Module 23: Generative AI

Video Transcripts

Video 23.1: Introduction

In February 2019, Philip Wang created ThisPersonDoesNotExist.com. Every time the page was refreshed, a new realistic human face was generated.

At the time, this technology was largely unknown outside the research community. As James Vincent wrote for The Verge at the time, “The ability of AI to generate fake visuals is not yet mainstream knowledge, but a new website—ThisPersonDoesNotExist.com—offers a quick and persuasive education… the ability to manipulate and generate realistic imagery at scale is going to have a huge effect on how modern societies think about evidence and trust. Such software could also be extremely useful for creating political propaganda and influence campaigns. In other words, ThisPersonDoesNotExist.com is just the polite introduction to this new technology. The rude awakening comes later.”

Now, many years later, this technology has evolved to the point where you can train a model to generate pictures of your own face, then provide text prompts like “Josh Hug riding a Unicorn”, and get back an image like the one shown.

These technologies are examples of what are called “Generative AI.” Generative AI is a broad domain of technologies which use artificial intelligence to generate useful output.

The most basic models simply draw random samples from some learned distributions, such as ThisPersonDoesNotExist, which tries to learn the distribution of human faces.

More complex models allow you to direct or condition the output by providing some input. For example, there are models which take in text and then they generate a picture of that text. Or models that take in text and generate audio that mimics a specific person's voice saying that text. Or models which take spoken audio and then generate text transcripts.

The pace of progress has been rapid, and investors have poured vast sums of money speculating on a possible bonanza whose shape and scale is impossible to predict. As the Wall Street Journal reported in 2024, “[In 2023] investors poured $21.8 billion into generative AI deals, up fivefold from the prior year, according to the research firm CB Insights... Venture capitalists are betting that some of these startups will be the pioneers in a tech revolution that could outshine even the birth of the Internet. They point to the meteoric rise of OpenAI, whose chatbot ChatGPT became the fastest-growing consumer app in Internet history. OpenAI went from zero to more than $1 billion in revenue last year, a brisk growth rate even by the breakneck standards of Silicon Valley.”

So today, we'll talk about how some of these technologies work. The field is quite large, so we'll cover only a small subset of generative AI topics, and we'll only do so at a high level. We'll provide references where appropriate for those of you who want to explore more deeply. Let's dig in.

Video 23.2: Generative Adversarial Networks (GANs)

In the image shown, we see several faces that were generated using ThisPersonDoesNotExist.com, which we discussed earlier. These faces are

the output of a neural network called a “generator” that takes as input a vector of random noise and produces as output a realistic image of a human face.

We've already seen how we can train a model to classify images as being a face, for example, using this convolutional neural network that we saw in an earlier module. The output of this network is simple, it's a vector of probabilities that the image belongs to each known class.

By contrast, an image-generating network must output hundreds of thousands or even millions of numerical values representing red, green, and blue pixel values, and all of these must be connected together in a harmonious way to resemble a human face. And that image needs to be believable to us humans who were trained over millions of years of evolution to recognize the faces exquisitely well.

As I mentioned before, Philip Wang released ThisPersonDoesNotExist.com in February 2019. The underlying technology was not his own invention. Instead, Nvidia had just released an open-source model called StyleGAN that was capable of generating faces. His website was actually just a thin wrapper around Nvidia's technology, bringing its capabilities to a bigger audience.

The GAN and StyleGAN stands for “Generative Adversarial Network.” The big idea is that we build two neural networks that are enemies of each other.

The first network is called the discriminator. It takes as input an image and it outputs the probability that the image is of a face. That's exactly like the image classifiers that we discussed in earlier modules.

The second neural network is called the generator. It takes as input random noise and it outputs a grid of red, green, and blue values that are supposed to be a face. In principle, this network's architecture could be anything. It could be a dense neural network, something with convolutional layers, etcetera. But we won't talk about any specific architecture for the generator, since that's not the main idea here.

GANs rely on a clever training procedure where the two networks are locked into a competitive game where they're trying to beat each other. The generator will try to produce realistic human faces, and the discriminator will try to tell which faces are real and which ones are fake. If we can train them both to a high level of skill, then we'll hopefully end up with the generator that can fool even humans.

For our example, I'll use as my real faces the "CelebA faces" dataset, a commonly used library of faces created by the Chinese University of Hong Kong.

During each round of training, we first select N faces from our real face dataset, and then we use our generator to generate a complementary set of N fake faces.

When the process starts, the generator will be terrible at its job. Here, you can see the five faces that I got back when I ran my own GAN training experiment for this video. You can see it's just noise because the generator has not yet learned anything.

We then provide this mix of real and fake faces as a labeled batch to the discriminator, where it tries to learn to differentiate the real images from the fake images. The discriminator will just use some standard loss function for classification, like cross-entropy loss.

Using this mixed batch of fake and real faces, the discriminator takes only a single step of stochastic gradient descent. As these two input types are so different, it'll learn very quickly. In my experiment, it got its accuracy up to 66% in this single SGD step.

After the discriminator's brief moment of learning, we let the generator have its turn to learn. The clever idea is that we use the discriminator's performance as its loss function. That is, if x is an image generated by the generator, and D(x) is the probability that the discriminator thinks it is real, we can use something like the negative log of D(x) as the loss.

The generator gets to learn for a single SGD step and gets slightly better at generating images.

By alternating back and forth between the discriminator and the generator, both of them slowly get better and better.

For example, after 100 steps of stochastic gradient descent, my generator was able to output images like the one shown. These still aren't faces, but they're starting to show some kind of subtle structure.

Even more training yields an even more skillful generator. After running the training process on an A-100 GPU for around 30 minutes, I ended up with a pretty good generator. Here are five images that it was capable of generating when I stopped the training process.

You'll note these images are low resolution. That's because the model I trained was only outputting 64×64-pixel images. Turns out that in practice, image generation is often done by first generating a small image like this, then training additional networks to be able to convincingly upscale the

images. Discussion of this upscaling is beyond the scope of our discussion today.

While GANs are capable of generating impressive outputs, training again is notoriously tricky. One failure mode is for the discriminator to get too good at its job too quickly. As a thought experiment, imagine that the discriminator correctly feels 100% certain that the generator's images are fake. In that case, the loss would be the negative log of zero, that is, Infinity. Since the loss is totally saturated, there are no gradients for the learning algorithm to use to improve. This is often known as the “vanishing gradient problem.”

We can catch a glimpse of this potential problem if we look at the logs of the losses of the generator and discriminator as I ran my experiment. At the start of the experiment, the discriminator very rapidly learns to beat the untrained generator, resulting in very low loss for the discriminator and very high loss for the generator. Here, the gradients are no doubt rather small. However, around batch number 6000, despite the small gradients, the generator manages to innovate in some way that keeps the process going and the networks battle it out until we stop the training.

However, if the generator had not gotten lucky with its innovation, we would have ended up with a bad generator whose outputs were barely better than noise.

Another infamous failure mode is that the generator gets stuck in a creative rut, for example, only generating pictures of children. This is known as “mode collapse” since the generator is missing some modes of the distribution from which it is trying to generate samples.

Various tricks exist for these common failure modes, such as adding noise to the input seen by the discriminator, though GAN training remains a significant challenge.

Video 23.3: Diffusion

An alternate technology for generating images is known as diffusion. Unlike GANs, diffusion models are generally much easier to train, though, as we'll see, the downside is that sample generation is typically much slower. As before, I'm focusing here only on the big ideas, and I'm emitting a lot of important details that would be necessary to train a real diffusion model from scratch.

As an introductory thought experiment, imagine the following scenario: First, we create a large collection of N real images from some distribution of interest. We then add a small amount of Gaussian noise to every image, where the noise added to each image is different. You can see here that the noise is barely perceptible.

Now, suppose we train a neural network to denoise these images. That is, we provide pairs of images as a supervised learning dataset, the inputs of the noisy images and the outputs of the original noiseless images. One obvious use for this network is that we could use it to denoise images that were slightly noisy.

However, we have bigger ambitions. Imagine that instead of just adding noise one time, we add noise over and over, creating a series of increasingly noisy images. Eventually, the noise will dominate the original signal and will be left with an image which is completely indistinguishable from Gaussian noise. Let's give a name to these increasingly noisy images as follows. Level 0 images are the original non-noisy images. Level 1 images are the

images after the first noise is added. Level 2 images are after two rounds of noise are added all the way until we reach Level T images after T rounds of noise are added.

Now imagine that we train T different neural networks, where the first neural network takes Level 1 images and learns to output level images, where the next neural network takes Level 2 images and learns to output Level 1 images and so forth.

If we were successful in training such networks, we'd then be able to do the following. Provide an image of random noise, then feed it through the Level T network, then feed the output of that network into the Level T – 1 network and so forth until we get to the Level 0 network, which would produce an image from our distribution.

This is the basic idea behind diffusion, though a number of improvements to this basic idea are used in real life to make the process more efficient. For example, as with GANs, the denoising is done with lower-resolution images, which are then upscaled. Another improvement is that instead of building T-distinct networks, we instead build a single network that takes in two inputs: a noisy image and a parameter that indicates what level of noise was applied to the image. These improvements are beyond the scope of our discussion.

In practice, this model works amazingly well, and at the time of the creation of this video, diffusion models are generally more popular than GANs, though, as I noted earlier, the time it takes to generate images using diffusion is typically much longer than a GAN.

Video 23.4: Live Diffusion Demo

Behold this image. I generated this using an image generator called Flux, and the specific prompt I used was “Josh Hug showing off his cool new T-shirt at the beach. A shark is jumping out of the water in the back.” Now, there's a couple of problems with this image.

One of them is there's no shark jumping out of the water, and that's typical for these kind of models, which may ignore your prompt to some degree. The other problem is that this is not me. The model just doesn't know who Josh Hug is.

So, these models like Flux and Stable Diffusion are great at generating images of things they've seen but not so much at things they have not seen.

In this video, I'll show you how to fix that. So, in particular, what I'm going to do is provide a number of prompt image pairs, and in this case, all of the prompts say simply “a photo of Josh Hug”, and then the images show some picture of me. Not the most flattering, but hey, that's me.

So, you give it some pictures of yourself in various different guises, angles, lighting conditions, and it will learn how to imitate you. Now, it doesn't have to be a particular person or object or whatever to learn. It could also be something like “a painting made by Josh Hug”, and then it'll learn to imitate your style. So, whatever it is you want it to learn, it can learn it.

So, let's see what happens. So, what I've done is I've provided a notebook that you can run if you're using Google Colab that will allow you to train a model.

When you run one of these model training scripts, you end up with a model that's learned something about you. It's fine-tuned. Now, I'm using Google

Colab. You don't have to, but one important point is that you need a powerful GPU in order to fine-tune these models. I would say that something around 24 gigabytes of VRAM is a what's necessary.

So, that would be something like the Google Colab is a bunch of different choices. I used an A100. The A100 is pretty expensive to rent or own. So, now that I've clicked that button, I'm spending about a dollar to $2 an hour when I use this A100 and the training process, if you run it takes, in this particular case, it took around 28 minutes to train this model for 1000 iterations. So, you can use whatever vendor you want. But I've provided a Google Colab example that should be relatively easy to adapt to other sources. But I find that Google Colab is one of the more user friendly to use.

So, after I've trained this model and let it run for 28 minutes, which I won't do here for obvious reasons, then I can look and see what the models learned. So, one of the things that this model outputs is a series of images demonstrating what it learned as it went. I have a bunch of prompts here that are largely defaults that it provides, though I added some of my own where I just swapped in my name, and you can see how well the model is able to learn to generate a picture of me in each of these scenarios.

So, let's pop back over to those images. This is the original untrained version of the Flux diffusion model, but after 200 training iterations, it produces this different Josh Hug. Same shirt. You'll notice. Now, this Josh Hug definitely looks more like me, but much more Rodenty, and I think, pretty far from what I look like. Iteration 400. Arguably kind of better, though I do recognize my eye. Once I get to iteration 600, it's doing better, 800 even better though, sort of broy, and then by iteration about 1000, it's looking pretty good, and I could keep going, but I would believe that this is an image of myself looking.

I have other examples that I produced. So, for example, here's me DJing with a martini, with laser light in the background, and again, not my prompt. It's just built-in, and we could see that as it goes, it gets better and better at generating me. Though this last one at iteration 1000, I think it actually regresses a bit. So, it's not a monotonic process of necessarily getting better, and of course, the outputs are random, so maybe a slightly different random feed; iteration 1000 would look even better.

Here's another example of me generating images of myself. So, this is Josh Hug demonstrating how to fine-tune an image generation model, and we see that as we increase the number of iterations, it produces increasingly good pictures of me with a funny jump at 1000, where suddenly I'm quite handsome. Look at that guy. It's like I could be in Bollywood or something.

So, you'll find that these Stable Diffusion models or Flux models or whatever else that they're quite good at learning to generate pictures of yourself.

If you provided examples or pictures of other pieces of art, and I've plenty other examples here, I guess I'll showcase one more that I think is pretty good, which is this guy who's building a chair. So, at first, he's got this long beard, but as we run iterations, it slowly kind of morphs into me. So that, by iteration maybe 800, I mean, it's starting to look me-ish, and then at iteration 1000, it's definitely approaching me. I still wouldn't think that's me. But if you let it keep going, maybe it'll get better.

So, feel free to play around with this. I find it really fun to see what you can do and this whole process of seeing yourself in different circumstances. These other people morphed into you. It's pretty fun. Hope you enjoy.

Video 23.5: A Simple Generative Language Model

Let's now turn our attention to text generation. When ChatGPT was publicly released in late 2022, it took the world by storm. Far better than any previously released chat AI, over 30 million users signed up within the first two months.

As the New York Times wrote in December 2022, “For most of the past decade, A.I. chatbots have been terrible…but ChatGPT feels different. Smarter. Weirder. More flexible. It can write jokes (some of which are actually funny), working computer code and college-level essays. It can also guess at medical diagnoses, create text-based Harry Potter games, and explain scientific concepts at multiple levels of difficulty.”

At first glance, it might seem like this is a sign of human-like intelligence. And indeed, there are many out there who've been tricked into thinking that there's true reasoning going on or that ChatGPT is sentient.

So, how does this nearly miraculous technology work if there's nobody there doing the thinking? You might have heard that the technology underneath ChatGPT is essentially a glorified autocomplete, and this is true. ChatGPT works by taking an existing string of text and then repeatedly predicting what the next text should be.

For example, if you give the model the prompt "Once Upon A...", it might add the word time. Then it will consider the text "Once upon a time" and might come up with "there". Each time the model is simply choosing from among a pool of likely possibilities to continue the text.

This basic idea of generation through auto-completion is very old. The earliest generative language model with which I'm familiar is given in Claude Shannon's 1948 paper, A Mathematical Theory of Communication.

Shannon's model is what today might be called an “N-gram model”. The idea is very simple. Given a corpus of English text, you build a frequency table of how often every sequence of N symbols appears. For example, suppose the vocabulary of symbols we're considering is the 26 letters of the English alphabet, plus a space for a total vocabulary of 27 symbols, and suppose the N we choose is 3.

Suppose our corpus of text is the Project Gutenberg copy of the novel *Moby-Dick*, and then we just count. We get a table back where “AAA” occurs zero times, “AAB” occurs zero times, “AAL” occurs one times, “AAM” occurs three times, “AAN” occurs two times, and so forth.

To use this model to generate text, we start with an input sequence, for example, “THIS IS THE Q.” Since this is a 3-gram model, we consider only the two most recent characters to continue generating output. In this case, “<space> Q”. From our frequency table, we see that “space Q” is followed by “U” 470 times and by “<space>” time. We generate our next symbol according to this distribution. For example, there's a one in 471 chance that we pick a “<space>” and a 470 out of 471 chance that we pick “U”. So, suppose we pick the letter “U.”

To generate the next symbol, we then consult the relative frequencies of the two-letter sequence “QU” as collected from *Moby-Dick*. These are in decreasing order of frequency: “QUE” 501, “QUI” 174, “QUA” 149, and “QUO” 84. So, “E” will be selected with a probability of 501 / 908, “I” with a probability of 174 out of 908, and so forth.

If I let this process keep going, I get an output like this example, “THIS IS THE QUEG BRICKS THAVION FIF THE YES GRAGGREAD.” That's basically just gibberish, though it does bear some resemblance to English.

There's some obvious modifications we could make. For example, we could select a different N, so if we did that for N = 5, we would be building a table of all 14,348,907 combinations of five characters.

Or we could pick a different vocabulary of symbols. For example, we could use words instead of letters. So, a 3-gram model using “words” as our “symbols” would count all occurrences of triplets of words. For example, “THE WHITE WHALE” appears in *Moby-Dick* 38 times, “THE WHITE SHARK” appears two times, etcetera. Then, when generating, we'd consider sequences of words. So, if we started with “THE WHITE,” we consider what entire word might come next.

N-gram models are an interesting, conceptual starting point but suffer from severe limitations. We'll explore these in an exercise which follows this video and in the next video, I'll talk about how ChatGPT overcomes these limitations.

Video 23.6: Modern Text Generation: Embeddings and Tokenization

Text generation has come a long way since Claude Shannon. Whereas N-gram models do a “hard” lookup, only literally replicating exact sequences of symbols that have been seen before, modern text generation instead performs what we might call a “soft” lookup, where the underlying meaning of the symbols are considered when figuring out which symbol to pick next. Let's talk about how this works.

First, let's talk about text representation. At present, the most popular technique is to represent each symbol as an embedding. For now, let's assume each symbol is an English word, such as “cow”, “horse”, “ocean”, or “lake”.

The embedding of each word is a vector in some high-dimensional space. In this demo, I'll be using a 300-dimensional model called “word2vec-google-news-300” that was generated using a corpus of Google News articles.

By using the “genism” library, I can download the embeddings provided by this model. After loading up the 1.5-gigabyte model, I can ask for the representation of a word. For example, I can write code that says, “cow_vector = word_vectors ["cow"],” which gives me back the 300 numbers that represent the word “cow”, which are 0.189, -0.075, -0.156, and so forth.

Of course, these specific 300 numbers mean nothing on their own. What matters is the relationships between embeddings of words.

For example, if I take the vector for “horse” and the vector for “cow” and compute the dot product, I get 4.74.

If I then take the vector for “horse” and the vector for “ocean”, I get the dot product 0.68. Unsurprisingly, that dot product is smaller, as “oceans” and “horses” are less similar to each other than “horses” and “cows”.

You can do some pretty neat stuff with embeddings, which you can play around with in the provided notebook. For example, if I take the embedding for the word “man” and subtract the embedding for the word “woman,” I get a “difference” vector that encodes masculinity somehow.

So, if I take that “difference” vector and I add it to the word “mother,” and I look for the closest word, I get back “father.”

Or if I add that “difference” vector to “girl” and look for the closest word, I get “boy.”

These semantic relationships can get subtle. For example, if I subtract “he” from “his,” I am left with a vector that might represent something like possessive inflection.

And indeed, if I add this “difference” vector to you and look for the closest word, I get “your.”

Embeddings can be applied to other units of text than words. For example, ChatGPT breaks your text into a series of tokens. Some tokens are entire words, whereas others are fragments of words or symbols.

As an example, consider the text, "Who'd use "后" vs. "後"?” Couple of Chinese characters. If we use the tokenizer on ChatGPT's website, we see that ChatGPT would break the sentence into 11 tokens. The first token is “Who,” the second is “<apostrophe>d,” the third is “<space>use.” The fourth is the simplified Chinese character "后", the fifth is a “<quote>”, the sixth is “<space>vs,” the seventh is “<period>”, the eighth is “<space><quote>”, the ninth is the traditional Chinese character "後", the tenth is “<quote>”, and the eleventh is a “?”.

Note that tokens are space- and case-sensitive, so the token “who” all lowercase is different than the token “Who,” and likewise “<space>use” is different from just “use.”

The nice thing about tokens is that they allow us to encode any input text, even things like misspelled words, new words, random sequences of characters, etc.

And as noted above, an embedding can be used to represent arbitrary tokens, even fairly abstract tokens like a single quote or a period.

Embeddings can also be applied to larger units of text, such as an entire document. For example, short stories about dinosaurs and outer space might all point in a similar direction. Embeddings can also be used to represent other input types entirely, such as pixels or entire images. So, a collection of pictures of burning barrels of trash might all have embeddings that point in the same direction.

Video 23.7: Transformer-Based Models

This brings us finally to ChatGPT. ChatGPT is an example of a transformer-based large language model. In this video, we'll discuss how transformer-based large language models use the idea of tokenization and embedding to generate text. In effect, they perform a "soft" lookup that takes into account meaning rather than a “hard” lookup like the n-gram model that just regurgitates previously seen sequences verbatim.

As a running example, imagine trying to complete the sentence "He devoured the orange". A human completing the sentence will probably decide that the word “orange”, in this case, refers to the fruit and not the color, though both meanings are possible. For example, for the fruit, a completion might be, "He devoured the orange hungrily", or for the color, a completion might be, "He devoured the orange candy". Somehow, our language model should tend to steer towards the former case of thinking of

the oranges of fruit while still allowing for an interpretation that “orange” could refer to the color.

At a high level, transformer-based large language models have three phases, which I'll call embedding, transformation, and unembedding. In the embedding phase, the input text is converted into a matrix of size D by N, where D is the dimensionality of the embedding space, and N is the number of tokens. Each column of this matrix is the embedding for the i-th token. Let's call this converted and put an embedded text matrix. Now, where do these D values come from? Well, the D values for each token in the vocabulary are stored in D by V matrix called the embedding matrix, where V is the number of tokens in the vocabulary for the model. So, for ChatGPT 3.5, D is 12,288 dimensions and V is a bit over 100,000 tokens.

For example, suppose we're trying to get a completion for our phrase, “He devoured the orange” using GPT 3.5. The GPT 3.5 tokenizer breaks this into five tokens. "He,” “<space>dev,” “oured,” “<space>the” “<space>orange.” We then form a matrix where the left most column are the 12,288 values corresponding to "He" as determined by the embedding matrix. The next column are the 12,288 values corresponding to “<space>dev”, and similarly for the other three tokens. Thus, "He devoured the orange" is ultimately represented by a 5 by 12,288 embedded text matrix of floating-point values with one column per token. Note that for now, these embeddings for each token are independent of each other. For example, the embedding for "orange" is some fuzzy vector that sits between all possible meanings of the word "orange", including the color and the fruit.

Turns out that creating this embedded text matrix can be done with a matrix multiplication of the original input by the embedding matrix. We'll explore how to do this in the exercises for this module.

That brings us to the second phase of the model, "Transformation". The goal of this phase is to adjust the embeddings of all of the tokens so that they take into account the context provided by the other tokens.

The transformation phase is done using a neural network called a transformer. Each level of the transformer takes as input D by N embedded text matrix and outputs another D by N embedded text matrix that is said to be enriched. That is, each token in the matrix now better takes into account the context of the rest of the sentence. There's a limit to end, known as the context window. Transformers cannot consider more tokens than this at the same time. For GPT 3.5, for example, the context window is 4096 tokens. That is, the model cannot consider tokens that occurred more than 4096 tokens ago.

We don't have time to go through the details, which could easily occupy an entire module, but at a high level, each layer of a transformer first figures out which pairs of tokens are related, then linearly adjusts the embeddings for those tokens to take into account their inferred relationships by adding a context vector. This is similar to earlier, where we saw how a possessive inflection vector could push the word "you" to "your".

So, for our orange example, the tokens for “<space>dev” and “oured” might add something like an “edible thing” vector to the “<space>orange” token, pushing it towards the food version of orange rather than the color.

If you'd like to learn more on your own, this process of allowing tokens to talk to each other is known as the “self-attention mechanism”. After applying the self-attention operation, each transformer layer then feeds each token individually through the same D-dimensional dense neural network, which allows the network to learn useful non-linear relationships

between embedding dimensions, in essence, manipulating the meaning of individual tokens in arbitrary ways.

Again, this is an incredibly concise explanation of a very deep topic. The main take away is that the transformer takes D by N embedded text matrix, and that puts a new D by N embedded text matrix that is richer.

Just how rich? Let's consider the final phase, the unembedding phase. In the unembedding phase, we take only the final column of the fully enriched embedded text matrix and use just that column to decide on the next token. So, for our running example, that means we extract that fifth and final column of our 12,288 by 5 matrix, that is a vector of length 12,288. We then multiply by a matrix called the unembedding matrix, which is of size D by V, where V is the size of a vocabulary. The resulting vector of length V gives the relative probabilities after a softmax that we should select each token. That is, after the transformation phase, that final token needs to be so enriched that it could be used all by itself to make a prediction for the next token.

We then sample from that distribution. For example, when I pass "He devoured the orange" using the OpenAI API, it tells me that the most likely completion is "He devoured the orange in". That is, the new token is “<space>in” with a probability that ChatGPT gives me of 13.6%, but there are other possibilities such as, “<comma>” at 10%, “<space>and” at 7.81%, “<space>with” at 5.03% and so forth. If we look at the list, the first example where orange seems to have been interpreted as an adjective is “<space>juice” at 1.53%.

As a totally different example, consider, "For my 16th birthday, Poseidon gave me a..." After the transformation phase, the token for "a" will need to

somehow take into account the fact that a gift is being given, that it is a birthday gift, that it is for a teenager, and that the giver is the Greek God Poseidon. There's a huge amount of domain knowledge here that has to be packed into the token for "a". For example, some of the likely next words when I ran an experiment are “trident”, “necklace”, and “horse”. Amazingly, this same strategy works even when the number of tokens is large. So, even with a couple 1000 tokens, that final token is still so rich that it could be used to predict an appropriate next token all by itself.

Note that text generation models have a setting called temperature, which can adjust the relative probabilities, in effect controlling how creative the model is allowed to be. If we pick a temperature of zero, we're saying, "No creativity; you must always pick the next token that has the highest probability". If we pick a temperature of one, then it uses exactly the probabilities that I listed. And then, if we get a little wild and pick a temperature larger than one, then the probabilities all get flattened. So, the relative likelihood of the less common tokens grows, and you get more creative output.

One natural and final question is where all those parameters even come from. So, the embedding matrix, the transformer weights, the unembedding matrix, and so forth. Well, those all come from training in the same basic way as the earlier networks we've discussed in class, albeit at an enormous scale. In this case, the input values to our model when we're training it are fragments of text from the real Internet, so posts on Reddit, news articles and so forth. And then, the output value for training is just the next token in the post that we've mined from online. To improve the number of training examples, we use every subsequence available. So, if we have a 1000-token Reddit post, we get 1000 training examples just from that post. That corresponds to predicting the first token in the post, the second token given

the first, the third given the first two, and so forth. For each of the training examples that we have, the gradient across all of our parameters, which may number in billions, is computed, and then we take a stochastic gradient descent step. After spending hundreds of thousands to maybe hundreds of millions of dollars repeating this process for all of the text on the entire Internet, we arrive at a set of weights that approximates the probability distribution of text on the Internet, and the results speak for themselves.

Video 23.8: Conclusion

Today, we've seen a survey of some select techniques used by generative AI models to produce output.

The first two big ideas we saw were the generative adversarial network and the diffusion model. Both of these are clever training procedures that could be used to coax neural networks into generating outputs. Today, we showed how they could be used to generate images, but they can also be used to generate other types of output.

The next big idea we saw was the embedding. An embedding function takes an input from a domain of interest such as text, sound or images and outputs a length D vector. Today, we focused only on embeddings for single tokens and words, but the idea can also be adapted towards other domains.

The final big idea we considered was the transformer. The idea here is that a transformer takes a sequence of embeddings and produces a new sequence of enriched embeddings that takes into account the relationships between those embeddings. In the context of text prediction, we take into account how words affect the meanings of other words that appear in the same sentence. So, we saw how “devoured” pushes the meaning of “orange” towards the fruit, not the color. This enrichment process is

surprisingly powerful. For example, for text generation, the next token can be generated using only the enriched embedding for the current token. This is true even if the sequence is thousands of tokens long.

The ideas from today can be mixed and matched in all sorts of fascinating ways that we will not cover. One obvious omission from today is how we can use text prompts to guide the output of an image generator, like a diffusion model or a GAN. The answer, it turns out, is embeddings. We first come up with an embedding scheme that maps images and text descriptions to the same embedding space. So, for example, the text “a goat wearing a crown” and an image of a goat wearing a crown should point in roughly the same direction in that shared space. By feeding the text embedding to an image generation model, we can guide the model to produce an image with a similar embedding though the details are beyond the scope of this module.

There are many important concepts that we haven't covered at all today, such as the specific architectures used by image generation networks, the idea of latent variables, and variational auto encoders.

Though we've only scratched the surface, I hope you leave the day with an appreciation for some of the biggest ideas in generative AI and solid preparation for a deeper dive in the future. That's it for today, and I'll see you next time.