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

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, 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as the nascent open-source Llama, Alpaca, and other models, anything written in this paper would already be impossibly out of date by the time it was published[30].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. Now that we understand what embeddings are, what should we do with them? After all, we're not

[29] BERT search announcement

[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

doing this just as a math exercise. If there is one thing to take away from this entire text, it is this:

> The final goal of all industrial machine learning (ML) projects is to develop ML products and rapidly bring them into production. [37]

The model that is deployed is always better and more accurate than the model that is only ever a prototype. 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 in pre-transformer architectures, 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 neural network layer of our own, custom model. In addition to transfer learning, there are also 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].

Armed with this knowledge, we can think of several use cases of embeddings, given their flexibility as a data structure.

- **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 HNSW.

[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 Embeddings in Practice

Many companies are working with embeddings in all of these contexts today, across areas that span all aspects of information retrieval. Embeddings generated with deep learning models are being generated for use in wide and deep models for App Store recommendations at Google Play [73], dual embeddings for product complementary content recommendations at Overstock [38], personalization of search results at Airbnb via real-time ranking [25], using embeddings for content understanding at Netflix [16], for understanding visual styles at Shutterstock [24], and many other examples.

5.1.1 Pinterest

One notable example is Pinterest. Pinterest as an application has a wide variety of content that needs to be personalized and classified for recommendation to users across multiple surfaces, particularly the Homefeed and shopping tab. The scale of generated content - 350 million monthly users and 2 billion items - Pins — or cards with an image described by text — necessitates a strong filtering and ranking policy.

To represent a user's interest and surface interesting content, Pinterest developed PinnerSage [50], which represents user interests through multiple 256-dimension embeddings that are clustered based on similarity and represented by **medioids** — an item that is a representative of a center of a given interest cluster.

The foundation of this system is a set of embeddings developed through an algorithm called PinSage [72]. Pinsage generates embeddings using a Graph Convolutional neural network, which is a neural net that takes into account the graph structure of relationships between nodes in the network. The algorithm looks at the nearest neighbors of a pin and samples from nearby pins based on related neighborhood visits. The input is embeddings of a Pin: the image embeddings, and the text embeddings, and finds the nearest neighbors.

Pinsage embeddings are then passed to Pinnersage, which takes the pins the user has acted on for the past 90 days and clusters them. It computes the medioid and takes the top 3 medioids based on importance, and, given a user query that is a medioid, performs an approximate nearest neighbors search using HNSW to find the pins closest to the query in the embedding space.

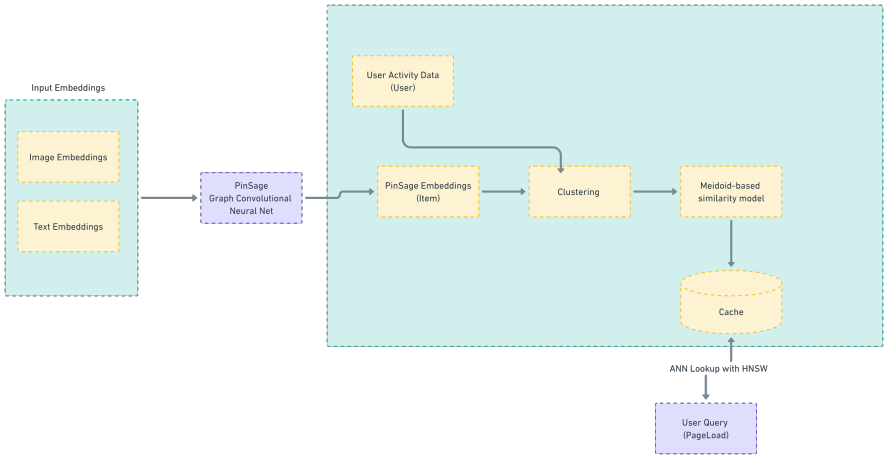

Figure 46: *Pinnersage and Pinsage embeddings-based similarity retrieval*

5.1.2 YouTube and Google Play Store

YouTube

YouTube was one of the first large companies to publicly share their work on embeddings used in the context of a production recommender system with "Deep Neural Networks for YouTube Recommendations."

YouTube has over 800 million pieces of content (videos) and 2.6 billion active users that they'd like to recommend those videos to. The application needs to recommend existing content to users, while also generalizing to new content, which is uploaded frequently. 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 [11], YouTube shares how they created a two-stage recommender system for videos based on two deep learning models. The machine learning task is to predict the correct next video to show the user at a given time in YouTube recommendations so that they click. The final output is formulated as a classification problem: given a user's input features and the input features of a video, can we predict a class for the user that includes the predicted watch time for the user for a specific video with a specific probability.

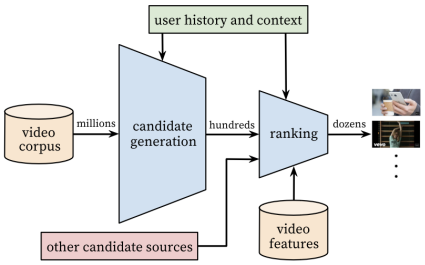

Figure 47: *YouTube's end-to-end video recommender system, including a candidate generator and ranker [11]*

We set this task given a user, U and **context** C[32]

Given the size of the input corpus, we need to formulate the problem as a two-stage recommender: the first is the **candidate generator** that reduces the candidate video set to hundreds of items and a second model, similar in size and shape, called a **ranker** that ranks these hundreds of videos by the probability that the user will click on them and watch.

The candidate generator is a **softmax** deep learning model with several layers, all activated with **ReLU** activation functions – rectified linear unit activation that outputs the input directly if positive; otherwise, it's zero. The uses both embedded and tabular learning features, all of which are combined and

To build the model, we use two sets of embeddings as input data: one that's the user plus context as features, and a set of video items. 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

These are all combined into a single item embedding, and in the case of the user, a single embedding that's a blended map of all the user embedding features, and fed into the models' softmax layers, which compare the distance between the output of the softmax layer, i.e. the probability that the user will click on an item, and a set of ground truth items, i.e. a set of items that the user has already interacted with. The log probability of an item is the dot product of two n-dimensional vectors, i.e. the query and item embeddings.

We consider this an example of **Implicit** feedback - feedback the user did not explicitly give, such as a rating, but that we can capture in our log data. Each class response, of which there are approximately a million, is given a probability as output.

The DNN is a generalization of the matrix factorization model we discussed earlier.

[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

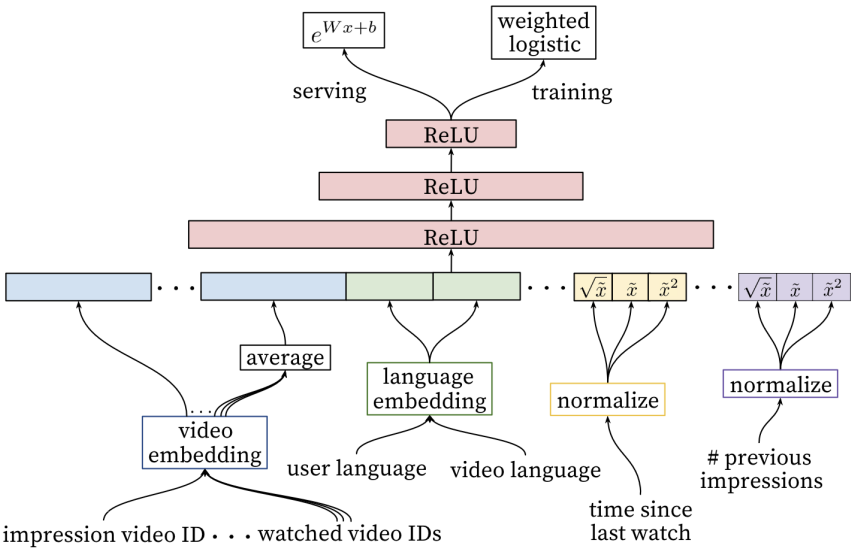

Figure 48: *YouTube's multi-step neural network model for video recommendations using input embeddings [11]*

Google Play App Store

Similar work, although with a different architecture, was done in the App Store in Google Play in "Wide and Deep Learning for Recommender Systems" [7]. This one crosses the search and recommendation space because it returns 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.

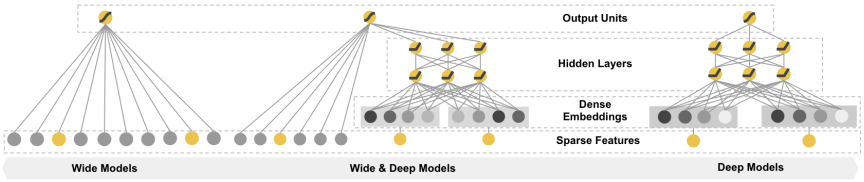

Figure 49: *Wide and deep [7]*

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 patterns by learning how items occur together given the historical data, and **generalization** - the model needs to be able to give new recommendations the user has not seen before that are still relevant to the user, improving recommendation diversity. Generally, one model alone cannot encompass both of these tradeoffs.

Wide and deep is made up of two models that look to complement each other:

- 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.

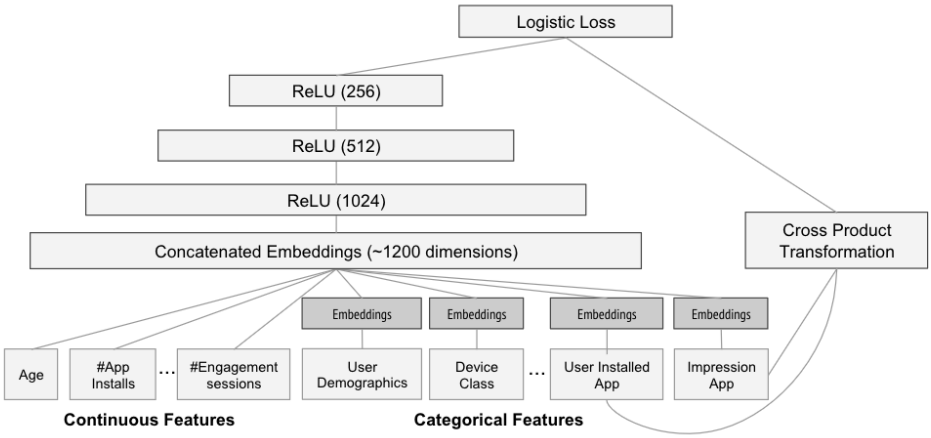

Figure 50: *The deep part of the wide and deep model [7]*

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 % relative to the control group.

5.1.3 Twitter

At **Twitter**, pre-computed embeddings were a critical part recommendations for many app surface areas including user onboarding topic interest prediction, recommended Tweets, home timeline construction, users to follow, and

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

27. 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.

28. 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.

29. 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.

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

31. 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.

32. 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.

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

34. 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`.

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

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

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

38. 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.

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

40. 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.

41. 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.

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

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

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

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

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

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

48. 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.

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

50. 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.

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

52. Steffen Rendle, Walid Krichene, Li Zhang, and John Anderson. Neural collaborative filtering vs. matrix factorization revisited. In *Proceedings of the 14th ACM Conference on Recommender Systems*, pages 240–248, 2020.

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

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

55. 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.

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

57. 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.

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

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

60. 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.

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

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

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

64. 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.

65. 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.

66. 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.