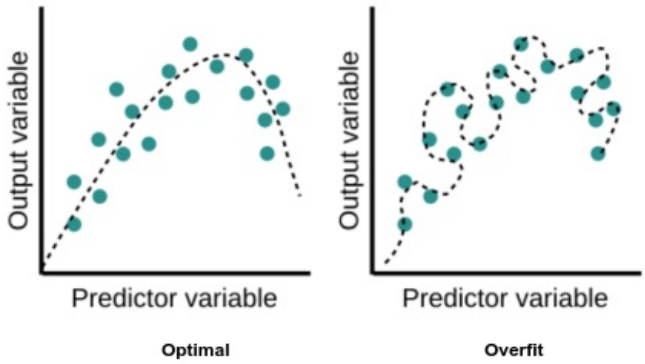
Ads prediction online metrics drop due to feature computation differences

Overfitting

Overfitting happens when a model learns the intrinsic details in the training data to the extent that it negatively impacts the performance of the model on new or unseen data.

If our model performance is lower in the live system but it still performs well on our training and validation set then there is a good chance that we have overfit our data by trying to improve the performance a bit too much. One good way of ensuring that we don't get into this problem is to use a hidden test set which is not used for tuning hyperparameters and only use it for final model quality measurement.

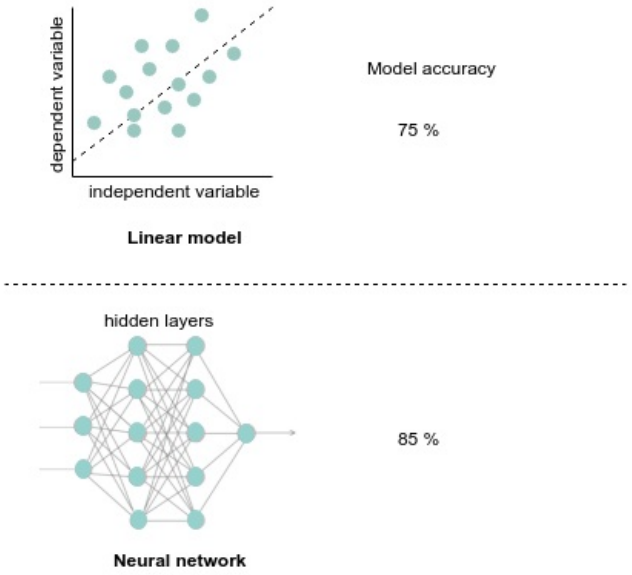
Another important part is to have a comprehensive and large test set to cover all possible scenarios in a fairly similar distribution to how we anticipate them in live traffic. For example, consider an image object prediction system whose test set only has large-sized objects - (covering 50% pixels or more of the image) for 90% samples and small-sized objects for 10% samples (covering 10% pixels or less). If the live traffic has the opposite distribution of large and small size objects, then the model might not perform well on the live set.



Overfit and optimal model

Under-fitting

One indication from training the model could be that the model is unable to learn complex feature interactions especially if we are using a simplistic model. So, this might indicate to us that using slightly higher-order features, introduce more feature interactions, or use a more complex /expensive model such as a neural network.



Neural network outperforms linear regression model because it can capture non-linear relationship between features

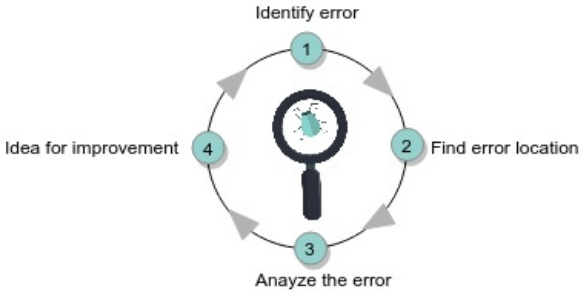
Iterative model improvement

As discussed above that the first version of the model deployed is generally not the most optimized one as our goal is to get the ML system out sooner to get feedback. Afterward, *Model debugging and testing* is generally done to come up with ideas to improve the quality of the model resulting in our metrics improvement.

Few cases that we discussed above regarding overfitting and under-fitting are still the questions that we should continue to ask during iterative model improvement but let's discuss a few more below.

The best way to iterative improve model quality is to start looking at failure cases of our model prediction and using that come up with the ideas that will help in improving model performance in those cases.

Let's go over some common types of ideas that will emerge as part of these failures examples debugging process.



Debugging flow

> Debugging in current context doesn't mean to optimize overall architecture and system design that we have deeply discussed in this course like how to set up a problem, setting up architecture etc. It comprises of general methods that we use to observe issues in current model to continue to optimize it.

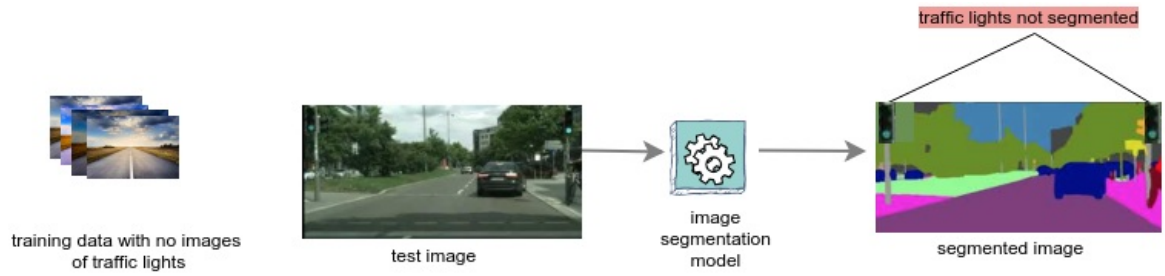
Missing important feature

Digging deeper into failures examples can identify missing features that can help us perform better in failures cases, e.g., consider a scenario where a movie actually liked by the user was ranked very low by our recommendation system. On debugging, we figure out that the user has previously watched two movies by the same actor, so adding a feature on previous ratings by the user for this movie actor can help our model perform better in this case.

Insufficient training examples

We may also find that we are lacking training examples in cases where the model isn't performing well. We will cater to all possible scenarios where the model is not performing well and update the training data accordingly. For example, for the image segmentation problem, the segmentation model is not able to segment the image when there are multiple traffic lights. One way to look at that point will be to count the number of such training examples that we have in our data set with multiple traffic lights.

So we will add more training examples of multiple traffic lights and enable the model to learn to segment multiple traffic lights better. The following illustration shows the above concept by taking an example of a test image from cityscapes dataset.



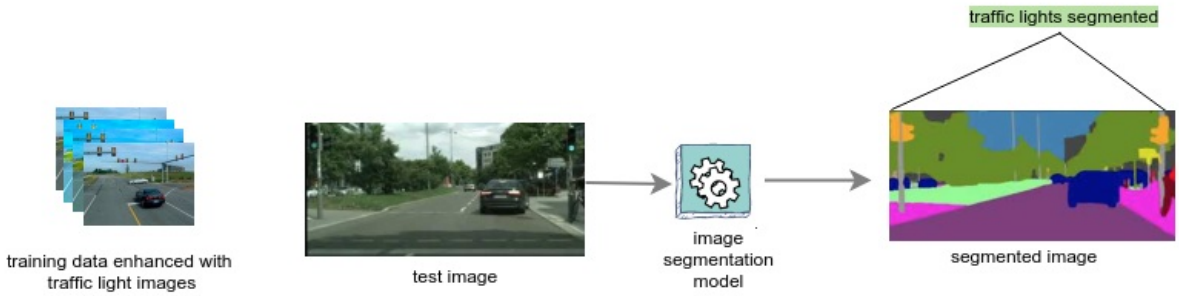


image segmentation model output performance increased in case of sufficient training data

Debugging large scale systems

In the case of debugging large scale systems with multiple components(or models), we need to see which part of the overall system is not working correctly. It could be done for one failure example or over a set of examples to see where the opportunity lies to improve the metrics.

The following are a few key steps to think about iterative model improvement for large scale end to end ML systems:

- **Identify the component**

  This accounts for finding the architectural component resulting in a high number of failures in our failure set. Suppose that in the case of the search ranking system, we have opted for a layered model approach that we will discuss in detail in our Search problem discussion.

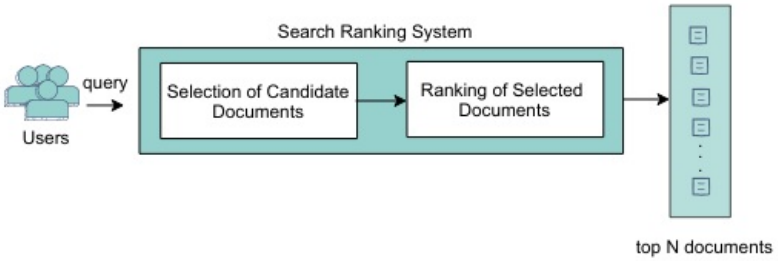
  In order to see the cause of failure, we will look at each layers' performance to understand the opportunity to significantly improve the quality of our search system. Let's assume that our search system has two key components 1) *Document selection* 2) *Ranking* of selected documents. Document selection focus is to ensure that all the top relevant documents get selected for the query while Ranker then ensures that our rank order is correct based on the relevance of the top 100 documents.

  If we look at few hundred failures of our overall search system, we should be able to identify the component that's resulting in more failures and as a result, decide to improve the quality of that component, e.g., if 80% of our overall search system failures are because of the ideal document not being selected in candidate selection component, we will debug the model deeply in that layer to see how to improve the quality of that model. Similarly, if ideal documents are mostly selected but are ranked lower by our ranking component, we will invest in finding the reason for failures in that layer and improve the quality of our ranking model.

- **Improve the quality of component**

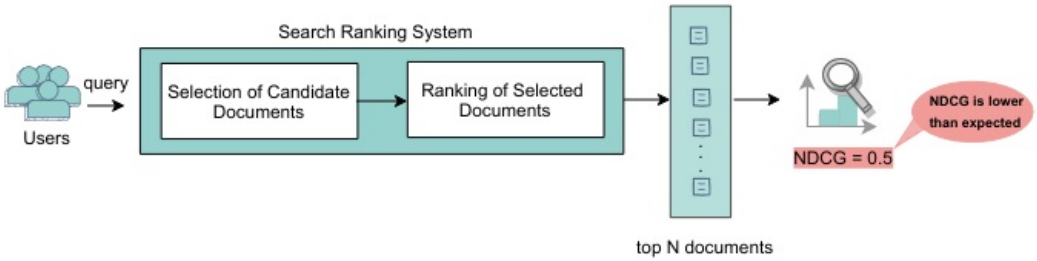
  Some of the model improvement methods that we have discussed above like adding more training data,

features, modeling approach in case of overfitting and underfitting will still be the same once we identify the component that needs work, e.g., if we identify that the candidate selection layer needs improvement in our search, we will try to see missing features, add more training data or play around with ML model parameters or try a new model.



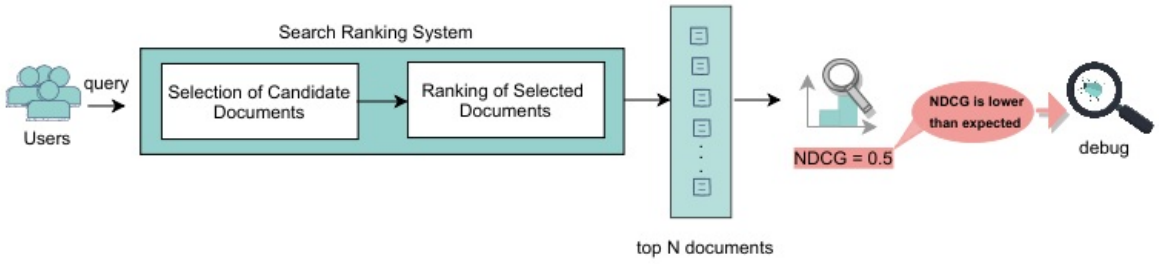
Search ranking system

1 of 10



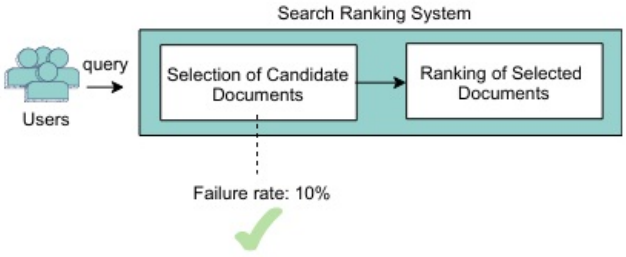
Evaluation of the system in terms of NDCG score

2 of 10



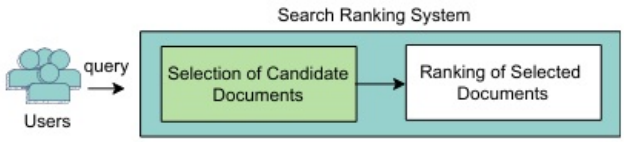
Evaluation of the system shows NDCG score is very low. So debugging is required.

3 of 10



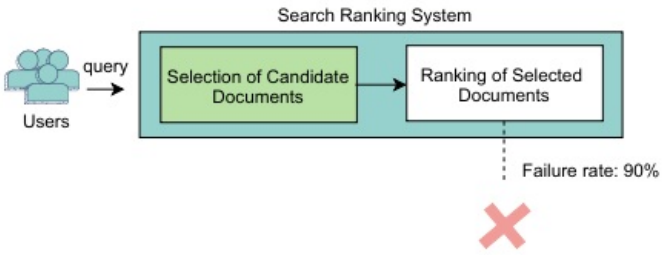
Check failure rate of Document Selection component

4 of 10



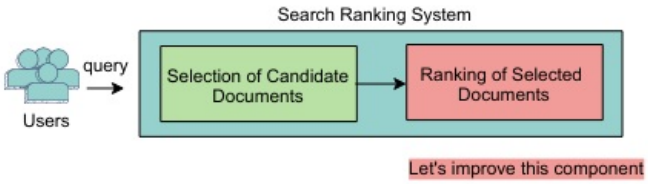
The selection component is working well

5 of 10



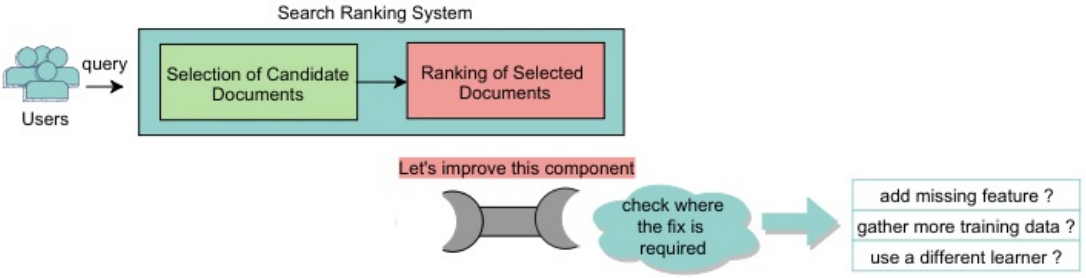
Failure rate is high for ranking component while debugging our failure set

6 of 10



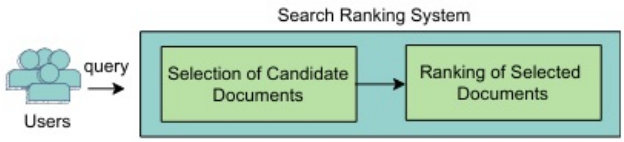
Improve ranking component ML model

7 of 10



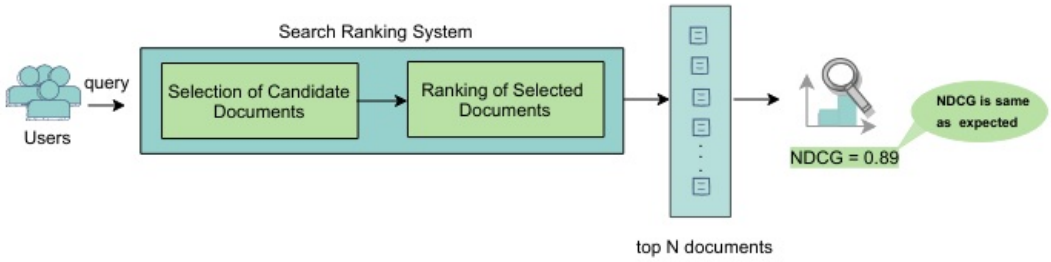
Improve model in ranking by model improvement strategies with features, training data or ML model

8 of 10



The ranking component is fixed now

9 of 10



NDCG of the system is improved after bug fixing

10 of 10

Back

Transfer Learning

Next

Problem Statement

Mark as Completed

Report an Issue

Ask a Question