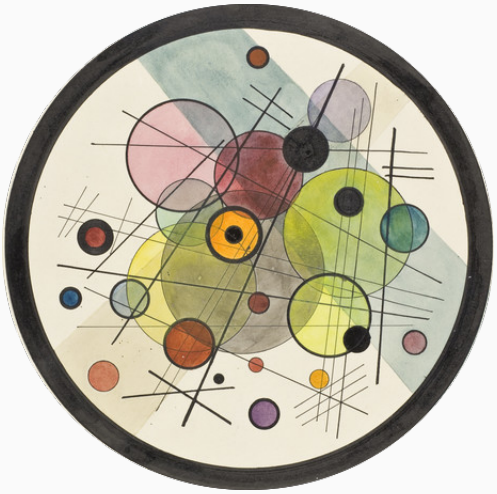


What are embeddings

Vicki Boykis

Abstract

Over the past decade, embeddings — numerical representations of non-tabular machine learning features used as input to deep learning models — have become a foundational data structure in industrial machine learning systems. TF-IDF, PCA, and one-hot encoding have always been key tools in machine learning systems as ways to compress and make sense of large amounts of textual data. However, traditional approaches were limited in the amount of context they could reason about with increasing amounts of data. As the volume, velocity, and variety of data captured by modern applications has exploded, creating approaches specifically tailored to scale has become increasingly important.

Google's Word2Vec paper made an important step in moving from simple statistical representations to semantic meaning of words. The subsequent rise of the Transformer architecture and transfer learning, as well as the latest surge in generative methods has enabled the growth of embeddings as a foundational machine learning data structure. This survey paper aims provide a deep dive into what embeddings are, their history, and usage patterns in industry.

Colophon

This paper is typeset with LaTeX. The cover art is Kandinsky's "Circles in a Circle" , 1923. ChatGPT was used to generate some of the figures.

Code, LaTeX, and Website

The latest version of the paper and code examples are available here. The website for this project is here.

About the Author

Vicki Boykis is a machine learning engineer. Her website is vickiboykis.com and her semantic search side project is viberary.pizza. Her email is vickiboykis@gmail.com.

Acknowledgements

I'm grateful to everyone who has graciously offered their technical feedback but especially to Nicola Barbieri, Peter Baumgartner, James Kirk, and Ravi Mody. All remaining errors, typos, and bad jokes are mine. Thank you to Dan for your patience, encouragement, for parenting while I was in the latent space, and for once casually asking, "How do you generate these 'embeddings', anyway?"

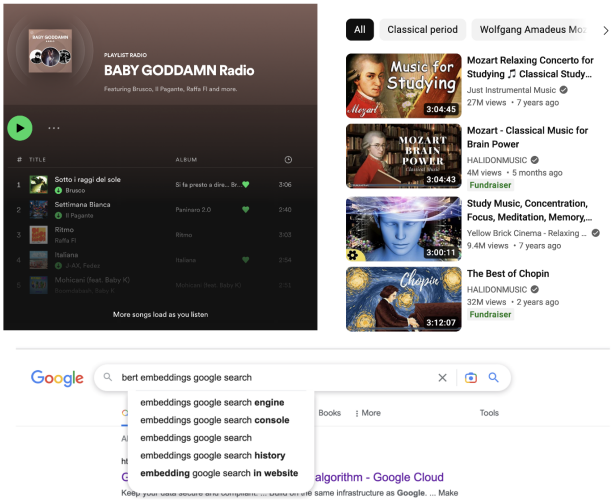
License

Contents

1 Introduction

Implementing deep learning models has become an increasingly important machine learning strategy[1] for companies looking to build data-driven products. In order to build and power deep learning models, companies collect and feed hundreds of millions of terabytes of multimodal[2] data into deep learning models. As a result, **embeddings** — deep learning models' internal representations of their input data — are quickly becoming a critical component of building machine learning systems.

For example, they make up a significant part of Spotify's item recommender systems [26], YouTube video recommendations of what to watch [11], and Pinterest's visual search [30]. Even if they are not explicitly presented to the user in the form of recommendations UIs, embeddings are also used internally at places like Netflix to make content decisions around which shows to develop based on user preference popularity.



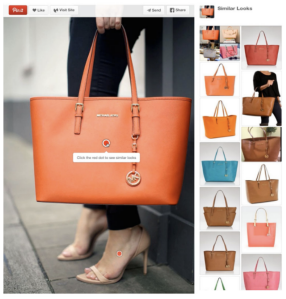


Figure 1: Similar Looks: We apply object detection to localize products such as bags and shoes. In this prototype, users click on objects of interest to view similar-looking products.

Figure 1: *Left to right: Products that use embeddings used to generate recommended items: Spotify Radio, YouTube Video recommendations, visual recommendations at Pinterest, BERT Embeddings in suggested Google search results*

The usage of embeddings to generate compressed, context-specific representations of content exploded in popularity after the publication of Google's Word2Vec paper [46].Building and expanding on the concepts in Word2Vec, the Transformer [64] architecture, with its self-attention mechanism, a much more specialized case of calculating context around a given word, has become the de-facto way to learn representations of growing multimodal vocabularies, and its rise in popularity both in academia and in industry has caused embeddings to become a staple of deep learning workflows.

However, the concept of embeddings can be elusive because they're neither data flow inputs or output results - they are intermediate elements that live

[1]Check out the machine learning industrial view Matt Turck puts together every year, which has exploded in size.

[2]Multimodal means a variety of data usually including text, video, audio, and more recently as shown in Meta's ImageBind, depth, thermal, and IMU.

within machine learning services to refine models. So it's helpful to define them explicitly from the beginning.

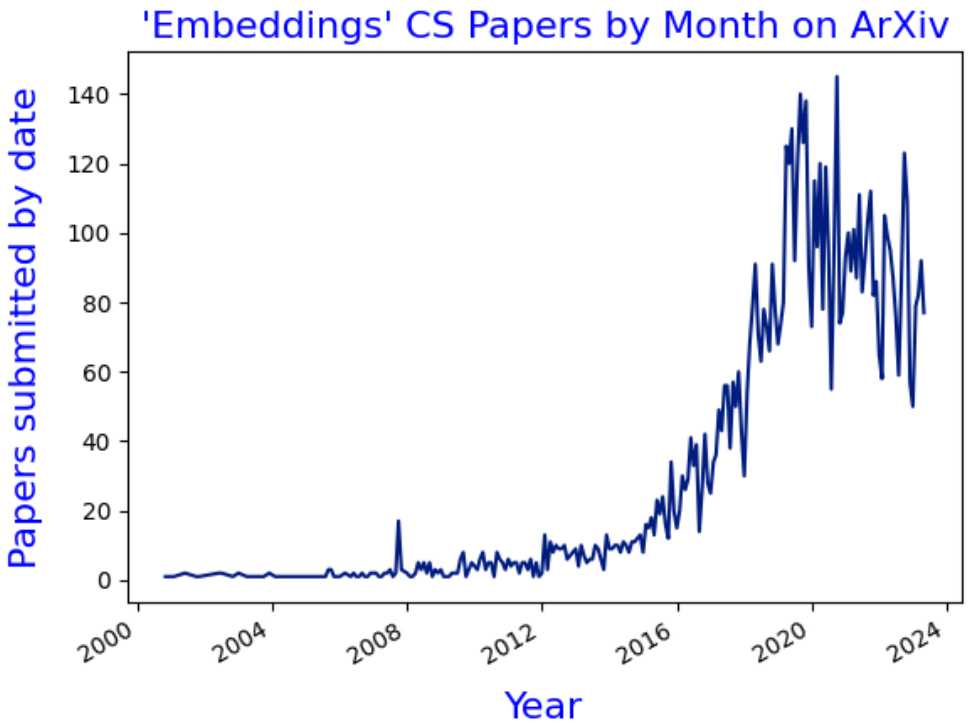


Figure 2: *Embeddings papers in Arxiv by month. source*

As a general definition, embeddings are data that has been transformed into matrices for use in deep learning computations. The process of embedding (as a verb):

- *Transforms* multimodal input into representations that are easier to perform intensive computation on, in the form of **vectors**, tensors, or graphs[50]. For the purpose of machine learning, we can think of vectors as a list (or array) of numbers.
- *Compresses* input information for use in a machine learning **task** — the type of methods available to us in machine learning to solve specific problems — such as summarizing a document or identifying tags or labels for social media posts or performing **semantic search** on a large text corpus. The process of compression changes variable feature dimensions into fixed inputs, allowing them to be passed efficiently into downstream components of machine learning systems.
- *Creates an embedding space* that is specific to the data the embeddings were trained on but that, in the case of deep learning representations, can also generalize to other tasks and domains through **transfer learning** — the ability to switch contexts — which is one of the reasons embeddings have exploded in popularity across machine learning applications

What do embeddings actually look like? Here is one single embedding,

also called a **vector**, in three **dimensions**. We can think of this as a representation of a single element in our dataset. For example, this hypothetical embedding represents a single word "fly", in three dimensions. Generally, we represent individual embeddings as row vectors.

$$\begin{bmatrix} 1 & 4 & 9 \end{bmatrix} \tag{1}$$

And here is a **tensor**, also known as a **matrix**[3], which is a multidimensional combination of vector representations of multiple elements. For example, this could be the representation of "fly", and "bird."

$$\begin{bmatrix} 1 & 4 & 9 \\ 4 & 5 & 6 \end{bmatrix} \tag{2}$$

These embeddings are the output of the process of **learning** embeddings, which we do by passing raw input data into a machine learning model. We transform that multidimensional input data by compressing it, through the algorithms we discuss in this paper, into a lower-dimensional space. The result is a set of vectors in an **embedding space.**

Figure 3: *The process of embedding.*

We often talk about item embeddings being in X dimensions, ranging anywhere from 100 to 1000, with diminishing returns in usefulness somewhere beyond 200-300 in the context of using them for machine learning problems[4]. This means that each item (image, song, word, etc) is represented by a vector of length X, where each value is a coordinate in an X-dimensional space.

We just made up an embedding for "bird", but let's take a look at what a real one for the word "hold" would look like in the quote, as generated by the BERT deep learning model,

> "Hold fast to dreams, for if dreams die, life is a broken-winged bird that cannot fly." — Langston Hughes

We've highlighted this quote because we'll be working with this sentence as our input example throughout this text.

[3]The difference between a matrix and a tensor is that it's a matrix if you're doing linear algebra and a tensor if you're an AI researcher.

[4]Embeddings are tunable as a hyperparameter but so far there have only been a few papers on optimal embedding size, with most of the size of embeddings set through magic and guesswork

```
import torch
from transformers import BertTokenizer, BertModel

# Load pre-trained model tokenizer (vocabulary)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

text = """Hold fast to dreams, for if dreams die, life is a broken-winged bird
    ↪  that cannot fly."""

# BERT token generation code truncated to show the final output, an embedding

[tensor([-3.0241e-01, -1.5066e+00, -9.6222e-01,  1.7986e-01, -2.7384e+00,
             -1.6749e-01,  7.4106e-01,  1.9655e+00,  4.9202e-01, -2.0871e+00,
             -5.8469e-01,  1.5016e+00,  8.2666e-01,  8.7033e-01,  8.5101e-01,
              5.5919e-01, -1.4336e+00,  2.4679e+00,  1.3920e+00, -3.9291e-01,
             -1.2054e+00,  1.4637e+00,  1.9681e+00,  3.6572e-01,  3.1503e+00,
             -4.4693e-01, -1.1637e+00,  2.8804e-01, -8.3749e-01,  1.5026e+00,
             -2.1318e+00,  1.9633e+00, -4.5096e-01, -1.8215e+00,  3.2744e+00,
              5.2591e-01,  1.0686e+00,  3.7893e-01, -1.0792e-01,  5.1342e-01,
             -1.0443e+00,  1.7513e+00,  1.3895e-01, -6.6757e-01, -4.8434e-01,
             -2.1621e+00, -1.5593e+01,  1.5249e+00,  1.6911e+00, -1.2916e+00,
              1.2339e+00, -3.6064e-01, -9.6036e-01,  1.3226e+00,  1.6427e+00,
              1.4588e+00, -1.8806e+00,  6.3620e-01,  1.1713e+00,  1.1050e+00,
              ↪  ...
             2.1277e+00])

```

Figure 4: *Analyzing Embeddings with BERT. See full notebook source*

We can see that this embedding is a PyTorch tensor object, a multidimensional matrix containing multiple levels of embeddings, and that's because in BERT's embedding representation, we have 13 different layers, each of which represents our given token, in this case word, one for each layer of our neural network. We can get the final embedding by pooling several layers, details we'll get into as we work our way up to understanding embeddings generated using BERT.

When we create an embedding for a word, sentence, or image that represents the artifact in the multidimensional space, we can do any number of things with this embedding. For example, for tasks that focus on content understanding in machine learning, we are often interested in comparing two given items to see how similar they are. Projecting text as a vector allows us to do so with mathematical rigor and compare words in a shared embedding space.

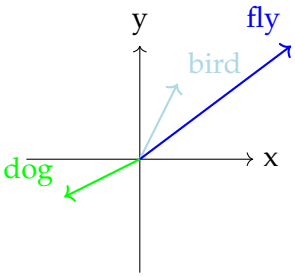


Figure 5: *Projecting words into a shared embedding space*

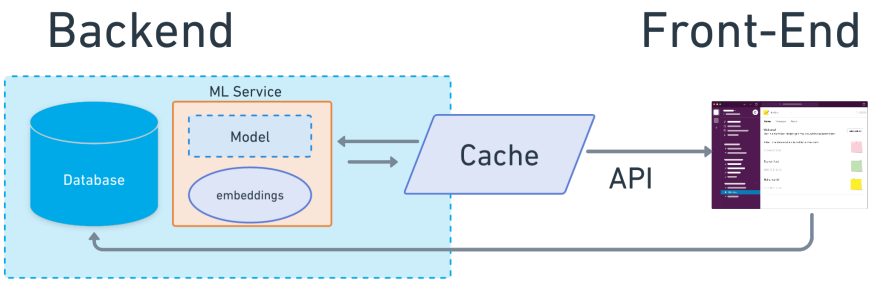


Figure 6: *Embeddings in the context of an application.*

Engineering systems based on embeddings can be computationally expensive to build and maintain[59]. The need to create, store, and manage embeddings has also recently resulted in the explosion of an entire ecosystem of related products. For example, the recent rise in the development of vector databases to facilitate production-ready use of nearest neighbors semantic queries in machine learning systems[5], and the rise of embeddings as a service[6].

As such, it's important to understand their context both as end-consumers, product management teams, and as developers who work with them. But in my deep-dive into the embeddings reference material, I found that there are two types of resources: very deeply technical academic papers, for people who are already NLP experts, and surface-level marketing spam blurbs for people looking to buy embeddings-based tech, and that neither of these overlap in what they cover.

In Systems Thinking, Donella Meadows writes, "You think that because you understand 'one' that you must therefore understand 'two' because one and one make two. But you forget that you must also understand 'and.'"[44] In order to understand the current state of embedding architectures and be able to decide how to build them, we must understand how they came to be. In building my own understanding, I wanted a resource that was technical enough to be useful enough to ML practitioners, but one that also put embeddings in their correct business and engineering contexts as they become more often used in ML architecture stacks. This is, hopefully, that text.

In this text, we'll examine embeddings from three perspectives, working our way from the highest level view to the most technical. We'll start with the business context, followed by the engineering implementation, and finally

[5]For a survey of the vector database space today, refer to this article

[6]Embeddings now are a key differentiator in pricing between on-demand ML services

look at the machine learning theory, focusing on the nuts and bolts of how they work. On a parallel axis, we'll also travel through time, surveying the earliest approaches and moving towards modern embedding approaches.

In writing this text, I strove to balance the need to have precise technical and mathematical definitions for concepts and my desire to stay away from explanations that make people's eyes glaze over. I've defined all technical jargon when it appears for the first time to build context. I include code as a frame of reference for practitioners but don't go as deep as a code tutorial would[7]. So, it would be helpful for the reader to have some familiarity with programming and machine learning basics, particularly after the sections that discuss business context. But, ultimately the goal is to educate anyone who is willing to sit through this, regardless of level of technical understanding.

It's worth also mentioning what this text does not try to be: it does not try to explain the latest advancements in GPT and generative models, it does not try to explain transformers in their entirety, and it does not try to cover all of the exploding field of vector databases and semantic search. I've tried my best to keep it simple and focus on really understanding the core concept of embeddings.

2 Recommendation as a business problem

Let's step back and look at the larger context with a concrete example before diving into implementation details. Let's build a social media network, **Flutter**, the premier social network for all things with wings. Flutter is a web and mobile app where birds can post short snippets of text, videos, images, and sounds, to let other birds, insects and bats in the area know what's up. Its business model is based on targeted advertising, and its app architecture includes a "home" feed based on birds that you follow, made up of small pieces of multimedia content called **"flits"**, which can be either text, videos, or photos. The home feed itself is by default in reverse chronological order that is curated by the user. But we also would like to offer personalized, recommended flits so that the user finds interesting content on our platform that they might have not known about before.

[7] In other words, I wanted to straddle the "explanation" and "reference" quadrants of the Diátaxis framework

Figure 7: *Flutter's content timeline in a social feed with a blend of organic followed content, advertising, and recommendations.*

How do we solve the problem of what to show in the timeline here so that our users find the content relevant and interesting, and balance the needs of our advertisers and business partners?

In many cases, we can approach engineering solutions without involving machine learning. In fact, we should definitely start without it[72] because machine learning adds a tremendous amount of complexity to our working application[55]. In the case of the Flutter home feed, though, machine learning forms a business-critical function part of the product offering. From the business product perspective, the objective is to offer Flutter's users content that is relevant[8], interesting, and novel so they continue to use the platform. If we do not build discovery and personalization into our content-centric product, Flutter users will not be able to discover more content to consume and will disengage from the platform.

This is the case for many content-based businesses, all of which have feed-like surface areas for recommendations, including Netflix, Pinterest, Spotify, and Reddit. It also covers e-commerce platforms, which must surface relevant items to the user, and information retrieval platforms like search engines, which must provide relevant answers to users upon keyword queries. There is a new category of hybrid applications involving question-and-answering in semantic search contexts that is arising as a result of work around the GPT

[8]The specific definition of a relevant item in the recommendations space varies and is under intense academic and industry debate, but generally it means an item that is of interest to the user

series of models, but for the sake of simplicity, and because that landscape changes every week, we'll stick to understanding the fundamental underlying concepts.

In subscription-based platforms[9], there is clear business objective that's tied directly to the bottom line, as outlined in this 2015 paper [62] about Netflix's recsys:

> The main task of our recommender system at Netflix is to help our members discover content that they will watch and enjoy to maximize their long-term satisfaction. This is a challenging problem for many reasons, including that every person is unique, has a multitude of interests that can vary in different contexts, and needs a recommender system most when they are not sure what they want to watch. Doing this well means that each member gets a unique experience that allows them to get the most out of Netflix. As a monthly subscription service, member satisfaction is tightly coupled to a person's likelihood to retain with our service, which directly impacts our revenue.

Knowing this business context, how and why might we use embeddings in machine learning workflows in Flutter to show users flits that are interesting to them personally, knowing that personalized content is more relevant and generally gets higher rates of engagement?[29] than non-personalized forms of recommendation on online platforms [10] We need to first understand how web apps work and where embeddings fit into them.

2.1 Building a web app

Most of the apps we use today — Spotify, Gmail, Reddit, Slack, and Flutter — are all designed based on the same foundational software engineering patterns. They are all apps available on web and mobile clients. They all have a front-end where the user interacts with the various **product features** of the applications, an API that connects the front-end to back-end elements, and a database that processes data and remembers state.

> As an important note, **features** have many different definitions in machine learning and engineering. In this specific case, we mean collections of code that make up some front-end element, such as a button or a panel of recommendations. We'll refer to these as **product features**, in contrast with **machine learning features**, which are input data into machine learning models.

[9] In ad-based services, the line between retention and revenue is a bit murkier, and we have often what's known as a multi-stakeholder problem, where the actual optimized function is a balance between meeting the needs of the user and meeting the needs of the advertiser[71]. In real life, this can often result in a process of enshittifcation[15] of the platform that leads to extremely suboptimal end-user experiences. So, when we create Flutter, we have to be very careful to balance these concerns, and we'll also assume for the sake of simplification that Flutter is a Good service that loves us as users and wants us to be happy.

[10] For more, see this case study on personalized recommendations as well as the the intro section of this paper which covers many personalization use-cases

This application architecture is commonly known as **model-view-controller** pattern [19], or in common industry lingo, a **CRUD** app, named for the basic operations that its API allows to manage application state: create, read, update, and delete.

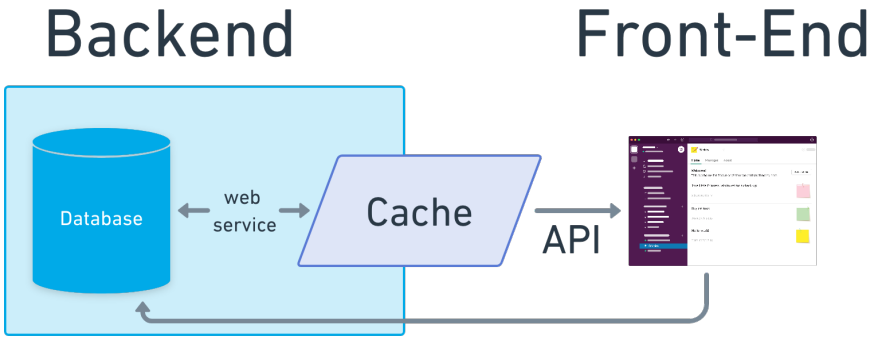


Figure 8: *Typical CRUD web app architecture*

When we think of structural components in the architectures of these applications, we might think first in terms of product features. In an application like Slack, for example, we have the ability to post and read messages, manage notifications, and add custom emojis. Each of these can be seen as an application feature. In order to create features, we have to combine common elements like databases, caches, and web services. All of this happens as the client talks to the API, which talks to the database to process data. At a more granular, program-specific level, we might think of foundational data structures like arrays or hash maps, and lower still, we might think about memory management and network calls. These are all foundational elements of modern programming.

At the feature level, though, we see that it not only includes the typical CRUD operations, such as the ability to post and read Slack messages, but also elements that are more than operations that alter database state. Some features such as personalized channel suggestions, returning relevant results through search queries, and predicting Slack connection invites necessitates the use of machine learning.

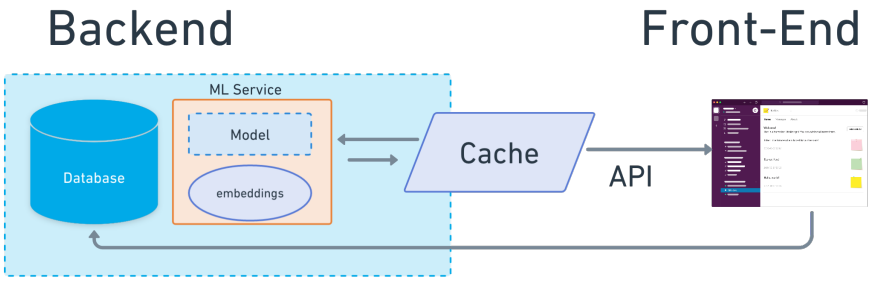


Figure 9: *CRUD App with Machine learning service*

2.2 Differences between rules and machine learning

Machine learning systems live within the backend of web applications. They are typically integrated into production workflows, but they process data much differently, and use the application's data to generate machine learning models that act upon and feed data back into the application, to the user. For more on the specifics of how to think about these data-centric engineering systems, see Kleppmann[34].

To understand where embeddings fit here, it first makes sense to understand how machine learning works at Flutter, or any given company, as a whole. In a typical consumer company, the user-facing app is made up of product features written in code. To add a new web app feature, we add business logic, which acts on data in the app to develop our new feature and add that back into the application. When we think of the typical data-centric software development lifecycle, it looks like this. We start with the business logic. For example, let's take the ability to post messages. We'd like users to be able to input text and emojis in their language of choice, have the messages sorted chronologically, and render correctly on web and mobile. These are the business requirements. We use the input data, in this case, user messages, and format them correctly and sort chronologically, at low latency, in the UI.

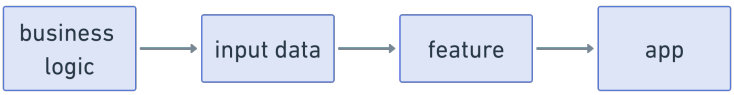


Figure 10: *A typical application development lifecycle*

For applications powered by machine learning, though, we need to start not with code that shows text in the app, but input data that we use to build a model that will suggest pieces of content, such as personalized channel suggestions for example. This requires thinking about application development slightly differently, and when we write an application that includes machine learning models as input, however, we're inverting the traditional app lifecycle. What we have instead, is data + our desired outcome. The data is combined into a model, and it is this model which instead generates our business logic that builds features.

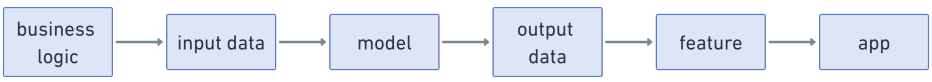


Figure 11: *ML Development lifecycle*

In short, the difference between programming and machine learning development is that we are not generating answers through actions, but rules through data. These rules are then re-incorporated into the application to generate more answers.

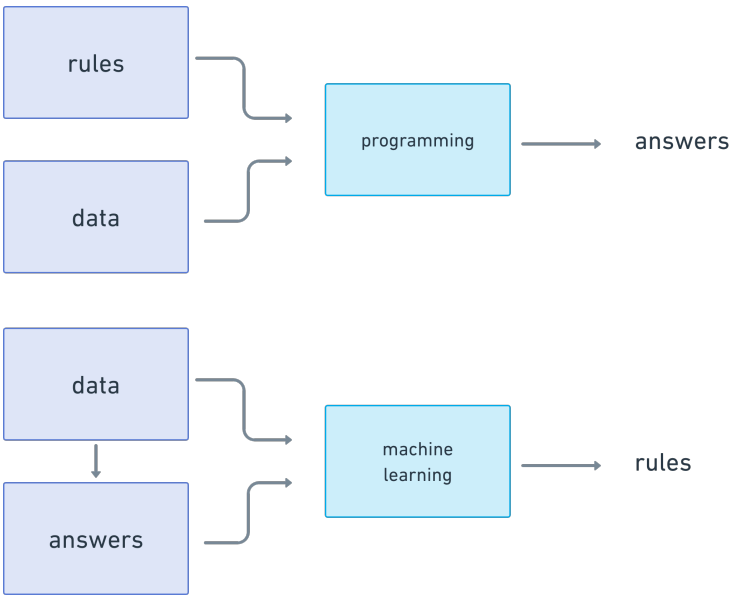


Figure 12: *Generating answers via machine learning. The top chart shows a classical programming approach with rules and data as inputs, while the bottom chart shows a machine learning approach with data and answers as inputs. [8]*

2.3 Building a web app with machine learning

All machine learning systems can be examined through how they accomplish these four steps. When we build models, our key questions should be, "what kind of input do we have and how is it formatted", and "what do we get as a result." We'll be asking this for each of the approaches we look at. When we build a machine learning system, we start by processing data and finish by serving a learned model artifact.

The four components of a machine learning system are[11]:

- **Input data** - processing data from a database or streaming from a production application for use in modeling
- **Feature Engineering and Selection** - The process of examining the data and cleaning it to pick features[12] for machine learning. This piece always takes the longest in any given machine learning system, and is also known as finding **representations**[4] of the data that best fit the machine learning algorithm. This is where, in the new model architectures, we use embeddings as input.
- **Model Building** - We select the features that are important and train our model, iterating on different performance metrics over and over again until we have an acceptable model we can use. Embeddings are also the output of this step that we can use in other, downstream steps.
- **Model Serving** - Now that we have a model we like, we serve it to production, where it hits a web service, potentially cache, and our

[11] There are infinitely many layers of horror in ML systems[36]. These are still the foundational components.

API where it then propagates to the front-end for the user to consume as part of our web app

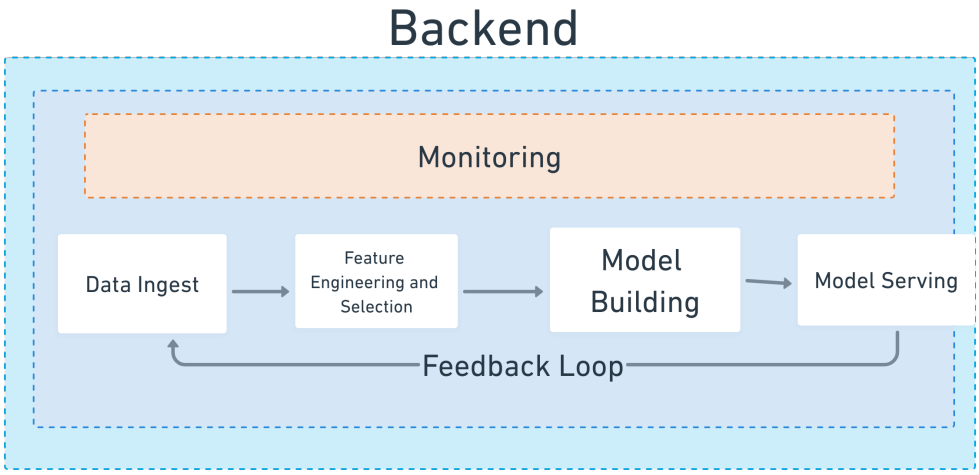


Figure 13: *CRUD app with ML*

Within machine learning, there are many approaches we can use to fit different tasks. Machine learning workflows that are most effective are formulated as solutions to both a specific business need and a machine learning **task**. Tasks can best be thought of as approaches to modeling within the categorized solution space. For example, learning a regression model is a specific case of a task. Others include clustering, machine translation, anomaly detection, similarity matching, or semantic search. The three highest-level types of ML tasks are **supervised**, where we have training data that can tell us whether the results the model predicted are correct according to some model of the world. The second is **unsupervised**, where there is not a single ground-truth answer. An example here is clustering of our customer base. A clustering model can detect patterns in your data but won't explicitly label what those patterns are. The third is **reinforcement learning** which is separate from these two categories and formulated as a game theory problem: we have an agent moving through an environment and we'd like to understand how to optimally move them through a given environment using explore-exploit techniques. We'll focus on supervised learning, with a look at unsupervised learning with PCA and Word2Vec.

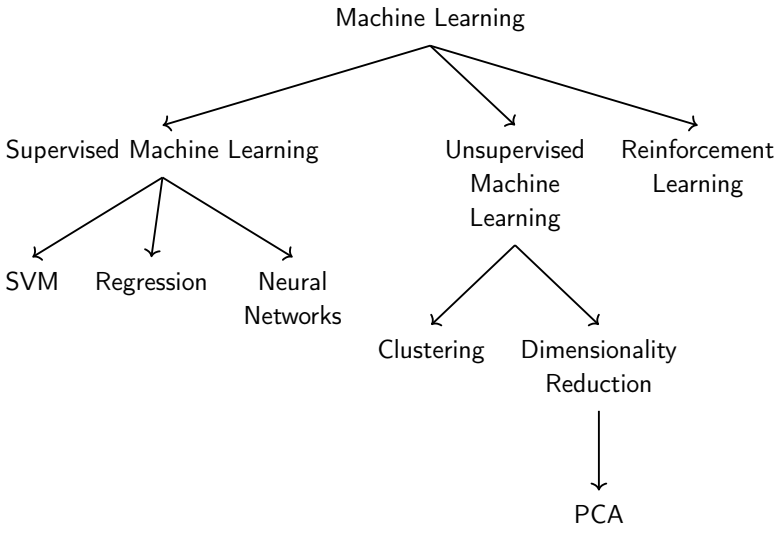


Figure 14: *Machine learning task solution space and model families*

2.4 Formulating a machine learning problem

To use embeddings effectively, we need to understand where they fit in a machine learning task. As we saw in the last section, machine learning is a process that takes data as input to produce rules for how we should classify something or filter it or recommend it, depending on the task at hand. In any of these cases, for example, to generate a set of potential candidates, we need to construct a **model**. The model is a set of instructions for how to use the input data we feed it and generate a set of results. For Flutter, an example of a model we'd like to build is a **candidate generator** that picks flits similar to flits our birds have already liked.

The machine learning model, in our case, the candidate generator, is a function, $f(X) -> y$. We **train** — or build — our model by initializing it with some set of weights as learnable input parameters that takes a set of inputs, X, and gives us an output Y. . When we say learnable, we mean that we can learn what the correct inputs into a model are through some kind of iterative process where we feed the model data and see if it improves.

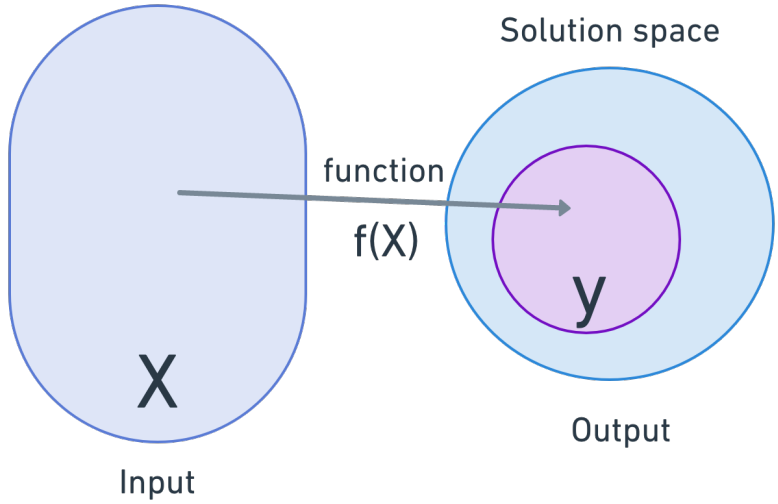


Figure 15: *How inputs map to outputs in ML functions [33]*

How do we know our model is any good? We initialize it with some set of values, weights, and we iterate on those weights, usually by minimizing a **cost function**. The cost function is a function that models the difference between our model's predicted value and the real value for that given model. The first output may not be the most optimal, so we iterate over the model space many times, optimizing for a specific metric that will make the model as representative of reality as possible and minimize the difference between the actual and predicted values.

Within the model, in order to evaluate the results, we have to formulate our input process and modeling methods in a specific way.

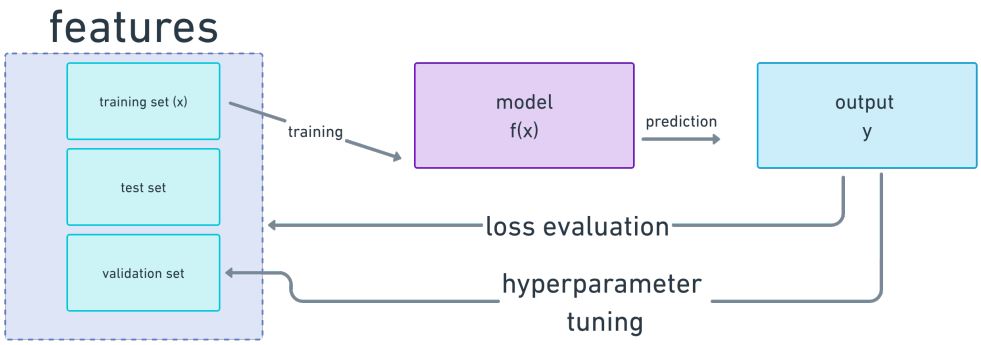


Figure 16: *The cycle of machine learning model development*

For traditional **supervised** modeling approaches using tabular data, we start with our input data, or a **corpus** as it's generally known in machine learning problems that deal with text in the field known as **NLP** (natural language processing). We then shape this data to select the pieces of it we'd like to use for modeling. These are our **features**. We take two parts of this data as holdout data that we don't feed into the model. The first part, the **test set**, we use to validate the final model on data it's never seen before. We use the second split, called the **validation set**, to check our **hyperparameters** during the model training phase.

2.4.1 The Task of Recommendations

When our business question is, "What content should we show our users," we are facing the machine learning task for recommendation. Recommender systems are systems set up for **information retrieval**, a field closely related to NLP that's focused on finding relevant information in large collections of documents. The goal of information retrieval is to synthesize large collections of unstructured text documents. Within information retrieval, there are two complementary industry solutions in how we can offer users the correct content in our app: search, and recommendations.

Search is the problem of directed[17] information seeking, i.e. the user offers the system a specific query and would like a set of refined results. Search engines at this point are a well-established traditional solution in the space. **Recommendation** is a problem where "man is the query."[56] Here, we don't know what the person is looking for exactly, but we would like to infer what they like, and recommend items based on their learned tastes and preferences.

The first industrial recommender systems were created to filter messages in email and newsgroups[21] at the Xerox Palo Alto Research Center based on a growing need to filter incoming information from the web. The most common recommender systems today are those at Netflix, YouTube, and other large-scale platforms that need a way to surface relevant content to users.

The goal of recommender systems is to synthesize and surface items that are relevant to the user using statistical approaches and machine learning. Within the framework of machine learning approaches for recommendation, the main machine learning task is to determine which items to show to a user in a given situation. [5].

There are several common ways to approach the recommendation problem.

- **Collaborative filtering** - The most common approach looks at user activity as recorded in log data (product ratings, clicks on content) in relation to the available items to be rated in order to compute the given utility of an item for a user. There are **neighborhood models**, where ratings are predicted initially by finding users similar to our given target user and we use similarity functions to compute the closeness of users.
  Another common approach is using **model-based** methods such **matrix factorization**, the process of representing users and items in a feature matrix made up of low-dimensional factor vectors, which in our case, are also known as embeddings, and learning those feature vectors through the process of minimizing a cost function. This process can be thought of as similar to Word2Vec[42], a deep learning model which we'll discuss in depth in this document. There are many different approaches to collaborative filtering, including matrix factorization and **factorization machines**.

- **Content filtering** - This approach uses metadata available about our items (for example in movies or music, the title, year released, genre, and so on) as initial or additional features input into models and work well when we don't have much information about user activity, although they are often used in combination with collaborative filtering approaches. Many embeddings architectures fall into this category since they help us model the features for our items.
- **Learn to Rank** - Usually machine learning problems like matrix factorization are formulated by minimizing the error of a cost function. Learn to rank methods focus on ranking items in relation to each other based on a known set of preferred rankings and the error is, instead, cases when pairs or lists of items are ranked incorrectly. Here,the problem is not presenting a single item but a set of items and how they interplay. This step would normally take place after candidate generation, in a filtering step because it's computationally expensive to rank extremely large lists.
- **Neural Recommendations** - The process of using neural networks to capture the same relationships that matrix factorization does without explicitly having to create a user/item matrix and based on the shape of the input data. This is where deep learning networks, and recently, large language models, come into play. Examples of deep learning architectures used for recommendation include Word2Vec and BERT, which we'll cover in this document, and convolutional and recurrent neural networks for sequential recommendation. The big benefit deep learning brings to the table is the ability to better model content-based recommendations and give us representations of our items in the embedding space.[70]

Recommender systems have evolved their own unique architectures[13], and they usually include constructing a four-stage recommender system that's made up of several machine learning models, each of which perform a different machine learning task.

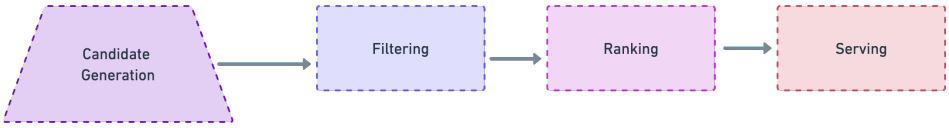


Figure 17: *Recommender systems as a machine learning problem*

- **Candidate Generation** - First, we ingest data from the web app. This data goes into the initial piece, which hosts our first-pass model generating **candidate recommendations**. This is where collaborative filtering takes place, and we whittle our list of potential candidates down from millions to thousands or hundreds.

[13]For a good survey on the similarities and difference between search and recommendations, read this great post on system design

- **Filtering** - Once we have a generated list of candidates, we want to continue to filter them, using business logic (i.e. we don't want to see NSFW content, or items that are not on sale, for example.). This is generally a heavily heuristic-based step.
- **Ranking** - Finally, we need a way to order the filtered list of recommendations based on what we think the user will prefer the most, so the next stage is **ranking**, and then we serve them out in the timeline or the ML product interface we're working with.
- **Retrieval** - This is the piece where the web application usually hits a model endpoint to get the final list of items served to the user through the product UI.

Databases have become the fundamental tool in building backend infrastructure that performs data lookups. Embeddings have become similar building blocks in the creation of many modern search and recommendation product architectures.Embeddings are a type of **machine learning feature** that we use first as input into the feature engineering stage, and the first set of results that come from our candidate generation stage, that are then incorporated into downstream processing steps of ranking and retrieval to produce the final items the user sees.

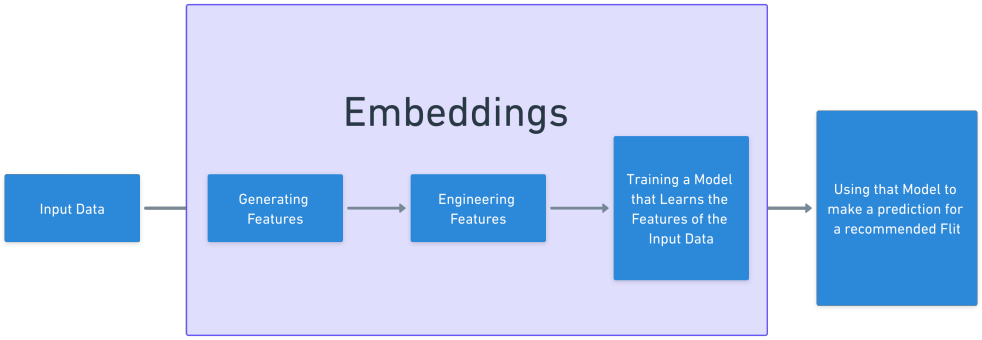


Figure 18: *Embeddings in the context of building an ML candidate generator model*

2.4.2 Machine learning features

Now that we have a high-level conceptual view of recommender systems, let's build towards the candidate generation model that will offer relevant flits.

Let's start by modeling a traditional machine learning problem and contrast it with our NLP problem. For example, let's say that one of our business problems is predicting whether a bird is likely to continue to stay on Flutter or to churn[14] — disengage and leave the platform.

When we predict churn, we have a given set of machine learning feature inputs for each user and a final binary output of 1 or 0 from the model, 1 if the bird is likely to churn, or 0 if the user is likely to stay on the platform.

We might have the following inputs:

- How many posts the bird has clicked through in the past month (we'll call this `bird_posts` in our input data)

[14] An extremely common business problem to solve in almost every industry where either customer population or subscription based on revenues is important

- The geographical location of the bird from the browser headers (`bird_geo`)
- How many posts the bird has liked over the past month (`bird_likes`)

Table 1: *Tabular Input Data for Flutter Users*

bird_id	bird_posts	bird_geo	bird_likes
012	2	US	5
013	0	UK	4
056	57	NZ	70
612	0	UK	120

We start by selecting our model features and arranging them in tabular format. We can formulate this data as a table (which, if we look closely, is also a matrix) based on rows of the bird id and our bird features.

Tabular data is any structured data. For example, for a given Flutter user we have their user id, how many posts they've liked, how old the account is, and so on. This approach works well for what we consider traditional machine learning approaches which deal with tabular data. As a general rule, the creation of the correct formulation of input data is perhaps the heart of machine learning. I.e. if we have bad input, we will get bad output. So in all cases, we want to spend our time putting together our input dataset and engineering features very carefully.

These are all discrete features that we can feed into our model and learn weights from, and is fairly easy as long as we have numerical features. But, something important to note here is that, in our bird interaction data, we have both numerical and textual features (bird geography). So what do we do with these textual features? How do we compare "US" to "UK"?

The process of formatting data correctly to feed into a model is called **feature engineering**. When we have a single continuous, numerical feature, like "the age of the flit in days", it's easy to feed these features into a model. But, when we have textual data, we need to turn it into numerical representations so that we can compare these representations.

2.5 Numerical Feature Vectors

Within the context of working with text in machine learning, we represent features as numerical vectors. We can think of each row in our tabular feature data as a vector. And a collection of features, or our tabular representation, is a matrix. For example, in the vector for our first user, $[012, 2, 'US', 5]$, we can see that this particular value is represented by four features. When we create vectors, we can run algorithmic computations over them and use them as inputs into ML models in the numerical form we require.

Mathematically, vectors are collections of coordinates that tell us where a given point is in space among many dimensions. For example, in two dimensions, we have a point $[2, 5]$, representing `bird_posts` and `bird_likes`.

In three dimensions, with three features including the bird id, we would

have a matrix

$$\begin{bmatrix} 0 & 0 & 12 \\ 0 & 2 & 0 \\ 0 & 0 & 5 \end{bmatrix} \tag{3}$$

which tells us where that user falls on all three axes.

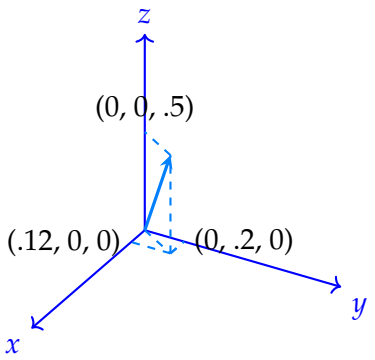


Figure 19: *Projecting a vector into the 3d space, scaled to 1*

But how do we represent "US" or "UK" in this space? Because modern models work converge by performing operations on matrices [38], we need to encode geography as some sort of numerical value so that the model can calculate them as inputs[15]. So, once we have a combination of vectors, we can compare it to other points. So in our case, each row of data tells us where to position each bird in relation to any other given bird based on the combination of features. And that's really what our numerical features allow us to do.

2.6 From Words to Vectors in Three Easy Pieces

In "Operating Systems: Three Easy Pieces", the authors write, "Like any system built by humans, good ideas accumulated in operating systems over time, as engineers learned what was important in their design."[3] The same is true for today's large language models. They were built on hundreds of foundational ideas over the course of decades. There are, similarly, several fundamental concepts that make up the work of transforming words to numerical representations.

These show up over and over again, in every deep learning architecture and every NLP-related task[16]:

- **Encoding** - We need to represent our non-numerical, multimodal data as numbers so we can create models out of them. There are many different ways of doing this.
- **Vectors** - we need a way to store the data we have encoded and have the ability to perform mathematical functions in an optimized

[15]There are some models, specifically decision trees, where you don't need to do text encoding because the tree learns the categorical variables out of the box, however implementations differ, for example the two most popular implementations, scikit-learn and XGBoost[1], can't.

[16]When we talk about tasks in NLP-based machine learning, we mean very specifically, what the machine learning problem is formulated to do. For example, we have the task of ranking, recommendation, translation, text summarization, and so on.

way on them. We store encodings as vectors, usually floating-point representations.

- **Lookup matrices** - Often times, the end-result we are looking for from encoding and embedding approaches is to give some approximation about the shape and format of our text, and we need to be able to quickly go from numerical to word representations across large chunks of text. So we use lookup tables, also known as hash tables, also known as attention, to help us map between the words and the numbers.

As we go through the historical context of embeddings, we'll build our intuition from encoding to BERT and beyond[17]. What we'll find as we go further into the document is that the explanations for each concept get successively shorter, because we've already done the hard work of understanding the building blocks at the beginning.

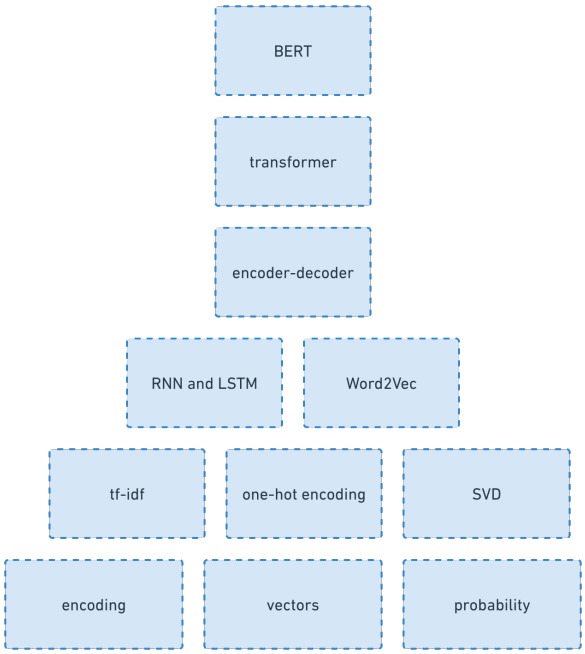


Figure 20: *Pyramid of fundamental concepts building to BERT*

3 Historical Encoding Approaches

3.1 History of Embeddings

Compressing content into lower dimensions for compact numerical representations and calculations is not a new idea in machine learning. For as long as humans have been overwhelmed by information, we've been trying to synthesize it so that we can make decisions based on it. In the context of information retrieval and machine learning, early traditional approaches have included one-hot encoding, TF-IDF, bag-of-words, LSA, and LDA.

[17] Original diagram from this excellent guide on BERT

The earlier approaches were **count-based** methods. They focused on counting how many times a word appeared relative to other words and generating encodings based on that. LDA and LSA can be considered statistical approaches, but they are still concerned with inferring the properties of a dataset through heuristics rather than modeling. **Prediction-based** approaches came later and instead learned the properties of a given text through models such as support vector machines, Word2Vec, BERT, and the GPT series of models, all of which use learned embeddings instead.

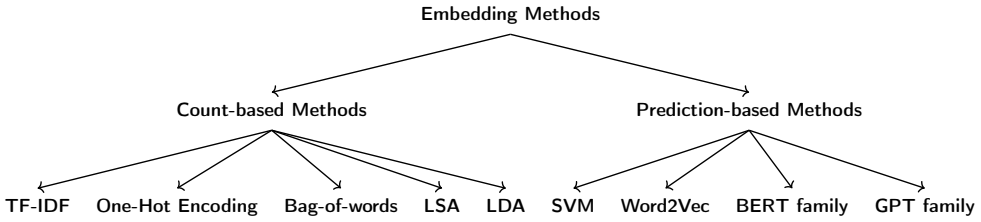


Figure 21: *Embedding Method Solution Space*

A Note on the Code In looking at these approaches programmatically, we'll start by using `scikit-learn`, the de-facto standard machine learning library for smaller datasets, with some implementations in native Python for clarity in understanding functionality that scikit wraps. As we move into deep learning, we'll move to PyTorch, a deep learning library that's quickly becoming industry-standard for deep learning implementation. There are many different ways of implementing the concepts we discuss here, these are just the easiest to illustrate using Python's ML lingua franca libraries.

3.2 Early Approaches

The first approaches to generating textual features were count-based, relying on simple counts or high-level understanding of statistical properties: they were **descriptive** instead of models, which are **predictive** and attempt to guess a value based on a set of input values. The first methods were **encoding methods**, a precursor to embedding. Encoding is often a process that still happens as the first stage of data preparation for input into more complex modeling approaches. There are several methods to create text features using a process known as encoding so that we can map the geography feature into the vector space:

- Ordinal encoding
- Dummy encoding
- One-Hot encoding

In all these cases, what we are doing is creating a new feature that maps to the text feature column but is a numerical representation of the variable so that we can project it into that space for modeling purposes. We'll motivate these examples with simple code snippets from sklearn, the most common

library for demonstrating basic ML concepts. We'll start with **count-based** approaches.

3.3 Encoding

Ordinal encoding Let's again come back to our dataset of flits. We encode our data using sequential numbers. For example, "1" is "finch", "2" is "bluejay" and so on. We can use this method only if the variables have a natural ordered relationship to each other. For example, in this case "bluejay" is not "more" than "finch" and so would be incorrectly represented in our model. The case is the same, if, in our flit data, we encode "US" as 1 and "UK" as 2.

Table 2: *Bird Geographical Location Encoding*

bird_id	bird_posts	bird_geo	bird_likes	enc_bird_geo
012	2	US	5	2
013	0	UK	4	1
056	57	NZ	70	0
612	0	UK	120	1

```
from sklearn.preprocessing import OrdinalEncoder

data = [['US'], ['UK'], ['NZ']]
print(data)
[['US']
 ['UK']
 ['NZ']]

# our label features
encoder = OrdinalEncoder()
result = encoder.fit_transform(data)
print(result)
[[2.]
 [1.]
 [0.]]
```

Figure 22: *Ordinal Encoding in Scikit-Learn source*

3.3.1 Dummy and one-hot encoding

Dummy encoding is, given n categories (i.e. "US", "UK", and "NZ") encodes the variables into $n - 1$ categories, creating a new feature for each category. So, if we have three variables, dummy encoding encodes into two dummy variables. Why would we do this? If the categories are mutually exclusive, as they usually are in point-in-time geolocation estimates, if someone is in the US, we know for sure they're not in the UK and not in NZ, so it reduces computational overhead.

If we instead use all the variables and they are very closely correlated, there is a chance we'll fall into something known as the **dummy variable trap**.

We can predict one variable from the others, which means we no longer have feature independence. This generally isn't a risk for geolocation since there are more than 2 or 3 and if you're not in the US, it's not guaranteed that you're in the UK. So, if we have US = 1, UK = 2, and NZ = 3, and prefer more compact representations, we can use dummy encoding.

However, many modern ML approaches don't require linear feature independence and use L1 regularization[18] to prune feature inputs that don't minimize the error, and as such only use one-hot encoding.

One-hot encoding is the most commonly-used of the count-based methods. This process creates a new variable for each feature that we have. Everywhere the element is present in the sentence, we place a "1" in the vector. We are creating a mapping of all the elements in the feature space, where 0 indicates a non-match and 1 indicates a match, and comparing how similar those vectors are.

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

enc = OneHotEncoder(handle_unknown='ignore')
data = np.asarray([['US'], ['UK'], ['NZ']])
enc.fit(data)
enc.categories_
# Result: [array(['NZ', 'UK', 'US'], dtype='<U2')]
onehotlabels = enc.transform(data).toarray()
onehotlabels
# Result:
array([[0., 0., 1.],
           [0., 1., 0.],
           [1., 0., 0.]])
```

Figure 23: *One-Hot Encoding in Scikit-learnsource*

Table 3: *Our one-hot encoded data with labels*

bird_id	US	UK	NZ
012	1	0	0
013	0	1	0
056	0	0	1

Now that we've encoded our textual features as numbers, we can feed them into the model we're developing to predict churn. The function we've been learning will minimize the loss of the model, or the distance between the model's prediction and the actual value, by predicting correct parameters for each of these features. The learned model will then return a value from 1 to 0 that is a probability that the event, either churn or no-churn, has taken

[18]Regularization is a way to prevent our model from overfitting. Overfitting means our model can exactly predict outcomes based on the training data, but it can't learn new inputs that we show it, which means it can't generalize

place, given the input features of our particular bird. Since this is a supervised model, we then evaluate this model for accuracy by feeding our test data into the model and comparing the model's prediction against the actual data, which tells us whether the bird has churned or not.

What we've built is a standard **logistic regression model**. Generally these days the machine learning community has converged on using gradient-boosted decision tree methods for dealing with tabular data, but we'll see that neural networks build on simple linear and logistic regression models to generate their output, so it's a good starting point.

Embeddings as feature inputs

Once we have encoded our feature data, we can use this input for any type of model that accepts tabular features. In our machine learning task, we were looking for output that indicated whether a bird was likely to leave the platform based on their location and some usage data. Now, we'd like to focus specifically on surfacing flits that are similar to other flits the user has already interacted with so we'll need feature representations of either/or our users or our content.

Let's go back to the original business question we posed at the beginning of this document: how do we recommend interesting new content for Flutter users given that we know that past content they consumed (i.e. liked and shared)?

In the traditional **collaborative filtering** approach to recommendations, we start by constructing a user-item matrix based on our input data that, when factored, gives us the latent properties of each flit and allows us to recommend similar ones.

In our case, we have Flutter users who might have liked a given flit. What other flits would we recommend given the textual properties of that one?

Here's an example. We have a flit that our bird users liked.

> "Hold fast to dreams, for if dreams die, life is a broken-winged bird that cannot fly."

We also have other flits we may or may not want to surface in our bird's feed.

> "No bird soars too high if he soars with his own wings."

> "A bird does not sing because it has an answer, it sings because it has a song."

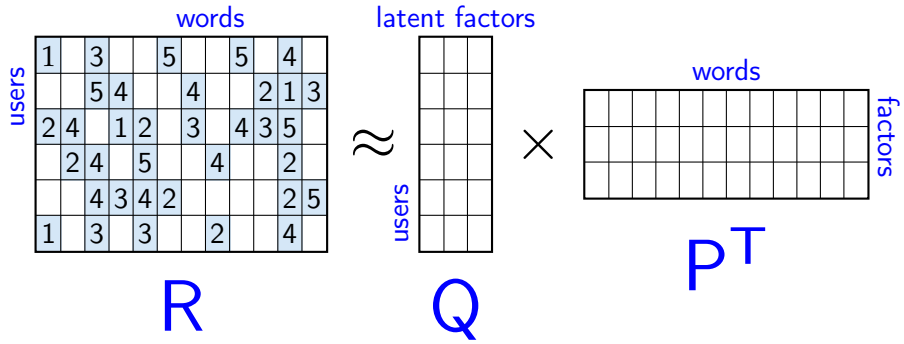
How would we turn this into a machine learning problem that takes features as input and a prediction as an output, knowing what we know about how to do this already? First, in order to build this matrix, we need to turn each word into a feature that's a column value and each user remains a row value.

The best way to think of the difference between tabular and free-form representations as model inputs is this illustration, where we're moving from looking at creating a row of individual features per bird, aka

`bird1 = [012,2,"US", 5]`, to a row a string of text. In both cases, each of these are vectors, or a list of values that represents a single bird.

In traditional machine learning, rows are our user data about a single bird and columns are features about the bird. In recommendation systems, our rows are the individual data about each user, and our column data represents the given data about each flit. If we can factor this matrix, that is decompose it into two matrices (Q and P^T) that, when multiplied, the product is our original matrix (R), we can learn the "latent factors" or features that allow us to group similar users and items together to recommend them.

Another way to think about this is that in traditional ML, we have to actively engineer features, but they are then available to us as matrices. In text and deep-learning approaches, we don't need to do feature engineering, but need to perform the extra step of generating valuable numeric features anyway.



The factorization of our feature matrix into these two matrices, where the rows in matrix Q are actually embeddings[42] for users and the rows in matrix P are embeddings for tweets, allows us to fill in values for flits that Flutter users have not explicitly liked, and then perform a search across the matrix to find other words they might be interested in. The end-result is our generated recommendation candidates, which we then filter downstream and surface to the user because the core of the recommendation problem is to recommend items to the user.

In this base-case scenario, each column could be a single word in the entire vocabulary of every flit we have and the vector we create, shown in the matrix frequency table, would be an insanely large, sparse vector that has a 0 of occurrence of words in our vocabulary. The way we can build toward this representation is to start with a structure known as a **bag of words**, or simply the frequency of appearance of text in a given document (in our case, each flit is a document.) This matrix is the input data structure for many of the early approaches to embedding.

In scikit, we can create an initial matrix of our inputs across documents using 'CountVectorizer'.

```
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd

vect = CountVectorizer(binary=True)
vects = vect.fit_transform(flits)

responses = ["Hold fast to dreams, for if dreams die, life is a broken-winged
↪  bird that cannot fly.", "No bird soars too high if he soars with his own
↪  wings.", "A bird does not sing because it has an answer, it sings because
↪  it has a song."]

doc = pd.DataFrame(list(zip(responses)))

td = pd.DataFrame(vects.todense()).iloc[:5]
td.columns = vect.get_feature_names_out()
term_document_matrix = td.T
term_document_matrix.columns = ['flit '+str(i) for i in range(1, 4)]
term_document_matrix['total_count'] = term_document_matrix.sum(axis=1)

print(term_document_matrix.drop(columns=['total_count']).head(10))

             flit_1  flit_2  flit_3
an                0       0       1
answer            0       0       1
because           0       0       1
bird              1       1       1
broken            1       0       0
cannot            1       0       0
die               1       0       0
does              0       0       1
dreams            1       0       0
fast              1       0       0
```

Figure 24: *Creating a matrix frequency table to create a user-item matrix source*

3.3.2 TF-IDF

There is a problem with the vectors we created in one-hot encoding: they are sparse. A sparse vector is one that is mostly populated by zeroes. They are sparse because most sentences don't contain all the same words as other sentences. For example, in our flit, we might encounter the word "bird" in two sentences simultaneously, but the rest of the words will be completely different.

```
sparse_vector = [1,0,0,0,0,0,0,0,0,0]
dense_vector = [1,2,2,3,0,4,5,8,8,5]
```

Figure 25: *Two types of vectors in text processing*

Sparse vectors result in a number of problems, among these **cold start**—the idea that we don't know to recommend items that haven't been interacted with, or for users who are new. What we'd like, instead, is to create dense vectors, which will give us more information about the data, the most important of which is accounting for the weight of a given word in proportion to other words. This is where we leave one-hot encodings and move into approaches that are meant to solve for this sparsity. Dense vectors are just vectors that have mostly non-zero values. We call these dense representations dynamic representations[66].

To address the limitations of one-hot encoding, TF-IDF, or term frequency-inverse document frequency was developed. TF-IDF was introduced in the 1970s[19] as a way to create a vector representation of a document by averaging all the document's word weights. It worked really well for a long time and still does in many cases. For example, one of the most-used search functions, BM25, uses TF-IDF as a baseline [54] as a default search strategy in Elasticsearch/Opensearch [20]. It extends TF-IDF to develop a probability associated with the probability of relevance for each pair of words in a document and it is still being applied in neural search today [63].

TF-IDF will tell you how important a single word is in a corpus by assigning it a weight and, at the same time, down-weight common words like "the, a, and." This calculated weight gives us a feature for a single word TF-IDF, and also the relevance of the features across the vocabulary.

We take all of our input data that's structured in sentences and break it up into individual words, and perform counts on its values, generating the bag of words. TF is term frequency, or the number of times a term appears in a document relative to the other terms in the document.

$$\text{tf}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \tag{4}$$

And IDF is the inverse frequency of the term across all documents in our vocabulary.

$$\text{idf}(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|} \tag{5}$$

Let's take a look at how to implement it from scratch:

[19] By Karen Spärck Jones, whose paper, "Synonymy and semantic classification is fundamental to the field of NLP

[20] You can read about how Elasticsearch implements BM25 here

```
1
2  import math
3
4  # Process documents into individual words
5  documentA  = ['Hold','fast','to','dreams','for','if','dreams','die,'
   ↪  ,'life','is','a','broken-winged','bird','that','cannot','fly']
6  documentB =  ['No','bird','soars','too','high','if',
   ↪  'he','soars','with','his','own','wings']
7
8  def tf(term: str, document:list[str]) -> float:
9      '''Term frequency of a word in a document  / total words in document'''
10     term_count = 0
11     total_count = 0
12
13     for word in document:
14             total_count +=1
15             if word == term:
16                     term_count += 1
17
18     return (term_count / total_count)
19
20 def idf(term:str, doc_list:list[str]) -> float:
21     '''Inverse frequency of term across a set of documents'''
22     total_docs = 0
23     total_docs_with_term = 0
24
25     for doc in doc_list:
26             total_docs +=1
27             if term in doc:
28                     total_docs_with_term +=1
29
30     idf = math.log(total_docs / total_docs_with_term)
31     return idf
32
33 def tf_idf(tf:float, idf:float) -> float:
34     tfidf = tf*idf
35     print("TF-IDF:{:0.4f}".format(tfidf))
36
37 tf = tf('dreams', documentA)
38 idf_docs = idf('dreams',[documentA, documentB])
39
40
41 tf_idf(tf, idf_docs)
42 # TF-IDF:0.0866
```

Figure 26: *Implementation of TF-IDFsource*

Once we understand the underlying fundamental concept, we can use the scikit-learn implementation which does the same thing, and also surfaces the TF-IDF of each word in the vocabulary.

```
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = ["Hold fast to dreams, for if dreams die, life is a broken-winged
↪  bird that cannot fly.",
              "No bird soars too high if he soars with his own wings."]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
dict(zip(vectorizer.get_feature_names_out(), X.toarray()[0]))

# How common or unique a word is in a vocabulary
{'bird': 0.17250274745682542,
 'broken': 0.24244659260336252,
 'cannot': 0.24244659260336252,
 'die': 0.24244659260336252,
 'dreams': 0.48489318520672503,
 'fast': 0.24244659260336252,
......}
```

Figure 27: *Implementation of TF-IDF in scikit-learnsource*

Given that inverse document frequency is a measure of whether the word is common or not across the documents, we can see that "dreams" is important because they are rare across the documents and therefore interesting to us more so than "bird." You'll also note the results are different than in our from-scratch implementation and this is dependent on how scikit processes the vocabulary.

TF-IDF enforces several important ordering rules on our text corpus:

- Uprank term frequency when it occurs many times in a small number of documents
- Downrank term frequency when it occurs many times in many documents, aka is not relevant
- Really downrank the term when it appears across your entire document base[54].

There are numerous ways to calculate and create weights for individual words in TF-IDF. In each case, we calculate a score for each word that tells us how important that word is in relation to each other word in our corpus.

Once we figure out how common each word is in the set of all possible flits and get a weighted score for the entire sentence in relation to other sentences.

Generally, when we work with textual representations, we're trying to understand which words, phrases, or concepts are similar to each other. Within our specific recommendations task, we are trying to understand which pieces of content are similar to each other, so that we can recommend content that users will like based on either their item history or the user history of users similar to them.

So, when we perform embedding in the context of recommender systems, we are looking to create neighborhoods from items and users, based on the activity of those users on our platform. This is the initial solution to the

problem of "how do we recommend flits that are similar to flit that the user has liked." This is the process of collaborative filtering.

There are many approaches to collaborative filtering including a neighborhood-based approach, which looks at weighted averages of user ratings and computes **cosine similarity**, between users. It then finds groups, or neighborhoods of users which are similar to each other.

A key problem that makes up the fundamental problem in collaborative filtering and in recommendation systems in general is the ability to find similar sets of items among very large collections[41].

Mathematically, we can do this by looking at the distance metric between any two given sets of items, and there are a number of different approaches, including Euclidean distance, edit distance (more specifically, Levenshtein distance and Hamming distance), cosine distance, and more advanced compression approaches like **minhashing**.

The most commonly used approach in most models where we're trying to ascertain the semantic closeness of two items is **cosine similarity**, which is the cosine of the angle between two objects represented as vectors, bounded between -1 and 1. -1 means the two items are completely "opposite" of each other and 1 means they are completely the same item. Zero means that you should probably use a distance measure other than cosine similarity because the vectors are entirely unrelated. One point of clarification here is that cosine distance is the actual distance measure and is calculated as $1 - cosinesimilarity$.

$$\text{similarity}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}} \tag{6}$$

We use cosine similarity over other measures like Euclidean distance for large text corpuses, for example, because in very large, sparse spaces, the direction of the vectors is just as, and even more important, than the actual values.

The higher the cosine similarity is for two words or documents, the better. We can use TF-IDF as a way to look at cosine similarity. Once we've given each of our words a tf-idf score, we can also assign a vector to each word in our sentence, and create a vector out of each quote to assess how similar they are.

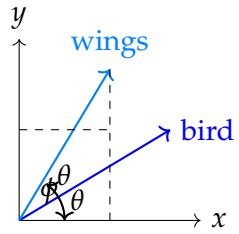


Figure 28: *Illustration of cosine similarity between bird and wings vectors.*

Let's take a look at the actual equation for cosine similarity. We start with

the dot product between two vectors, which is just the sum of each value multiplied by the corresponding value in our second vector, and then we divide by the normalized dot product.

```
v1 = [0,3,4,5,6]
v2 = [4,5,6,7,8]

def dot(v1, v2):
    dot_product = sum((a * b) for a,b in zip(v1,v2))
    return dot_product

def cosine_similarity(v1, v2):
    '''
    (v1 dot v2)/||v1|| *||v2||)
    '''
    products = dot(v1,v2)
    denominator = ( (dot(v1,v1) **.5) * (dot(v2,v2) ** .5) )
    similarity = products / denominator
    return similarity

print(cosine_similarity(v1, v2))
# 0.9544074144996451
```

Figure 29: *Implementation of cosine similarity from scratch source*

Or, once again, in scikit-learn, as a pairwise metric:

```
from sklearn.metrics import pairwise

v1 = [0,3,4,5,6]
v2 = [4,5,6,7,8]

# need to be in numpy data format
pairwise.cosine_similarity([v1],[v2])
# array([[0.95440741]])

```

Figure 30: *Implementation of cosine similarity in scikitsource*

Other commonly-used distance measures in semantic similarity and recommendations include:

- Euclidean distance - calculates the straight-line distance between two points
- Manhattan Distance - Measures the distance between two points by summing the absolute differences of their coordinates
- Jaccard Distance - Computes the dissimilarity between two sets by dividing the size of their intersection by the size of their union. Jaccard distance is caulcuated

- Hamming Distance - Measures the dissimilarity between two sets by dividing the size of their intersection by the size of their union

3.3.3 SVD and PCA

Several other related early approaches were used in lieu of TF-IDF for creating compact representations of items: **principal components analysis**(PCA) and **singular value decomposition**(SVD).

SVD and PCA are both dimensionality reduction techniques that, applied through matrix transformations to our original text input data, show us the latent relationship between two items by breaking items down into latent components through matrix transformations.

SVD is a type of matrix factorization that represents a given input feature matrix as the product of three matrices. It then uses the component matrices to create linear combinations of features that are the largest differences from each other and which are directionally different based on the variance of the clusters of points from a given line. Those clusters represent the "feature clusters" of the compressed features.

In the process of performing SVD and decomposing these matrices, we generate a matrix representation that includes the eigenvectors and eigenvalue pairs or the sample covariance pairs.

$$A = U\Sigma V^T = \underbrace{\begin{bmatrix}\mathbf{u}_1 & \mathbf{u}_2 & \dots & \mathbf{u}_r\end{bmatrix}}_{\text{Col } A} \underbrace{\begin{bmatrix}\mathbf{u}_{r+1} & \dots & \mathbf{u}_m\end{bmatrix}}_{\text{Nul } A^T} \begin{bmatrix}\sigma_1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 & 0 & \dots & 0 \\ \vdots & & & & & & \\ 0 & 0 & \dots & \sigma_r & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & & & & & & \\ 0 & 0 & \dots & 0 & 0 & \dots & 0\end{bmatrix} \begin{bmatrix}\mathbf{v}_1^T \\ \mathbf{v}_2^T \\ \vdots \\ \mathbf{v}_r^T \\ \mathbf{v}_{r+1}^T \\ \vdots \\ \mathbf{v}_n^T\end{bmatrix} \begin{matrix}\} \text{Row } A \\ \\ \\ \\ \} \text{Nul } A\end{matrix}$$

Figure 31: *SVD decomposes our feature matrix into three matrices*

PCA uses the same initial input feature matrix, but whereas one-hot encoding simply converts the text features into numerical features that we can work with, PCA also performs compression and projects our items into a two-dimensional feature space. The first principal component is the scaled eigenvector of the data, the weights of the variables that describe your data best, and the second is the weights of the next set of variables that describe your data best.

The resulting model is a projection of all the words, clustered into a single space based on these dimensions. While we can't get individual meanings of all these components, it's clear that the clusters of words, aka features, are **semantically similar**, that is they are close to each other in meaning[21]. This can be conceptually hard to visualize, so here's an illustration of the

[21]There are many definitions of semantic similarity - what does it mean for "king" and "queen" to be close to each other? - but a high-level approach involves using original sources like thesauri and dictionaries to create a structured knowledge base and offer a structured representation of terms and concepts based on nodes and edges, aka how often they appear near each other. [6]

principal components of the iris dataset among three features, represented in two vectors.

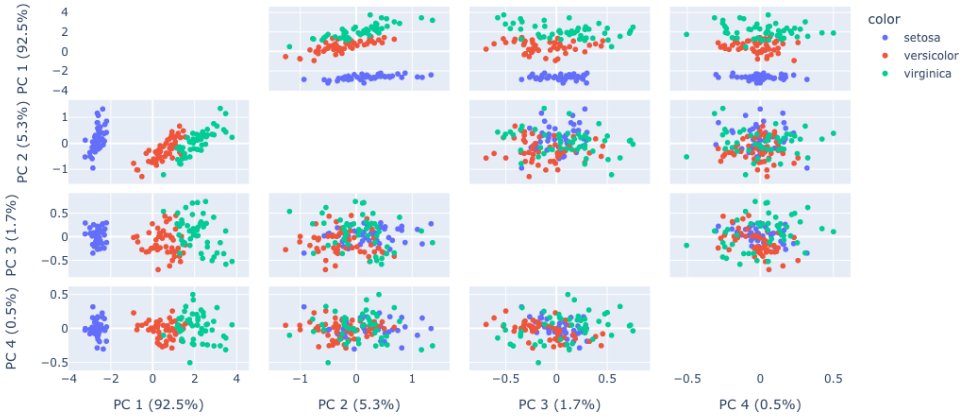


Figure 32: *PCA creates grouped clusters*

The difference between the two is often confusing (people admitted as much in the 80s [20] when these approaches were still being worked out), and for the purposes of this survey paper we'll say that PCA can often be implemented using SVD [22].

3.4 LDA and LSA

Because PCA performs computation on each combination of features to generate the two dimensions, it becomes immensely computationally expensive as the number of features grows. Many of these early methods, like PCA, worked well for smaller datasets, like many of the ones used in traditional NLP research, but as datasets continued to grow, they didn't quite scale.

Other approaches grew out of TF-IDF and PCA to address their limitations, including **latent semantic analysis** (LSA) and **latent dirichlet analysis** (LDA) LDA[12]. Both of these approaches start with the input document matrix that we built in the last section. The underlying principle behind both of these models is that words that occur close together more frequently have more important relationships. LSA uses the same word weighting that we used for TF-IDF and looks to combine that matrix into a lower rank matrix, a cosine similarity matrix. In the matrix, the values for the cells range from [-1,1], where -1 represents documents that are complete opposites and 1 means the documents are identical. LSA then runs over the matrix and groups items together.

LDA takes a slightly different approach. Although it uses the same matrix for input, it instead outputs a matrix where the rows are words and columns are documents. The distance measure, instead of cosine similarity, is the numerical value for the topic that the intersection of the word and document provide. The assumption is that any sentence we input will contain a collection of topics, based on proportions of representation in relation to the input

[22] This is how it's implemented in the scikit-learn package

corpus, and that there are a number of topics that we can use to classify a given sentence. We initialize the algorithm by assuming that there is a non-zero probability that each word could appear in a topic. LDA initially assigns words to topics at random, and then iterates until it converges to a point where it maximizes the probability for assigning a current word to a current topic. In order to do the word-to-topic mapping, LDA generates an embedding that creates a space of clusters of words or sentences that work together semantically.

3.5 Limitations of traditional approaches

All of these traditional methods look to address the problem of generating relationships between items in our corpus in various ways in the latent space - the relationships between words that are not explicitly stated but that we can tease out based on how we model the data.

However, in all these cases, as our corpus starts to grow, we start to run into two problems: the curse of dimensionality and compute scale.

3.5.1 The curse of dimensionality

As we one-hot encode more features, our tabular data sets grows. Going back to our churn model, what happens once we have 181 instead of two or three countries? We'll have to encode each of them into their own vector representations. What happens if we have millions of vocabulary words, for example thousands of birds posting millions of messages every day? Our sparse matrix for tf-idf becomes computationally intensive to factor.

Whereas our input vectors for tabular machine learning and naive text approaches is only three entries because we only use three features, multimodal data effectively has a dimensionality of the number of written words in existence and image data has a dimensionality of height times width in pixels, for each given image. Video and audio data have similar exponential properties. We can profile the performance of any code we write using **Big O notation**, which will classify an algorithm's runtime. There are programs that perform worse and those that perform better based on the number of elements the program processes. This means that one-hot encodings, in terms of computing performance, are $O(n)$ in the worst case complexity. So, if our text is a corpus of a million unique words, we'll get to a million columns, or vectors, each of which will be sparse, since most sentences will not contain the words of other sentences.

Let's take a more concrete case. Even in our simple case of our initial bird quote, we have 28 features, one for each word in the sentence, assuming we don't remove and process the most common **stop words** — extremely common words like "the", "who", and "is" that appear in most texts but don't add semantic meaning. How can we create a model that has 28 features? That's fairly simple if tedious - we encode each word as a numerical value.

Table 4: *One-hot encoding and the growing curse of dimensionality for our flit*

flit_id	bird_id	hold	fast	dreams	die	life	bird
9823420	012	1	1	1	1	1	1
9823421	013	1	0	0	0	0	1

Not only will it be hard to run computations over an linearly increasing set, once we start generating a large number of features (columns), we start running into the **curse of dimensionality**, which means that, the more features we accumulate, the more data we need in order to accurately statistically confidently say anything about them, which results in models that may not accurately represent our data[28] if we have extremely sparse features, which is generally the case in user/item interactions in recommendations.

3.5.2 Computational complexity

In production machine learning systems,the statistical properties of our algorithm are important. But just as critical is how quickly our model returns data, or the system's efficiency. System efficiency can be measured in many ways, and it is critical in any well-performing system to find the performance bottleneck that leads to latency, or the time spent waiting before an operation is performed[25]. If you have a recommendation system in production, you cannot risk showing the user an empty feed or a feed that takes more than a few milliseconds to render. If you have a search system, you cannot risk the results taking more than a few milliseconds to return, particularly in e-commerce settings [2]. From the holistic systems perspective then, we can also have latency in how long it takes to generate data for a model, read in data, and train the model.

The two big drivers of latency are:

- I/O processing - We can only send as many items over the network as our network speed allows
- CPU processing - We can only process as many items as we have memory available to us in any given system[23]

Generally, TF-IDF performs well in terms of identifying key terms in the document. However, since the algorithm processes all the elements in a given corpus, the time complexity grows for both the numerator and the denominator in the equation and overall, the time-complexity of computing the TF-IDF weights for all the terms in all the documents is $O(Nd)$, where N is the total number of terms in the corpus and d is the number of documents in the corpus. Additionally, because TF-IDF creates a matrix as output, what we end up doing is processing enormous state matrices. For example, if you have 100k documents and need to store frequency counts and features for the top five thousand words appearing in those documents, we get a matrix of size $100000 * 5000$. This complexity only grows.

[23] there has been discussion over the past few years on whether IO or CPU are really the bottleneck in any modern data-intensive application.

This linear time complexity growth becomes an issue when we're trying to process millions or hundreds of millions of **tokens** – usually a synonym for words but can also be sub-words such as syllables. This is a problem that became especially prevalent as, over time in industry, storage became cheap.

From newsgroups to emails, and finally, to public internet text, we began to generate a lot of digital exhaust and companies collected it in the form of append-only logs[35], a sequence of records ordered by time, that's configured to continuously append records.[24] .

Companies started emitting, keeping, and using these endless log streams for data analysis and machine learning. All of a sudden, the algorithms that had worked well on a collection of less than a million documents struggled to keep up.

Capturing log data at scale began the rise of the Big Data era, which resulted in a great deal of variety, velocity, and volume of data movement. The rise in data volumes coincided with data storage becoming much cheaper, enabling companies to store everything they collected on racks of commodity hardware.

Companies were already retaining analytical data needed to run critical business operations in relational databases, but access to that data was structured and processed in batch increments on a daily or weekly basis. This new logfile data moved quickly, and with a level of variety absent from traditional databases.

The resulting corpuses for NLP, search, and recommendation problems also exploded in size, leading people to look for more performant solutions.

3.6 Support Vector Machines

The first modeling approaches were **shallow** models — models that perform machine learning tasks using only one layer of weights and biases[9]. **Support vector machines** (SVM), developed at Bell Laboratories in the mid-1990s, were used in high-dimensional spaces for NLP tasks like text categorization[31]. SVMs separate data clusters into points that are linearly separable by a **hyperplane**, a decision boundary that separates elements into separate classes. In a two-dimensional vector space, the hyperplane is a line, in a three or more dimensional space, the separator also comes in many dimensions.

The goal of the SVM is to find the optimal hyperplane such that the distance between new projections of objects (words in our case) into the space maximizes the distance between the plane and the elements so there's less chance of mis-classifying them.

[24] Jay Kreps' canonical posts on how logging works are a must-read

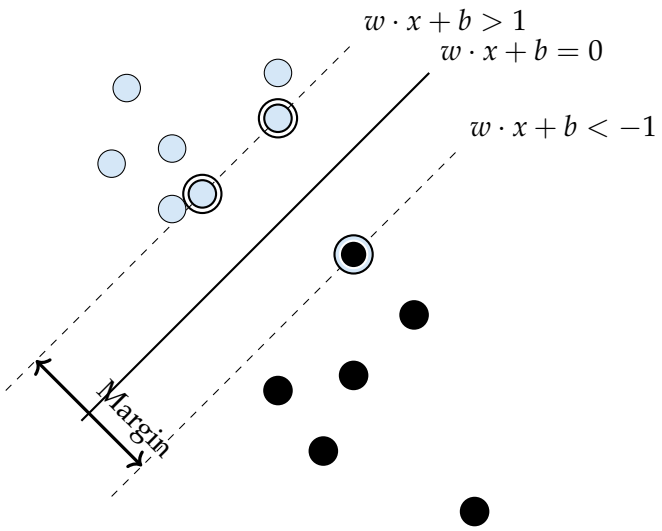


Figure 33: *Example of points in the vector space in an SVM separated by a hyperplane*

Examples of supervised machine learning tasks performed with SVMs included next word prediction, predicting the missing word in a given sequence, and predicting words that occur in a window. As an example, the classical word embedding inference task is autocorrect when we're typing on our phones. We type a word, and it's the job of the autocorrect to predict the correct word based on both the word itself and the surrounding context in the sentence. It therefore needs to learn a vocabulary of embeddings that will give it probabilities that it is selecting the correct word.

However, as in other cases, when we reach high dimensions, SVMs completely fail to work with sparse data because they rely on computing distances between points to determine the decision boundaries. Because in our sparse vector representations of elements most of the distances are zero, the hyperplane will fail to cleanly separate the boundaries and classify words incorrectly.

3.7 Word2Vec

To get around the limitations of earlier textual approaches and keep up with growing size of text corpuses, in 2013, researchers at Google came up with an elegant solution to this problem using neural networks, called Word2Vec[46].

So far, we've moved from simple heuristics like one-hot encoding, to machine learning approaches like LSA and LDA that look to learn a dataset's modeled features. Previously, like our original one-hot encodings, all the approaches to embedding focused on generating sparse vectors much . A sparse vector gives an indication that two words are related, but not that there is a semantic relationship between them. For example, "The dog changed the cat" and "the cat chased the dog" would have the same distance in the vector space, even though they're two completely different sentences.

Word2Vec is a family of models that has several implementations, each of which focus on transforming the entire input dataset into vector representations and, more importantly, focusing not only on the inherent labels of individual words, but on the relationship between those representations.

There are two modeling approaches to Word2Vec - **continuous bag of**

words (CBOW) and **skipgrams**, both of which generate dense vectors of embeddings but model the problem slightly differently. The end-goal of the Word2Vec model in either case is to learn the parameters that maximize that probability of a given word or group of words being an accurate prediction [22].

In training skipgrams, we take a word from the initial input corpus and predict the probability that a given set of words surround it. In the case of our initial flit quote, "Hold fast to dreams for if dreams die, life is a broken-winged bird that cannot fly", we train the model to generate a set of embeddings that's the distance between all the words in the dataset and fill in the next several probabilities for the entire phrase, using the word "fast" as input.

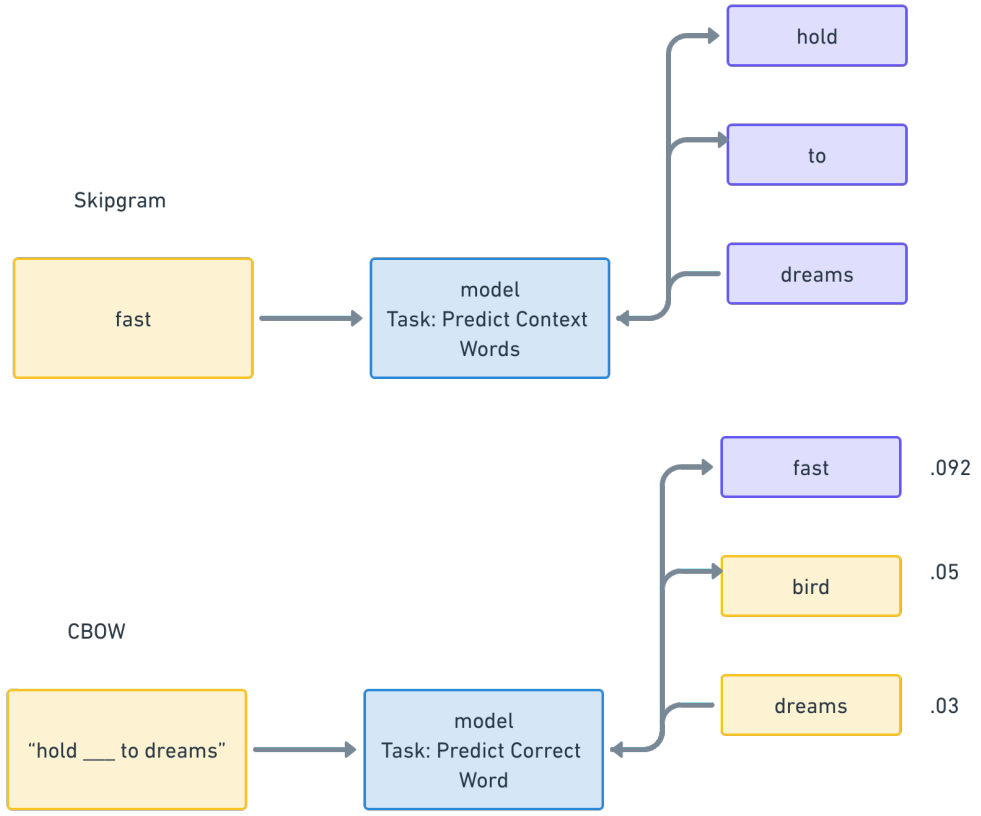


Figure 34: *Word2Vec Architecture*

In training CBOW, we do the opposite: we remove a word from the middle of a phrase known as the **context window** and train a model to predict the probability that a given word fills the blank, shown in the equation below where we attempt to maximize.

$$\arg\max_{\theta} \prod_{w \in Text} \left[\prod_{c \in C(w)} p(c|w;\theta) \right] \tag{7}$$

If we optimize these parameters - theta and Pi- and maximize the probability that the word belongs in the sentences, we'll learn good embeddings for our input corpus.

Let's focus on a detailed implementation of CBOW to better understand

how this works. This time, for the code portion, we'll move on from scikit-learn, which works great for smaller data, to PyTorch for neural net operations.

At a high level, we have a list of input words that are processed through a second layer, the embedding layer, and then through the output layer, which is just a linear model that returns probabilities.

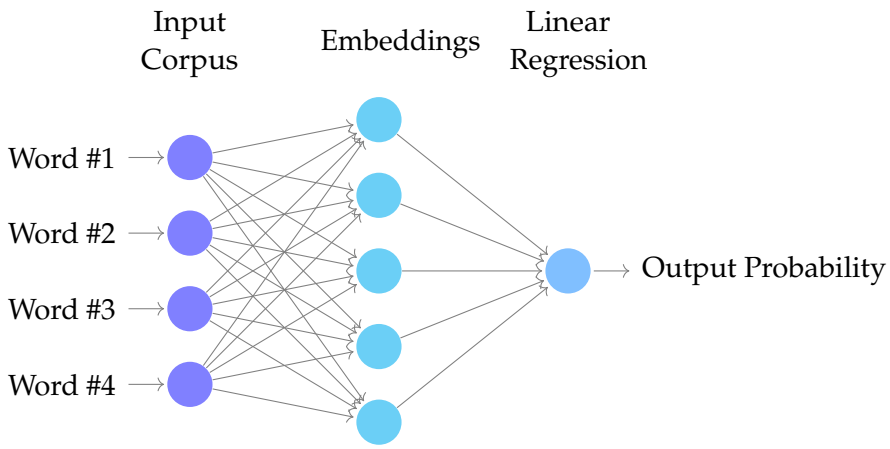


Figure 35: *Word2Vec Neural Network architecture*

We'll run this implementation in PyTorch, the popular library for building neural network models. The best way to implement Word2Vec, especially if you're dealing with smaller datasets, is using Gensim, but Gensim abstracts away the layers into inner classes, which makes for a fantastic user experience. But, since we're just learning about them, we'd like to see a bit more explicitly how they work, and PyTorch, although it does not have a native implementation of Word2Vec, lets us see the inner workings a bit more clearly.

To model our problem in PyTorch, we'll use the same approach as with any problem in machine learning:

- Inspect and clean our input data.
- Build the layers of our model. (For traditional ML, we'll have only one)
- Feed the input data into the model and track the loss curve
- Retrieve the trained model artifact and use it to make predictions on new items that we analyze

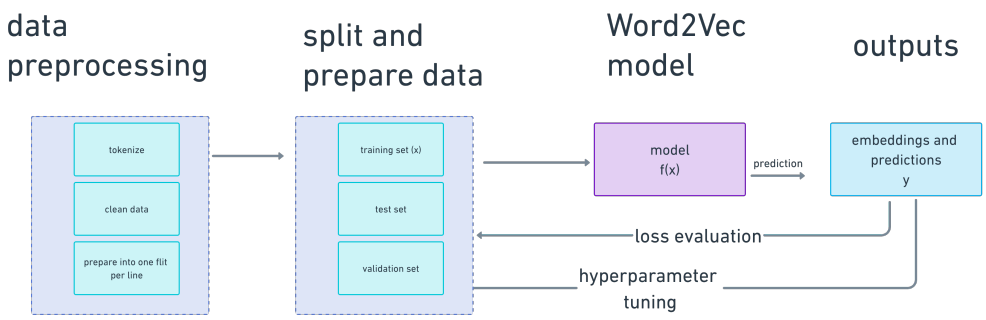


Figure 36: *Steps for creating Word2Vec model*

Let's start from our input data. In this case, our corpus is all of the flits we've collected. We first need to process them as input into our model.

```
responses = ["Hold fast to dreams, for if dreams die, life is a broken-winged
↪  bird that cannot fly.", "No bird soars too high if he soars with his own
↪  wings.", "A bird does not sing because it has an answer, it sings because
↪  it has a song."]
```

Figure 37: *Our Word2Vec input dataset*

Let's start with our input training data, which is our list of flits. To prepare input data for PyTorch, we can use the DataLoader or Vocab classes, which splits our text into tokens and **tokenizes** — or creates smaller, word-level representations of each sentence — for processing. For each line in the file, we generate tokens by splitting each line into single words, removing whitespace and punctuation, and lowercasing each individual word.

This kind of processing pipeline is extremely common in NLP and spending time to get this step right is extremely critical so that we get clean, correct input data. It typically includes[47]:

- **Tokenization** - transforming a sentence or a word into its component character by splitting it
- Removing noise - Including URLs, punctuation, and anything else in the text that is not relevant to the task at hand
- **Word segmentation** - Splitting our sentences into individual words
- Correcting spelling mistakes

```
class TextPreProcessor:
        def __init__(self) -> None:
            self.input_file = input_file

        def generate_tokens(self):
            with open(self.input_file, encoding="utf-8") as f:
                for line in f:
                    line = line.replace("\\", "")
                    yield line.strip().split()

        def build_vocab(self) -> Vocab:
            vocab = build_vocab_from_iterator(
                self.generate_tokens(), specials=["<unk>"], min_freq=100
            )
            return vocab
```

Figure 38: *Processing our input vocabulary and building a Vocabulary object from our dataset in PyTorchsource*

Now that we have an input vocabulary object we can work with, the next step is to create one-hot encodings of each word to a numerical position, and

each position back to a word, so that we can easily reference both our words and vectors. The goal is to be able to map back and forth when we do lookups and retrieval.

This occurs in the Embedding layer. Within the Embedding layer of PyTorch, we initialize an Embedding matrix based on the size we specify and size of our vocabulary, and the layer indexes the vocabulary into a dictionary for retrieval. The embedding layer is a lookup table[25] that matches a word to the corresponding word vector on an index by index basis. Initially, we create our one-hot encoded word to term dictionary. Then, we create a mapping of each word to a dictionary entry and a dictionary entry to each word. This is known as **bijection**. In this way, the Embedding layer is like a one-hot encoded matrix, and allows us to perform lookups. The lookup values in this layer are initialized to a set of random weights, which we next pass onto the linear layer.

Embeddings resemble hash maps and also have their performance characteristics ($O(1)$ retrieval and insert time), which is why they can scale easily when other approaches cannot. In the embedding layer, Word2Vec where each value in the vector represents the word on a specific dimension, and more importantly, unlike many of the other methods, the value of each vector is in direct relationship to the other words in the input dataset.

[25] Embedding Layer PyTorch documents

```
class CBOW(torch.nn.Module):
        def __init__(self):  # we pass in vocab_size and embedding_dim as hyperparams
            super(CBOW, self).__init__()
            self.num_epochs = 3
            self.context_size = 2  # 2 words to the left, 2 words to the right
            self.embedding_dim = 100  # Size of your embedding vector
            self.learning_rate = 0.001
            self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

            self.vocab = TextPreProcessor().build_vocab()
            self.word_to_ix = self.vocab.get_stoi()
            self.ix_to_word = self.vocab.get_itos()
            self.vocab_list = list(self.vocab.get_stoi().keys())
            self.vocab_size = len(self.vocab)

            self.model = None

            # out: 1 x embedding_dim
            self.embeddings = nn.Embedding(
                self.vocab_size, self.embedding_dim
            )  # initialize an Embedding matrix based on our inputs
            self.linear1 = nn.Linear(self.embedding_dim, 128)
            self.activation_function1 = nn.ReLU()

            # out: 1 x vocab_size
            self.linear2 = nn.Linear(128, self.vocab_size)
            self.activation_function2 = nn.LogSoftmax(dim=-1)
```

Figure 39: *Word2Vec CBOW implementation in Pytorchsource*

Once we have our lookup values, we can process all our words in their relationship to each other. For CBOW, we take a single word and we pick a sliding window, in our case, two words before, and two words after, and try to infer what the actual word is. This is called the **context vector**, and in other cases, we'll see that it's called attention. For example, if we have the phrase "No bird [blank] too high", we're trying to predict that the answer is "soars" with a given softmax probability, aka ranked against other words. Once we have the context vector, we look at the loss — the difference between the true word and the predicted word as ranked by probability — and then we continue.

The way we train this model is through context windows. For each given word in the model, we create a sliding window that includes that word and 2 words before it, and 2 words after it.

We activate the linear layer with a **ReLu** activation function, which decides whether a given weight is important or not. In this case, ReLu squashes all the negative values we initialize our embeddings layer with down to zero since we can't have inverse word relationships, and we perform linear regression by learning the weights of the model of the relationship of the words. Then, for each batch we examine the **loss**, the difference between the real word and

the word that we predicted should be there given the context window - and we minimize it.

At the end of each epoch, or pass through the model, we pass the weights, or **backpropagate** them, back to the linear layer,and then again, update the weights of each word, based on the probability. The probability is calculated through a **softmax** function, which converts a vector of real numbers into a probability distribution - that is, each number in the fector, i.e. the value of the probability of each words, is in the interval between 0 and 1 and all of the word numbers add up to one. The distance, as backpropogated to the embeddings table, should converge or shrink depending on the model understanding how close specific words are.

```

    def make_context_vector(self, context, word_to_ix) -> torch.LongTensor:
        """
        For each word in the vocab, find sliding windows of [-2,1,0,1,2] indexes
        relative to the position of the word
        :param vocab: list of words in the vocab
        :return: torch.LongTensor
        """
        idxs = [word_to_ix[w] for w in context]
        tensor = torch.LongTensor(idxs)


    def train_model(self):

        # Loss and optimizer
        self.model = CBOW().to(self.device)
        optimizer = optim.Adam(self.model.parameters(), lr=self.learning_rate)
        loss_function = nn.NLLLoss()

        logging.warning('Building training data')
        data = self.build_training_data()

        logging.warning('Starting forward pass')
        for epoch in tqdm(range(self.num_epochs)):
            # we start tracking how accurate our initial words are
            total_loss = 0

            # for the x, y in the training data:
            for context, target in data:
                context_vector = self.make_context_vector(context, self.word_to_ix)

                # we look at loss
                log_probs = self.model(context_vector)

                # compare loss
                total_loss += loss_function(
                    log_probs, torch.tensor([self.word_to_ix[target]])
                )

            # optimize at the end of each epoch
            optimizer.zero_grad()
            total_loss.backward()
            optimizer.step()

            # Log out some metrics to see if loss decreases
            logging.warning("end of epoch {} | loss {:2.3f}".format(epoch, total_loss))

        torch.save(self.model.state_dict(), self.model_path)
        logging.warning(f'Save model to {self.model_path}')
```

Figure 40: *W*
ord2Vec CBOW implementation in Pytorch, see full implementation here

Once we've completed our iteration through our training set, we have learned a model that retrieves both the probability of a given word being the correct word, and the entire embedding space for our vocabulary.

4 Modern Embeddings Approaches

Word2Vec became one of the first neural network architectures to use the concept of embedding to create a fixed feature vocabulary. But neural networks as a whole were gaining popularity for natural language modeling because of several key factors. First, in the 1980s, Geoffrey Hinton, a pioneer in the field of deep learning, co-published a paper on **backpropagation**[51]. Backpropagation is how a model learns to converge by calculating the gradient of the loss function with respect to the weights of the neural network, using the **chain rule**, a concept from calculus which allows us to calculate the derivative of a function made up of multiple functions. This mechanism allows the model to understand when it's reached a global minimum for loss and picks the correct weights for the model parameters, but training models through gradient descent. Earlier approaches, such as the perceptron learning rule, tried to do this, but had limitations, such as being able to work only on simple layer architectures, took a long time to converge, and experienced **vanishing gradients**, which made it hard to effectively update the model's weights.

These advances gave rise to the first kinds of multi-level neural networks, feed-forward neural networks. In 1998, a paper from used backpropagation over multilayer perceptrons to correctly perform the task of recognizing handwritten digit images[39], demonstrating a practical use-case practitioners and researchers could apply This MNIST dataset is now one of the canonical "Hello World" examples of deep learning.

Second, in the 2000s, the rise of petabytes of aggregated log data resulted in the creation of large databases of multimodal input data scraped from the internet. This made it possible to conduct wide-ranging experiments to prove that neural networks work on large amounts of data. For example, ImageNet was developed by researchers at Stanford who wanted to focus on improving model performance by creating a gold set of neural network input data, the first step in processing. FeiFei Li assembled a team of students and paid gig workers from Amazon Turk to correctly label a set of 3.2 million images scraped from the internet and organized based on categories according to WordNet, a taxonomy put together by researchers in the 1970s[53].

Researchers saw the power of using standard datasets. In 2015, Alex Krizhevsky, in collaboration with Ilya Sutskever, who now works at OpenAI as one of the leading researchers behind the GPT series of models that form the basis or the current generative AI wave, submitted an entry to the ImageNet competition called AlexNet. This model was a convolutional neural network that outperformed many other methods. There were two things that were significant about AlexNet. The first was that it had eight stacked layers of weights and biases, which was unusual at the time. Today, 12-layer neural. networks like BERT and other transformers are completely normal, but at the time, more than two layers was revolutionary. The second was that it ran on GPUs, a new architectural concept at the time, since GPUs were used mostly

for gaming.

Neural networks started to become popular as ways to generate representations of vocabularies. In particular, neural network architectures, such as **long short-term memory networks** (LSTMs) and **recurrent neural networks** (RNNs) also emerged as ways to deal with textual data for all kinds of machine learning tasks from NLP to computer vision.

4.1 Neural Networks

Neural networks are extensions on traditional machine learning models, but they have a few critical special properties. Let's think back to our definition of a model when we formalized a machine learning problem. A model is a function with a set of learnable input parameters that takes some set of inputs and one set of tabular input features, and gives us an output. In traditional machine learning approaches, there is one set, or layer, of learnable parameters and model. If our data doesn't have complex interactions, our model can learn the feature space fairly easily and make accurate predictions.

However, when we start dealing with extremely large, implicit feature spaces, such as are present in text, audio, or video, we will not be able to derive specific features that wouldn't be obvious if we were manually creating them. A neural network, by stacking neurons, each of which represent some aspect of the model, can tease out these latent representations. Neural networks are extremely good at learning representations of data, with each level of the network transforming a learned representation of the level to a higher level until we get a clear picture of our data[40].

4.1.1 Neural Network architectures

We've already encountered our first neural network, Word2Vec, which seeks to understand relationships between words in our text that the words themselves would not tell us. Within the neural network space, there are several popular architectures:

- Feed-forward networks that extract meaning from fixed-length inputs. Results of these model are not fed back into the model for iteration
- Convolutional neural nets (CNNs) - used mainly for image processing, which involves a convolutional layer made up of a filter that moves across an image to check for feature representations which are then multiplied via dot product with the filter to pull out specific features
- textbfrecurrent neural networks, which take a sequence of items and produce a vector that summarizes the sentence

RNNs and CNNs are used mainly in feature extraction - they generally do not represent the entire modeling flow, but are fed later into feed-forward models that do the final work of classification, summarization, and more.

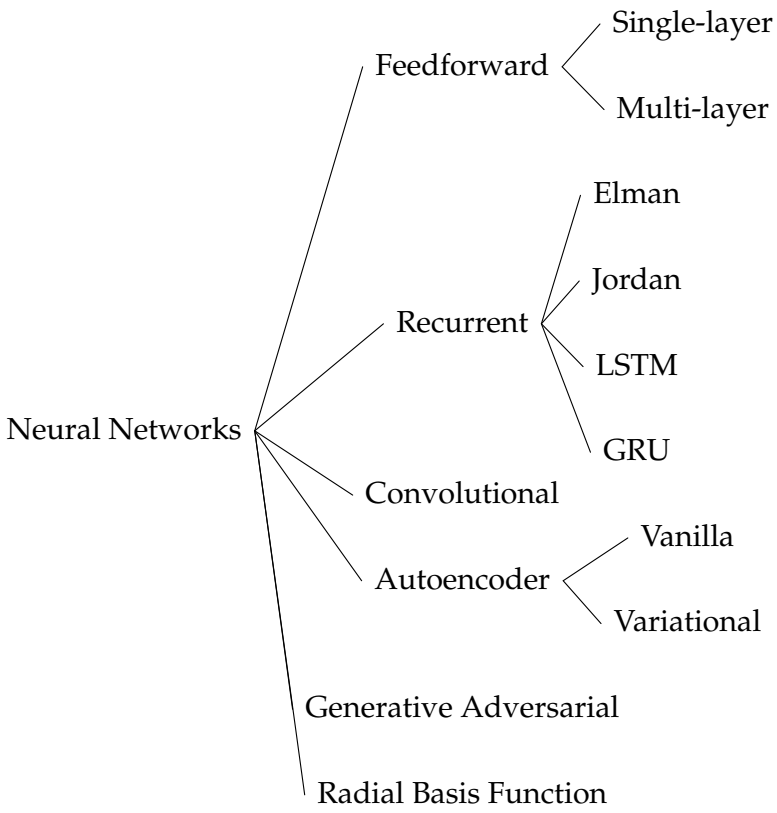


Figure 41: *Types of Neural Networks*

Neural networks are complex to build and manage for a number of reasons. First, they require extremely large corpuses of clean, well-labeled data. They also require special GPU architectures for processing, and, as we'll see in the production section, they have their own metadata management and latency considerations. Finally, within the network itself, we need to complete a large amount of passes we need to do over the model object using batches of our training data to get it to converge. The number of feature matrices that we need to run calculations over[26], and, consequently, the amount of data we have to keep in-memory through the lifecycle of the model ends up accumulating and requires a great deal of performance tuning.

These features made developing and running neural networks prohibitively expensive until the last fifteen years or so. First, the exponential increase in storage space provided by the growing size of commodity hardware both on-prem and in the cloud meant that we could now store that data for computation, and the explosion of log data gave companies such as Google a lot of training data to work with. Second, the rise of the GPU as a tool that takes advantage of the neural network's ability to perform **embarrassingly parallel** computation — a characteristic of computation when it's easy to separate steps out into ones that can be performed in parallel, such as word count for example . In a neural network, we can generally parallelize computation in any given number of ways, including at the level of a single neuron.

[26]There is no good single resource for calculating the computational complexity of a neural network given that there are many wide-ranging architectures but this post does a good job laying out the case that it's essentially $O(n^5)$

While GPUs were initially used for working with computer graphics, in the early 2000s[48], researchers discovered the potential to use them for general computation, and Nvidia made an enormous bet on this kind of computing by introducing CUDA, an API layer on top of GPUs, specifically for interfacing with PyTorch and Tensorflow and other deep learning frameworks.

Neural networks could now be trained and experimented with at scale. To come back to a comparison to our previous approaches, when we calculate TF-IDF, we need to loop over each individual word and perform our computations over the entire dataset in sequence to arrive at a score in proportion to all other words, which means that our computational complexity will be $O(ND)$[10].

However, with a neural network, we can either distribute the model training across different GPUs in a process known as **model parallelism**, or compute **batches** — the size of the training data fed into the model and used in a training loop before its hyperparameters are updated in parallel and update at the end of each **minibatch**, which is known as **data parallelism.**[58].

4.2 Transformers

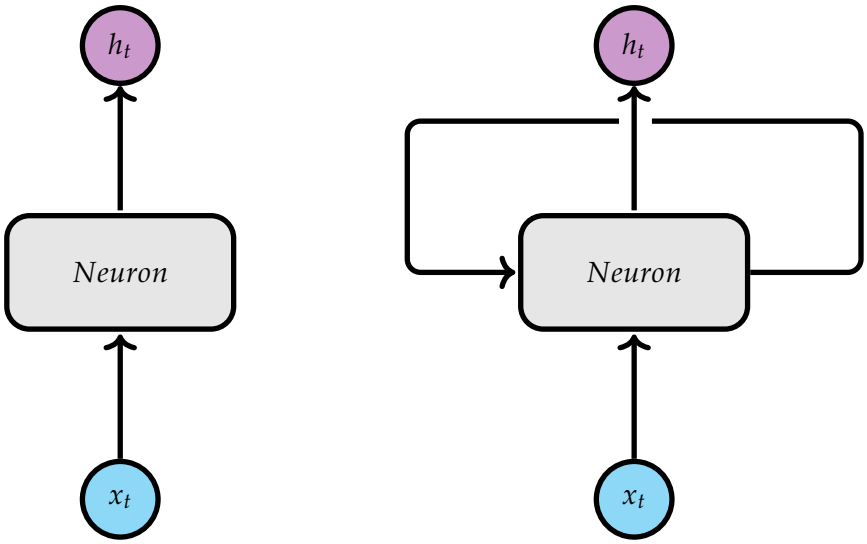
Word2Vec is a feed-forward network. The model weights and information only flows from the encoding state, to the hidden embedding layer, to the output probability layer. There is no feedback between the second and third layers, which means that each given layer doesn't know anything about the state of the layers that follow it. It can't make inference suggestions longer than the context window. This works really well for machine learning problems where we're fine with a single, static vocabulary.

However, it doesn't work well on long ranges of text that require understanding words in context of each other. For example, over the course of a conversation, we might say, "I read that quote by Langston Hughes. I liked it, but didn't really read his later work," we understand that "it" refers to the quote, context from the previous sentence, and "his" refers to "Langston Huges", mentioned two sentences ago.

One of the other limitations was that Word2Vec can't handle **out-of-vocabulary** words — words that the model has not been trained on and needs to generalize to. This means that if our users search for a new trending term or we want to recommend a flit that was written after our model was trained, they won't see any relevant results from our model.[14], unless the model is retrained frequently.

Another problem is that Word2Vec encounters context collapse around **polysemy** — the coexistence of many possible meanings for the same phrase: for example, if you have "jail cell" and "cell phone" in the same sentence, it won't understand that the context of both words is different. Much of the work of NLP based in deep learning has been in understanding and retaining that context to propagate through the model and pull out semantic meaning.

Different approaches were proposed to overcome these limitations. Researchers experimented with **recurrent neural networks**, RNNs. An RNN builds on traditional feed-forward networks, with the difference being that layers of the model give feedback to previous layers. This allows the model to keep memory of the context around words in a sentence.



(a) *Feedforward Neural Network* **(b)** *Recurrent Neural Network*

A problem with traditional RNNs was that because during backpropagation the weights had to be carried through to the previous layers of neurons, they experienced the problem of **vanishing gradients**. This occurs when we continuously take the derivative such that the partial derivative used in the chain rule during backpropagation approaches zero. Once we approach zero, the neural network assumes it has reached a local optimum and stops training, before convergence.

A very popular variation of an RNN that worked around this problem was the **long-short term memory network (LSTM)**, developed initially by Schmidhuber[27] and brought to popularity for use in text applications speech recognition and image captioning[32]. Whereas our previous model takes only a vector at a time as input, RNNs operate on sequences of vectors using a mechanism known as **gated recurrent units**, which allows the network to control how much information is passed in for analysis. While LSTMs worked fairly well, they had their own limitations. Because they were architecturally complicated, they took much longer to train, and at a higher computational cost, because they couldn't be trained in parallel.

4.2.1 Encoders/Decoders and Attention

Two concepts allowed researchers to overcome computationally expensive issues with remembering long vectors for a larger **context window** than what was available in RNNs and Word2Vec before it: the **encoder/decoder architecture**, and the **attention mechanism**.

The encoder/decoder architecture is a neural network architecture comprised of two neural networks, an encoder that takes the input vectors from our data and creates an embedding of a fixed length, and a decoder, also a neural network, which takes the embeddings encoded as input and generates a static set of outputs such as translated text or a text summary. In between the two types of layers is the **attention mechanism**, a way to hold the state

[27] If you read Schmidhuber, you will come the understanding that everything in deep learning was developed initially by Schmidhuber

of the entire input by continuously performing weighted matrix multiplications that highlight the relevance of specific terms in relation to each other in the vocabulary. We can think of attention as a very large, complex hash table that keeps track of the words in the text and how they map to different representations both in the input and the output.

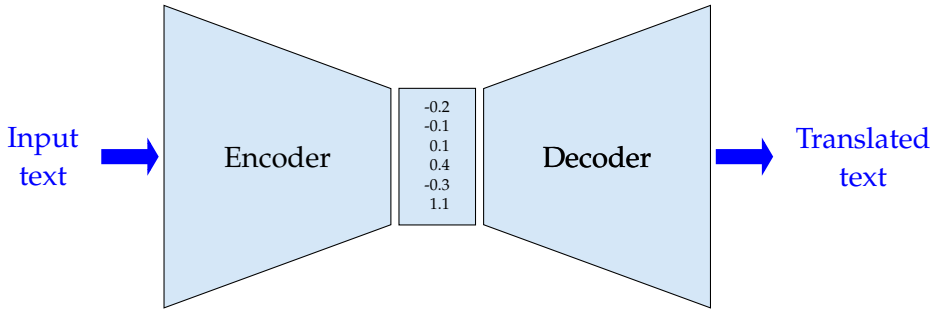


Figure 43: *The encoder/decoder architecture*

"Attention is All You Need"[64], released in 2017, combined both of these concepts into a single architecture. The paper immediately saw a great deal of success, and today Transformers are one of the de-facto models used for natural language tasks.

Based on the success of the original model, a great deal of variations on Transformer architectures have been released, followed by GPT and BERT in 2018, Distilbert, a smaller and more compact version of BERT in 2019, and GPT-3 in 2020[28].

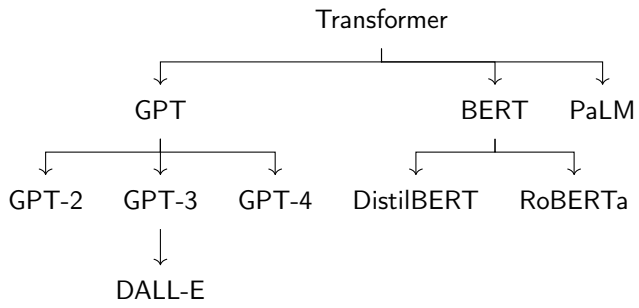


Figure 44: *Timeline of Transformer Models*

[28]For a complete visual transformer timeline, check out this link

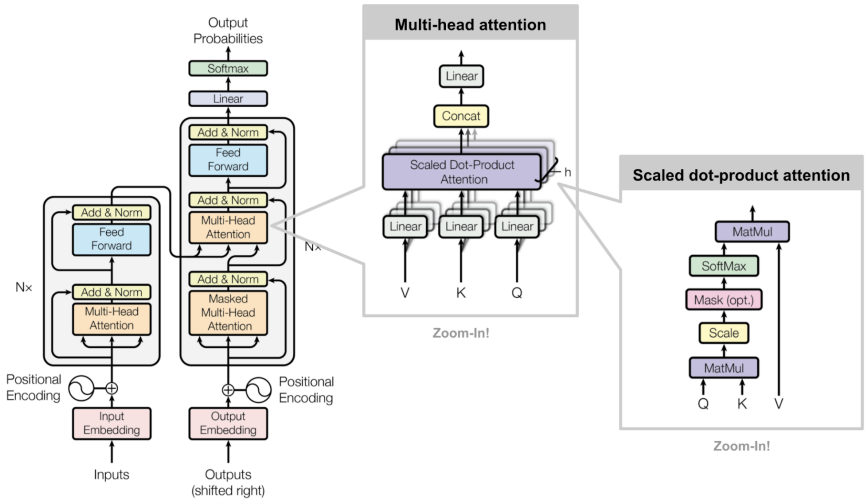


Figure 45: *Transformer architecture from the Attention is All You Need Paper, combined in this blog post*

Transformer architectures themselves are not new, but they contain all the concepts we've discussed so far: vectors, encodings, and hash maps. The ultimate goal of a transformer model is to take a piece of multimodal content, and learn the latent relationships by creating multiple videos of groups of words in the input corpus (multiple context windows). The self-attention mechanism, implemented as scaled dot-product attention in the Transformer paper, creates different context windows of the data a number of times through the six encoder and six decoder layers. The output is the result of the specific machine learning task — a translated sentence, a summarized paragraph – and the next-to-last layer is the model's embeddings, which we can use for downstream work.

Within the Transformer paper, we have six layers that perform encoding and six that perform decoding. As input into the model, we read in our corpus of flits. Each of the encoder's input layers consists of two sub-layers: a self-attention layer and a feed-forward neural network. In the encoder, we start by doing the same thing we've done in our previous data: we first convert our text to token embeddings. This is the same process as in Word2Vec: we simply assign each word to a position in a matrix and are able to retrieve it by index. These alone, though, will not help us with the context. So what we end up generating is a set of **positional embeddings**, which is an encoding process that incorporates the index of a given word in the sentence in addition to the token itself. The positional encoding is generated by sine and cosine functions of different frequencies. We are essentially capturing both the token itself and its position in relation to other tokens in the text. The vector values are initially set at random and iterated on through backpropagation.

Then, those embeddings are propagated to two layers: the self-attention model, and the feed-forward network. We pass the tokens through multi-headed self-attention using scaled dot product attention, which is specific to the transformer architecture.

Attention is a mechanism used in a feed-forward network model that creates an embedding matrix on several different dimensions. Attention is

just a term for how much we want to weigh the embeddings in a given matrix relative to the other embeddings. It allows us to capture the context in a sentence or group of sentences. The first layer is self-attention. This layer tells the other layers what to pay attention to in the context of the sentence. For each embedding, we generate a weighted average of each embedding based on attention weights. In transformer models, this is done using scaled dot-product attention.

The scaled dot-product attention is the product of three matrices: key, query, and value. These are initially all the same values that are outputs of previous layers - in the first pass through the model, they are initially all the same, initialized at random and adjusted at each step by gradient descent. For each embedding, we generate a weighted average value based on these learned attention weights. We calculate the dot product between query and key, and finally normalize the weights via softmax.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{8}$$

Each decoder layer has a self-attention component, an encoder-decoder attention component, and a feed-forward neural network layer. The decoder layers work the same way, using K and V from the encoder-decoder attention layer, which focuses the decoder on the proper output.

Within the decoder, we then finally embed and add positional encodings to fix the relationship of the vocabulary.

The final two layers of the Transformer model are a linear layer, which projects the float vector of embeddings plus positional encodings into a word by turning the score at each vector position into words with the help of a softmax layer, which turns the score into probabilities. The highest score is the output of what the word should be, plus its index in the embedding matrix.

Initially, we have our input data, the matrix operations of the encoder that embeds the text in the space of continuous numerical representations, and the decoder takes those numerical representations, one element at a time and generates an output sequence. At each step of the process, the decoder looks at the previous steps and generates based on those steps so we form a complete sentence[52].

```
class EncoderDecoder(nn.Module):
        """
           Defining the encoder/decoder steps
        """
        def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
            super(EncoderDecoder, self).__init__()
            self.encoder = encoder
            self.decoder = decoder
            self.src_embed = src_embed
            self.tgt_embed = tgt_embed
            self.generator = generator

        def forward(self, src, tgt, src_mask, tgt_mask):
            "Take in and process masked src and target sequences."
            return self.decode(self.encode(src, src_mask), src_mask,
                               tgt, tgt_mask)

        def encode(self, src, src_mask):
            return self.encoder(self.src_embed(src), src_mask)

        def decode(self, memory, src_mask, tgt, tgt_mask):
            return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)

class Generator(nn.Module):
        "Define standard linear + softmax generation step."
        def __init__(self, d_model, vocab):
            super(Generator, self).__init__()
            self.proj = nn.Linear(d_model, vocab)

        def forward(self, x):
            return F.log_softmax(self.proj(x), dim=-1)

```

Figure 46: *A typical encoder/decoder architecture From the Annotated Transformer*

We then get the predicted word, just like in Word2Vec. Other than learning context well, Transformers are also able to parallelize their training by passing multiple tokens to the encoder at the same time, computing states for all positions in the input text.

4.3 BERT

After the success of "Attention is All you Need", a variety of transformer architectures arose. **BERT** stands for Bi-Directional Encoder and was released 2018[13], based on a paper written by Google as a way to solve common natural language tasks like sentiment analysis, question-answering, and text summarization. BERT is a transformer model, also developed at Google, based on the attention mechanism, but its architecture is such that it only includes the encoder piece. Its most prominent usage is in Google Search, where it's the algorithm powering surfacing relevant search results. In the blog post

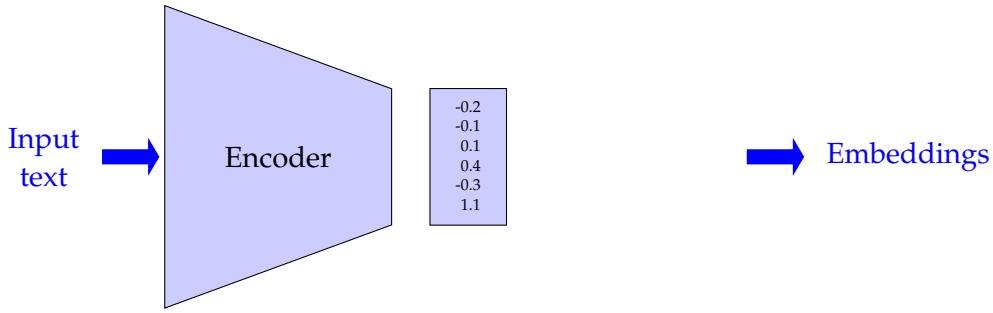


Figure 47: *BERT architecture*

they released on including BERT in search ranking in 2019, Google specifically discussed adding context to queries as a replacement for keyword-based methods as a reason they did this.[29]

BERT works as a **masked language model**. Masking is simply what we did when we implemented Word2Vec by removing words and building our context window. When we created our representations with Word2Vec, we only looked at sliding windows moving forward. The B in Bert is for bi-directional, which means it pays attention to words in both ways through scaled dot-product attention. BERT has 12 transformer layers. It starts by using **WordPiece**, an algorithm that segments words into subwords, into tokens. To train BERT, the goal is to predict a token given its context.

The output of BERT is latent representations of words and their context — a set of embeddings. BERT is, essentially, an enormous parallelized Word2Vec that remembers longer context windows.

Given how flexible BERT is, it can be used for a number of tasks, from translation, to summarization, to autocomplete. Because it doesn't have a decoder component, it can't generate text, which paved the way for GPT models to pick up where BERT left off.

4.4 GPT

BERT is now an industry-standard modeling approach. Around the same time that BERT was being developed, another transformer architecture, the GPT series, was being developed at OpenAI. GPT differs from BERT in that it encodes as well as decodes text from embeddings and therefore can be used for probabilistic inference.

The original, first GPT model was trained as a 12-layer, 12-headed transformer with only a decoder piece, based on data from Book Corpus. Subsequent versions built on this foundation to try and improve context understanding. The largest breakthrough was in GPT-4, which was trained with reinforcement learning from Human Feedback, a property which allows it to make inferences from text that feels much closer to what a human would write.

We've now reached the forefront of what's possible with embeddings in this paper. With the rise of generative methods and methods based on Reinforcement Learning with Human Feedback like OpenAI's ChatGPT and

[29] BERT search announcement

Microsoft's Sydney, as well as the nascent open-source Llama, Alpaca, and other models, anything written in this paper would already be impossibly out of date by the time it was published[30].

5 Embeddings in Production

With the advent of Transformer models, and more importantly, BERT, generating representations of large, multimodal objects for use in all sorts of machine learning tasks suddenly became much easier, the representations became more accurate, and if the company had GPUs available, computations could now be computed with speed-up in parallel.

We've gone through the process of training embeddings end to end here, but there are several modalities for working with embeddings. We can:

- **Train our own embeddings model** - We can train BERT or some variation of BERT from scratch. BERT uses an enormous amount of training data, so this is not really advantageous to us, unless we want to better understand the internals and have access to a lot of GPUS
- **Use pretrained embeddings and fine-tune** - There are many variations on BERT models and they all Variations of BERT have been used to generate embeddings to use as downstream input into many recommender and information retrieval systems. One of the largest gifts that the transformer architecture gives us is the ability to perform transfer learning. Before, when we learned embeddings, our representation of whatever dataset we had at hand was fixed - we couldn't change the weights of the words in TF-IDF without regenerating an entire dataset. There are numerous more compact models for BERT, such as Distilbert and RoBERTA and for many of the larger models in places like the HuggingFace model Hub.[31]. There are many variations on BERT models and they all Variations of BERT have been used to generate embeddings to use as downstream input into many recommender and information retrieval systems. One of the largest gifts that the transformer architecture gives us is the ability to perform transfer learning. Before, when we learned embeddings, our representation of whatever dataset we had at hand was fixed - we couldn't change the weights of the words in TF-IDF without regenerating an entire dataset.

Now, we have the ability to treat the output of the layers of BERT as input into the next layer of our own data that we have fine-tuned. We take our initial data, again tokenize it using the same tokenizer that our BERT model already has, and package the tokenized input into a layer that we add to the model. We remove the final layer of BERT, which is a classifier, and freeze all the weights for all the layers except our last one and then we update the final weights.

[30]There are already some studies about possible uses of LLMs for recommendations, including conversational recommender systems, but it's still very early days. For more information check out this post

[31]For a great writeup on the development of open-source machine learning deep learning, see "A Call to Build Models Like We Build Open-Source Software"

5.1 Three Real Use-Cases

Now that we understand what embeddings are, what can we do with them from a machine learning perspective? Given the flexibility of embeddings as data structures that can be used anywhere, several key uses came from them:

- **Feeding them into another model** - For example, we can now perform collaborative filtering using both user and item embeddings that were learned from our data instead of coding the users and items themselves.
- **Using them directly** - We can use item embeddings directly for content filtering - finding items that are closest to other items, a task recommendation shares with search. There are a host of algorithms used to perform vector similarity lookups by projecting items into our embedding space and performing similarity search using algorithms like Faiss and HNSWLib.

Many companies are working with embeddings in all of these contexts today, across areas that span search, recommendation, and everything in between. There are multiple examples of deep learning embeddings being used in recommendation tasks: a wide and deep model for App Store recommendations at Google Play[70], dual embeddings for product complementary content recommendations at Overstock[37], personalization of search results at Airbnb via real-time ranking[24], using embeddings for content understanding at Netflix[16], for understanding visual styles at Shutterstock[23], and many other examples.

In one popular case, Pinterest is a perfect example of another app that has a large variety of content that, from a business perspective, needs to be personalized and recommended for users. The scale of content created at Pinterest - 350 million monthlu users and 2 billion items (pins) necessitates a considered filtering and ranking policy.

The system they developed to deal with part of this problem is called PinnerSage, a two-stage embedding system that finds semantically similar Pins[49]. The foundation of PinnerSage is PinSage[69]. PinSage is a graph convolutional network algorithm used to generate embeddings that carry latent features explaining graph-level relationships and are constructed via a random walk over the graph. The embeddings are in 256 dimensions and retrieved at inference time from an Approximate Nearest Neighbors Index using HNSW.

Let's take a look at some more specific ones.

5.1.1 YouTube

YouTube was one of the first companies to release a paper on embeddings used in the context of a production recommender system with "Deep Neural Networks for YouTube Recommendations."[11]. YouTube has over 800 million pieces of content (videos) and 2.6 billion active users as of 2023 that they'd like to recommend those videos to. The application needs to be able to recommend content to users, quickly, while also generalizing the recommender model to

new content, and to blend new content with already-established videos. They need to be able to serve these recommendations at **inference time** — when the user loads a new page — with low latency.

In this paper, we create a two-stage recommender (in Tensorflow) consisting of a **candidate generator** that reduces the candidate set to hundreds of items and a second model, similar in size and shape, that's a **ranker** that ranks these videos by probability that the user will click on them.

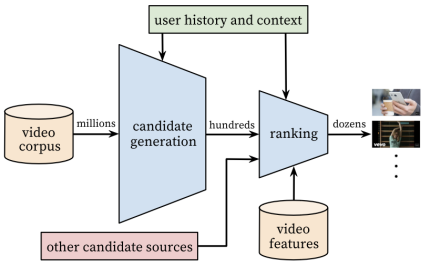


Figure 48: *YouTube's recommender system, including a candidate generator and ranker[11]*

With respect to the machine learning task, we'd like to ultimately predict, given a user, U and **context** C[32] the probability that the user is likely to click on a video at a specific time, based on two sets of embeddings: the user and the context, and a set of video items.

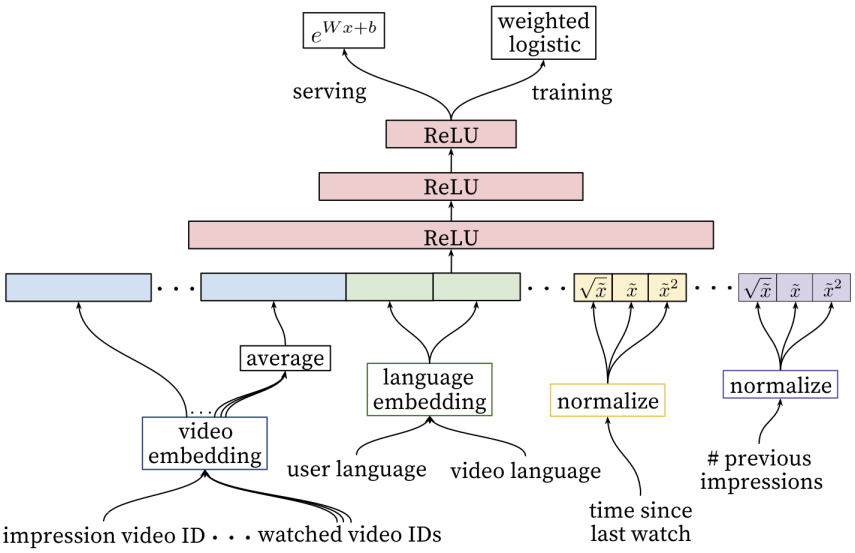


Figure 49: *YouTube's multi-step neural network model for video recommendations using input embeddings[11]*

Implicit feedback - whether a video was viewed or not - is used as part of the response variable. The response variable is then a user plus time and

[32] In recommender systems, we often think of four relevant items to formulate our recommender problem - user, item, context, and query. The context is usually the environment, for example the time of day or the geography of the user at inference time

whether the video was viewed or not. Each class response, of which there are approximately a million, is given a probability as output. The model has several hundreds of features, both tabular and embeddings-based. For the embeddings-based features, we include elements like:

- User watch history - represented by a vector of sparse video ID elements mapped into a dense vector representation
- User's search history - Maps search term to video clicked from the search term, also in a sparse vector mapped into the same space as the user watch history
- User's geography, age, and gender - mapped as tabular features
- The number of previous impressions a video had, normalized per user over time

The size of the vector embedding vocabulary is 1 million classes. The model, made of up of several hidden **ReLu** layers, is trained until convergence. Convergence in this case means using the data of videos that were clicked on and comparing them with the hypothetical videos that would be clicked on as the output of this model.

Similar work, although with a different architecture, was done in the App Store in Google Play in a popular paper, "Wide and Deep Learning for Recommender Systems"[7]." This one crosses the search and recommendation space because it relies on returning correct ranked and personalized app recommendations as the result of a search query. The input is clickstream data collected when a user visits the app store.

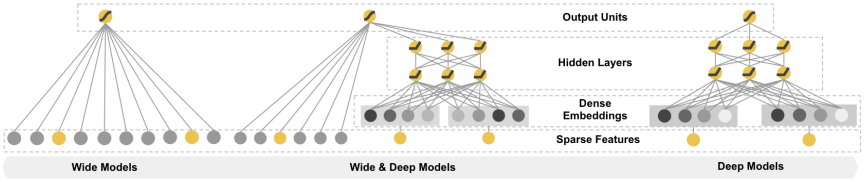


Figure 50: *Wide and deep[7]*

The recommendations problem is formulated here as two jointly-trained models. The weights are shared and cross-propagated between the two models between epochs.

There are two problems when we try to build models that recommend items: **memorization** - the model needs to learn how items occur together given the historical data, and **generalization**, meaning that the model needs to be able to learn and give new recommendations the user has not seen before, improving recommendation diversity. Generally, one model alone cannot encompass both of these tradeoffs, and so side and deep is made up of two models that try to compensate for each other's weaknesses:

- A **wide** model which uses traditional tabular features to improve the model's **memorization**. This is a general linear model trained on sparse, one-hot encoded features like $user_installed_app = netflix$ across thousands of apps. Memorization works here by creating binary features

that are combinations of features, such as $AND(user_installed_app = netflix, impression_app_pandora$, allowing us to see different combinations of co-occurrence in relationship to the target, i.e. likelihood to install app Y. However, this model cannot generalize if it gets a new value outside of the training data.

- A **deep** model that supports **generalization** across items that the model has not seen before, using a feed-forward neural network made up of categorical features that are translated to embeddings, such as user language, device class, and whether a given app has an impression. Each of these embeddings range from 0-100 in dimensionality. They are combined jointly into a concatenated embedding space with dense vectors in 1200 dimensions. and initialized randomly. The embedding values are trained to minimize loss of the final function, which is a logistic loss function common to the deep and wide model.

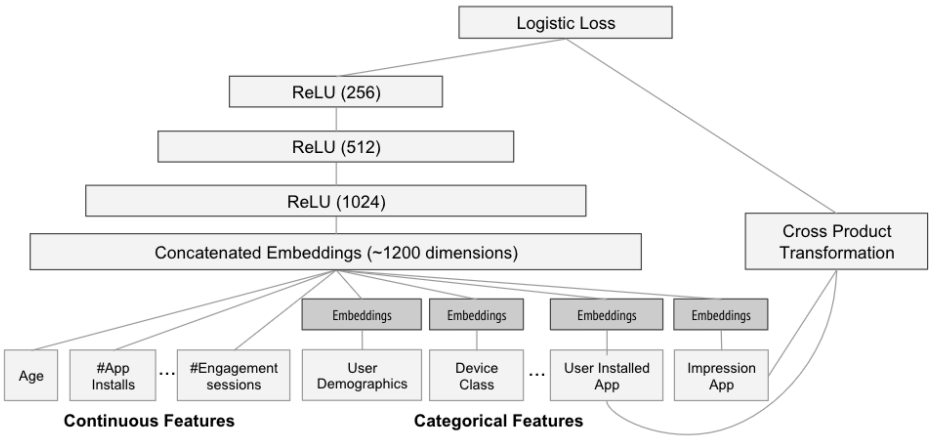


Figure 51: *The deep part of the wide and deep model[7]*

The model is trained on 500 billion examples, and evaluated offline using AUC and online using app acquisition rate, the rate at which people download the app. Based on the paper, using this approach improved the app acquisition rate on the main landing page of the app store by 3.9*percent* relative to the control group.

5.1.2 Twitter

Twitter, our model for Flutter, embeddings were a critical part of tweet recommendation. Twitter generated their own content-based embedding model, Twice (Twitter Content Embeddings)[43] for serving as signals in downstream predictive tasks, such as whether users were likely to click on tweets, which were then used as input signals into how to rank Tweets in the Home Timeline, Notifications, and Topics Models. Twice learns embeddings at the sentence level, capturing a dense representation of sentences and capturing similarity.

When we read the paper, we notice mentions of Word2Vec, particularly the Skipgram model, which Twice Builds on.

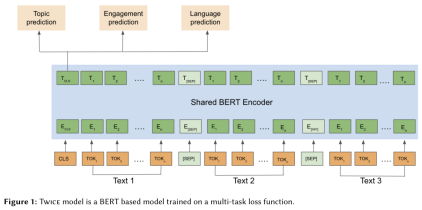


Figure 52: *Twitter's TWICe Embeddings, a trained BERT model[43]*

Twice is a BERT-based model that's trained on a multi-task loss function, including topic prediction (aka the topic associated with a tweet), engagement prediction (the likelihood a user is to engage with a tweet), and language prediction because we'd like for tweets of the same language to be clustered closer together. The model is trained on 200 million tweets sampled over 90 days, for five epochs.

Twitter also developed TWHIN[18], Twitter Heterogeneous Information Network, a set of graph-based embeddings[18], developed for tasks like personalized ads rankings, account follow-recommendation, offensive content detection, and search ranking, based on nodes (such as users and advertisers) and edges that represent entity interactions.

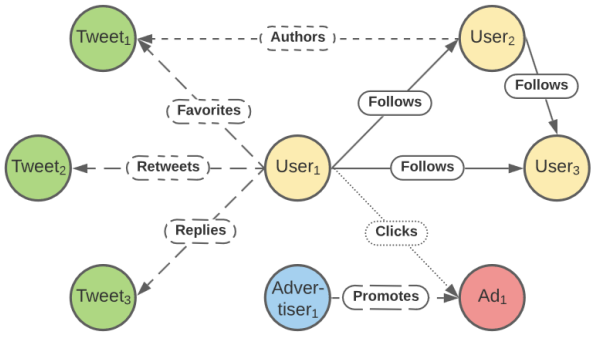


Figure 2: An example heterogeneous information network (HIN) where $|V| = 8$ and $|E| = 9$. There are four entity types ($\mathcal{T}$): 'User', 'Tweet', 'Advertiser', and 'Ad'. There are seven types of relationship ($\mathcal{R}$): 'Follows', 'Authors', 'Favorites', 'Replies', 'Retweets', 'Promotes', and 'Clicks'. See Section 3 for more details.

Figure 53: *Twitter's TWICe Embeddings, a trained BERT model[18]*

Joint embedding is performed by using data from user-tweet engagement, advertising, and following data, to create multi-model embeddings. TWHin is used for candidate generation. The candidate generator starts by finding users to follow or tweets to engage with with an HNSW or Faiss to retrieve candidate items. TWHin embeddings are then used to query candidate items and increase diversity in the candidate pool.

In fact, there are so many uses for embeddings at twitter that the ML platform engineering team created a centralized repository and store for them

5.1.3 Flutter

Given our new knowledge about how embeddings and recommender systems work, we can now incorporate embeddings into the recommendations we serve for flits at Flutter. If we want to recommend relevant content, we might do it in a number of different ways, depending on our business requirements.

We could learn our own embeddings model in training Word2Vec or tf-idf as a baseline model. If that performs well enough in evaluation, we're done. We could learn embeddings and use them in a downstream model that's specific to our use-case - maybe using them in the home timeline. We can use embeddings as inputs into a neural collaborative filtering model[27]. Embeddings are endlessly flexible and endlessly useful, and can empower and improve the performance of our multimodal machine learning workflows.

5.2 Embeddings as an Engineering Problem

Recommender systems generating and retrieving embeddings provide a great deal of utility in working with large, heterogeneous datasets. But this comes at an enormous cost of adding engineering complexity into our existing application. As we can see from several system diagrams, including PinnerSage and the Wide and Deep Model, these systems have many moving components.

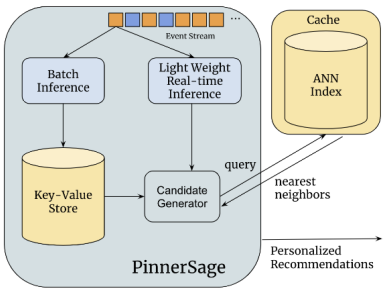


Figure 54: *PinnerSage model architecture[49]*

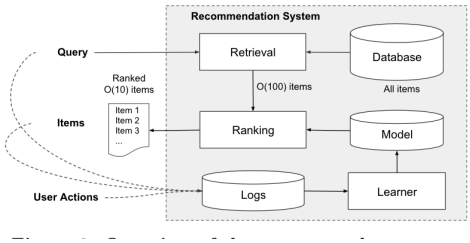


Figure 55: *Wide and Deep model architecture[7]*

In addition to our running business logic code, we are now keeping track of

- Generating embeddings
- Storing embeddings
- Embedding feature engineering and iteration
- Artifact retrieval
- updating embeddings
- versioning embeddings and data drift
- Inference and latency
- Interpreting embeddings and accuracy

5.2.1 Embeddings Generation

We've already seen that embeddings are usually generated as a byproduct of training neural network models, most often the penultimate layer that's used before a layer to classify or regress is added as the final output. We generate them as either the result of training our own models, or by using other, pre-trained models that we can transfer learn from. Considerations here include how many resources we have, both in terms of time and money, to generate our own embeddings, and how accurate pre-trained models are. Something else to consider is whether you want to use pre-trained models or train our own from scratch. Each of these has their own upsides and downsides.

There are pre-trained embeddings available. Pretrained embeddings are based on enormous internet corpuses. There are a lot of pre-trained word embeddings available: Word2Vec via GenSim, GLoVe, and FastText. BERT embeddings are also available. We can also query OpenAI embeddings, although doing so comes at a fairly steep cost.

5.2.2 Storage and Retrieval

Generally when embeddings are generated as part of a Tensorflow (or PyTorch or Gensim) artifact, they initially live in-memory and are then propagated to a model object, which is serialized onto memory and then loaded at re-training or inference time. But if we want to access the embeddings specifically and search through them, as well as potentially do search against other embeddings, we need something that acts as an embeddings store.

There are many requirements to consider when choosing an embeddings store, some high-level ones include:

- Do we access embeddings in batch or are we doing real-time lookup (inference)
- What kind of indexing
- Embedding feature engineering and iteration
- Artifact retrieval
- updating embeddings
- versioning embeddings and data drift
- Inference and latency
- Interpreting embeddings and accuracy

A general-form embeddings store contains the embeddings themselves, an index to map them back from the latent space into words, pictures, or text, and a way to do similarity comparisons between different types of embeddings using algorithms. We talked before about **cosine similarity** being a staple of comparing latent space representations via the normalized dot product of two matrices. This can become computationally expensive over millions of sets of vectors, because we'd need to do a pairwise comparison over every pair. To solve this, **approximate nearest neighbors (ANN)** algorithms were developed to, as in recommender systems, create neighborhoods out of elements of vectors and find a vector's k-nearest neighbors. The most frequently-used

algorithms and libraries tend to be HNSWlib and Faiss, both of which are standalone libraries and also implemented as part of

The tradeoff between this is that it's less precise, but it's much faster[33].

The simplest form of embedding store can be a numpy array[34]. The most complex and customizable is a vector database, and somewhere in-between are plugins for existing relational databases like Postgres, caches like Redis, and SQLite.

Here's an example of the system that Twitter built for exactly this use case, long before vector databases came into the picture. [60]. Given that Twitter uses embeddings in multiple sources as described in the previous section, Twitter made embeddings "first class citizens" by creating a centralized platform that reprocesses data and generates embeddings downstream into the feature registry.

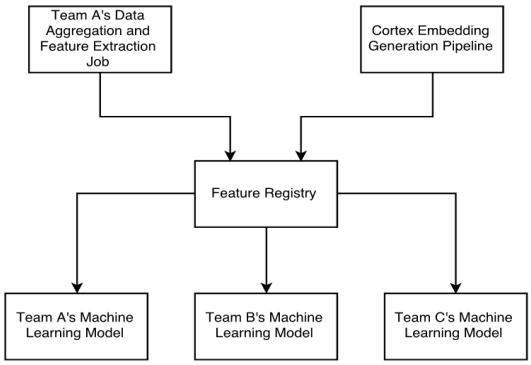


Figure 56: *Twitter's embeddings pipeline[60]*

5.2.3 Drift Detection, Versioning, and Interpretability

Once we've trained embeddings, we might think we're done. But embeddings, like any machine learning pipeline, need to be refreshed because we might run into a concept known as **concept drift**. Concept drift occurs when the data underlying your model changes. For example, let's say that your model includes, as a binary feature, people who have landlines or not. In 2023, this would no longer be as relevant of a feature in most of the world as most people have switched to cell phones as their primary telephones, so the model would lose accuracy. This phenomenon is even more prevalent with embeddings that are used for classification. For example, let's say you use embeddings for trending topic detection. The model has to generalize, as in the wide and deep model, to detect new classes, but if your classes change quickly, it may not be able to, so you need to retrain your embeddings frequently. Or, for example, if you model embeddings in a graph, as Pinterest does, and the relationships

[33] As this post points out, when we switch between precision and recall, you need to be aware of the tradeoffs and think about what our requirements are with respect to how accurate your embeddings are and their latency

[34] From a tweet by Andrej Karpathy in response to how he stores vector embeddings for a small movie recommendations side project in 2023,"np.array people keep reaching for much fancier things way too fast these days. tweet

between nodes in the graph change, you may have to update them[67]. You could also have an influx of spam or corrupted content which changes the relationship in your embeddings, in which case you'll have to retrain.

Once you retrain, you now have a secondary issue: how do you compare the first set of embeddings to the second; that is, how do you evaluate whether they are good representations of your data, given the case that embeddings are often unsupervised; i.e. — how do we know whether "king" should be close to "queen"? On their own, embeddings can be hard to interpret, because in a multidimensional space it's hard to understand which dimension of a vector corresponds to a decision to place items next to each other[61].how we will explain our embedding model to stakeholders. Understanding embeddings. Embeddings can be hard to understand (so hard that some people have even written entire papers about them) and harder to interpret. What does it mean for a king to be close to queen but far away from the knight? What does it mean for two flits to be close to each other in a projected embedding space?

We have two ways we can think about this, **intrinsic** and **extrinsic** evaluation. For the embeddings themselves (extrinsic evaluation), we can visualize them through either **UMAP**—Uniform Manifold Approximation and Projection for Dimension Reduction or **t-sne** — t-distributed stochastic neighbor embedding, algorithms that allow us to visualize highly-dimensional data in two or three dimensions, much like PCA. Or we can fit embeddings into our downstream task (for example, summarization, or classification), and analyze the same way we would with offline metrics. There are many different approaches [65] but the summary is that embeddings can be objectively hard to evaluate and we'll need to factor the time to perform this evaluation into our modeling time.

Finally, now let's say that we have two sets of embeddings, you need to version them and keep both sets so that, in case your new model doesn't work as well, you can fall back to the old one. This goes hand in hand with the problem of model versioning in ML operations, except in this case we need to both version the model and the output data. There are numerous different approaches to model and data versioning that involve building a system that tracks both the metadata and the location of the assets that are kept in a secondary data store. Another thing to keep in mind is that embeddings layers, particularly for large vocabularies, can balloon in size, so now we have to consider storage costs, as well.

5.2.4 Inference and Latency

When working with embeddings, we are operating not only in a theoretical, but practical engineering environment. The most critical part of any machine learning system that works in production is inference time — how quickly does it take to query the model asset and return a result to an end-user.

For this, we care about latency, which we can roughly define as any time spent waiting and is a critical performance metric in any production system[25]. Generally speaking, it's the time of any operation to complete – application request, database query, and so on. Latency at the level of the web service is generally measured in milliseconds, and every effort is made to

reduce this as close to zero as realistically possible. For use-cases like search and loading a feed of content, the experience needs to be instantaneous or the user experience will degrade and we could even lose revenue. In a study from a while back, Amazon found that every 100ms increase in latency cuts profits by 1

Given this, we need to think about how to reduce the footprint of our model and all the layers in serving it so that the response to the user is instantaneous. We do this by creating observability throughout our machine learning system, starting with the hardware the system is running on, to CPU and GPU utilization, the performance of our model architecture, and how that model interacts with other components. For example, when we are performing nearest neighbor lookup, the way we perform that lookup and the algorithm we use, the programming language we use to write that algorithm, all compound latency concerns.

As an example, in the wide and deep paper, the recommender ranking model scores over 10 million apps per second. The application was initially single-threaded, with all candidates taking 31 milliseconds. By implementing multithreading, they were able to reduce client-side latency to 14 milliseconds[7].

Operations of machine learning systems is an entirely other art and craft of study and one that's best left for another paper[36].

5.2.5 What makes embeddings projects successful

Finally, once we have all our algorithmic and engineering concerns lined up, there is the final matter to consider of what will make our project successful from a business perspective. We should acknowledge that we might not always need embeddings for our machine learning problem, or that we might not need machine learning at all, initially, if our project is based entirely on a handful of heuristic rules that can be determined and analyzed by humans[72].

If we conclude that we are operating in a data-rich space where automatically inferring semantic relationships between entities is correct, we need to ask ourselves if we're willing to put in a great deal of effort into producing clean datasets, the baseline of any good machine learning model, even in cases of large language models. In fact, clean, domain data is so important that many of the companies discussed here ended up training their own embeddings models, and, recently companies like Bloomberg[68] and Replit[57] are even training their own large language models to improve accuracy for their specific business domain.

Critically, to get to a stage where we have a machine learning system dealing with embeddings, we need a team that has multilevel alignment around the work that needs to be done. In larger companies, the size of this team will be larger, but most specifically work with embeddings requires someone who can speak to the use case specifically, someone who advocates for the use case and gets it prioritize, and a technical person who can do the work[45].

If all of these components come together, we now have an embeddings-based recommender system in production.

6 Conclusion

We have now walked through an end-to-end example of what it means to generate and need embeddings, from the early approaches, to modern-day approaches using Transformers and BERT. We've understood the business context of why we might include machine learning models in our application, how they work, how to incorporate them, and where embeddings —dense representations of deep learning model input and output data – can be best leveraged.

In summary, embeddings are a wonderful tool in any machine learning system, but one that's not it's not always easy to understand, maintain. Generating embeddings using the correct method, with the correct interpretation, is a project that takes the most important pieces of these being that we need to have clean data, and constantly think about the performance time of our embeddings

Although reducing dimensionality as a concept has always been important in machine learning systems to to decrease computational and storage complexity, compression has become even more important in the modern explosion of multimodal representations of data that comes from application log files, images, video, and audio, and the explosion of Transformer, generative, and diffusion models, combined with the cheap storage and explosion of data, has amended itself to architectures where embeddings are used more and more.

We now hopefully have a solid grasp of the fundamentals of embedding and can either leverage them – or explain why not to –in our next project.

Good luck navigating embeddings, and hope to see you in the latent space.

References

1. Gabriel Altay. Categorical variables in decision trees, Mar 2020. URL https://www.kaggle.com/code/gabrielaltay/categorical-variables-in-decision-trees.

2. Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 103–112, 2014.

3. Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, LLC, 2018.

4. Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

5. Pablo Castells and Dietmar Jannach. Recommender systems: A primer. *arXiv preprint arXiv:2302.02579*, 2023.

6. Dhivya Chandrasekaran and Vijay Mago. Evolution of semantic similarity—a survey. *ACM Computing Surveys (CSUR)*, 54(2):1–37, 2021.

7. Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

8. Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

9. Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.

10. Yingnan Cong, Yao-ban Chan, and Mark A Ragan. A novel alignment-free method for detection of lateral genetic transfer based on tf-idf. *Scientific reports*, 6(1):1–13, 2016.

11. Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

12. Toni Cvitanic, Bumsoo Lee, Hyeon Ik Song, Katherine Fu, and David Rosen. Lda v. lsa: A comparison of two computational text analysis tools for the functional categorization of patents. In *International Conference on Case-Based Reasoning*, 2016.

13. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

14. Giovanni Di Gennaro, Amedeo Buonanno, and Francesco AN Palmieri. Considerations about learning word2vec. *The Journal of Supercomputing*, pages 1–16, 2021.

15. Author Cory Doctorow. Pluralistic: Tiktok's enshittification (21 jan 2023), Feb 2023. URL https://pluralistic.net/2023/01/21/potemkin-ai/#hey-guys.

16. Melody Dye, Chaitanya Ekandham, Avneesh Saluja, and Ashish Rastogi. Supporting content decision makers with machine learning, Dec 2020. URL https://netflixtechblog.com/supporting-content-decision-makers-with-machine-learning-995b7b76006f.

17. Michael D Ekstrand and Joseph A Konstan. Recommender systems notation: proposed common notation for teaching and research. *arXiv preprint arXiv:1902.01348*, 2019.

18. Ahmed El-Kishky, Thomas Markovich, Serim Park, Chetan Verma, Baekjin Kim, Ramy Eskander, Yury Malkov, Frank Portman, Sofía Samaniego, Ying Xiao, et al. Twhin: Embedding the twitter heterogeneous information network for personalized recommendation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2842–2850, 2022.

19. Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

20. Jan J Gerbrands. On the relationships between svd, klt and pca. *Pattern recognition*, 14(1-6):375–381, 1981.

21. David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

22. Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

23. Raul Gomez Bruballa, Lauren Burnham-King, and Alessandra Sala. Learning users' preferred visual styles in an image marketplace. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pages 466–468, 2022.

24. Mihajlo Grbovic and Haibin Cheng. Real-time personalization using embeddings for search ranking at airbnb. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 311–320, 2018.

25. Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

26. Casper Hansen, Christian Hansen, Lucas Maystre, Rishabh Mehrotra, Brian Brost, Federico Tomasi, and Mounia Lalmas. Contextual and sequential user embeddings for large-scale music recommendation. In *Proceedings of the 14th ACM Conference on Recommender Systems*, pages 53–62, 2020.

27. Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

28. Michael E Houle, Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Can shared-neighbor distances defeat the curse of dimensionality? In *Scientific and Statistical Database Management: 22nd International Conference, SSDBM 2010, Heidelberg, Germany, June 30–July 2, 2010. Proceedings 22*, pages 482–500. Springer, 2010.

29. Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender systems: an introduction*. Cambridge University Press, 2010.

30. Yushi Jing, David Liu, Dmitry Kislyuk, Andrew Zhai, Jiajing Xu, Jeff Donahue, and Sarah Tavel. Visual search at pinterest. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1889–1898, 2015.

31. Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Machine Learning: ECML-98: 10th European Conference on Machine Learning Chemnitz, Germany, April 21–23, 1998 Proceedings*, pages 137–142. Springer, 2005.

32. Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, May 2015. URL https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

33. P.N. Klein. *Coding the Matrix: Linear Algebra Through Applications to Computer Science*. Newtonian Press, 2013. ISBN 9780615880990. URL https://books.google.com/books?id=3AA4nwEACAAJ.

34. Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

35. Jay Kreps. *I heart logs: Event data, stream processing, and data integration*. " O'Reilly Media, Inc.", 2014.

36. Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. *arXiv preprint arXiv:2205.02302*, 2022.

37. Giorgi Kvernadze, Putu Ayu G Sudyanti, Nishan Subedi, and Mohammad Hajiaghayi. Two is better than one: Dual embeddings for complementary product recommendations. *arXiv preprint arXiv:2211.14982*, 2022.

38. Valliappa Lakshmanan, Sara Robinson, and Michael Munn. *Machine learning design patterns*. O'Reilly Media, 2020.

39. Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

40. Yann LeCun, Yoshua Bengio, Geoffrey Hinton, et al. Deep learning. nature, 521 (7553), 436-444. *Google Scholar Google Scholar Cross Ref Cross Ref*, page 25, 2015.

41. Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive data sets*. Cambridge university press, 2020.

42. Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. *Advances in neural information processing systems*, 27, 2014.

43. Xianjing Liu, Behzad Golshan, Kenny Leung, Aman Saini, Vivek Kulkarni, Ali Mollahosseini, and Jeff Mo. Twice-twitter content embeddings. In *CIKM 2022*, 2022.

44. Donella H Meadows. *Thinking in systems: A primer*. chelsea green publishing, 2008.

45. Doug Meil. Ai in the enterprise. *Communications of the ACM*, 66(6):6–7, 2023.

46. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

47. Usman Naseem, Imran Razzak, Shah Khalid Khan, and Mukesh Prasad. A comprehensive survey on word representation models: From classical to state-of-the-art word representation language models. *Transactions on Asian and Low-Resource Language Information Processing*, 20(5):1–35, 2021.

48. Bogdan Oancea, Tudorel Andrei, and Raluca Mariana Dragoescu. Gpgpu computing. *arXiv preprint arXiv:1408.6923*, 2014.

49. Aditya Pal, Chantat Eksombatchai, Yitong Zhou, Bo Zhao, Charles Rosenberg, and Jure Leskovec. Pinnersage: Multi-modal user embedding framework for recommendations at pinterest. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2311–2320, 2020.

50. Delip Rao and Brian McMahan. *Natural language processing with PyTorch: build intelligent language applications using deep learning*. " O'Reilly Media, Inc.", 2019.

51. David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

52. Alexander M Rush. The annotated transformer. In *Proceedings of workshop for NLP open source software (NLP-OSS)*, pages 52–60, 2018.

53. Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

54. Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

55. David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt.(2014), 2014.

56. Nick Seaver. *Computing Taste: Algorithms and the Makers of Music Recommendation*. University of Chicago Press, 2022.

57. Reza Shabani. How to train your own large language models, Apr 2023. URL https://blog.replit.com/llm-training.

58. Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

59. Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training nlp models: A concise overview. *arXiv preprint arXiv:2004.08900*, 2020.

60. Dan Shiebler and Abhishek Tayal. Making machine learning easy with embeddings. *SysML http://www.sysml.cc/doc/115.pdf*, 2010.

61. Adi Simhi and Shaul Markovitch. Interpreting embedding spaces by conceptualization. *arXiv preprint arXiv:2209.00445*, 2022.

62. Harald Steck, Linas Baltrunas, Ehtsham Elahi, Dawen Liang, Yves Raimond, and Justin Basilico. Deep learning for recommender systems: A netflix case study. *AI Magazine*, 42(3):7–18, 2021.

63. Krysta M Svore and Christopher JC Burges. A machine learning approach for improved bm25 retrieval. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1811–1814, 2009.

64. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

65. Bin Wang, Angela Wang, Fenxiao Chen, Yuncheng Wang, and C-C Jay Kuo. Evaluating word embedding models: Methods and experimental results. *APSIPA transactions on signal and information processing*, 8:e19, 2019.

66. Yuxuan Wang, Yutai Hou, Wanxiang Che, and Ting Liu. From static to dynamic word representations: a survey. *International Journal of Machine Learning and Cybernetics*, 11:1611–1630, 2020.

67. Christopher Wewer, Florian Lemmerich, and Michael Cochez. Updating embeddings for dynamic knowledge graphs. *arXiv preprint arXiv:2109.10896*, 2021.

68. Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann. Bloomberggpt: A large language model for finance. *arXiv preprint arXiv:2303.17564*, 2023.

69. Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

70. Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM computing surveys (CSUR)*, 52(1):1–38, 2019.

71. Yong Zheng. Multi-stakeholder recommendation: Applications and challenges. *arXiv preprint arXiv:1707.08913*, 2017.

72. Martin Zinkevich. Rules of machine learning: Best practices for ml engineering. *URL: https://developers. google. com/machine-learning/guides/rules-of-ml*, 2017.