Module 23: Generative AI

Quick Reference Guide

Learning Outcomes:

1. Identify the limitations of generative techniques.
2. Distinguish between the learning techniques of Gen AI.
3. Configure a diffusion model to generate personalized AI images.
4. Test different Gen AI models to compare similarities and differences.
5. Generate a narrative using AI text generators in order to effectively evaluate data patterns.
6. Calculate the similarity between sentences using transformers.
7. Evaluate the effectiveness of Gen AI application examples.
8. Identify current business applications of Gen AI technologies in finance, education, or healthcare.
9. Generate a prompt for a Gen AI platform in order to analyze its effectiveness.

Video 1: Introduction

In February 2019, Philip Wayne created ThisPersonDoesNotExist.com. Every time this page was refreshed, a new realistic human face was generated. Now at the time, this technology was largely unknown outside the research community.

As James Vincent wrote for The Verge at that 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 on a Unicorn" and get back an image like the one shown.

Generative AI is a broad domain of technologies that use artificial intelligence to generate useful output.

Basic Models	**Complex Models**
Draw random samples from some learned distribution	Allow users to direct or condition the output by providing specific inputs
Example: Models that learn the distribution of human faces, such as ThisPersonDoesNotExist.com	**Example:** Models that: • Convert text into images • Generate audio that mimics a

	specific person's voice from text • Transcribe spoken audio into text

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."

Video 2: Generative Adversarial Networks (GANs)

In the image shown, we see several faces that were generated using ThisPersonDoesNotExist.com.

These faces are generated by a neural network known as a “generator”, which takes a vector of random noise as input and produces a realistic image of a human face as output.

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.

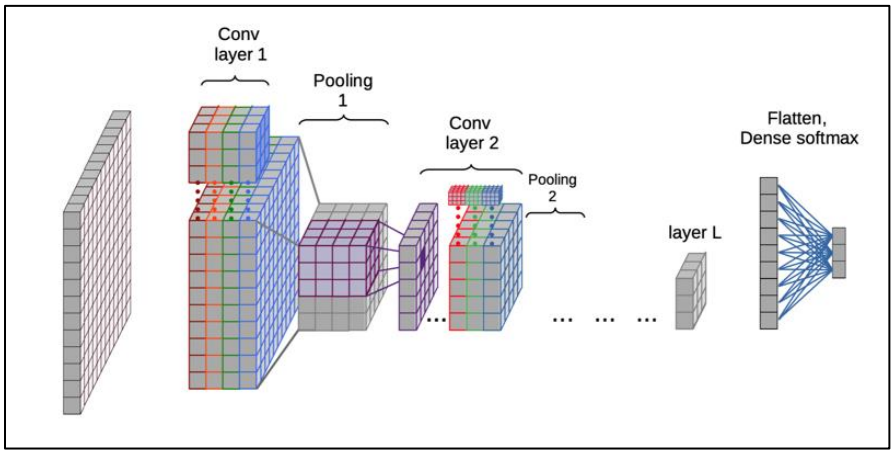
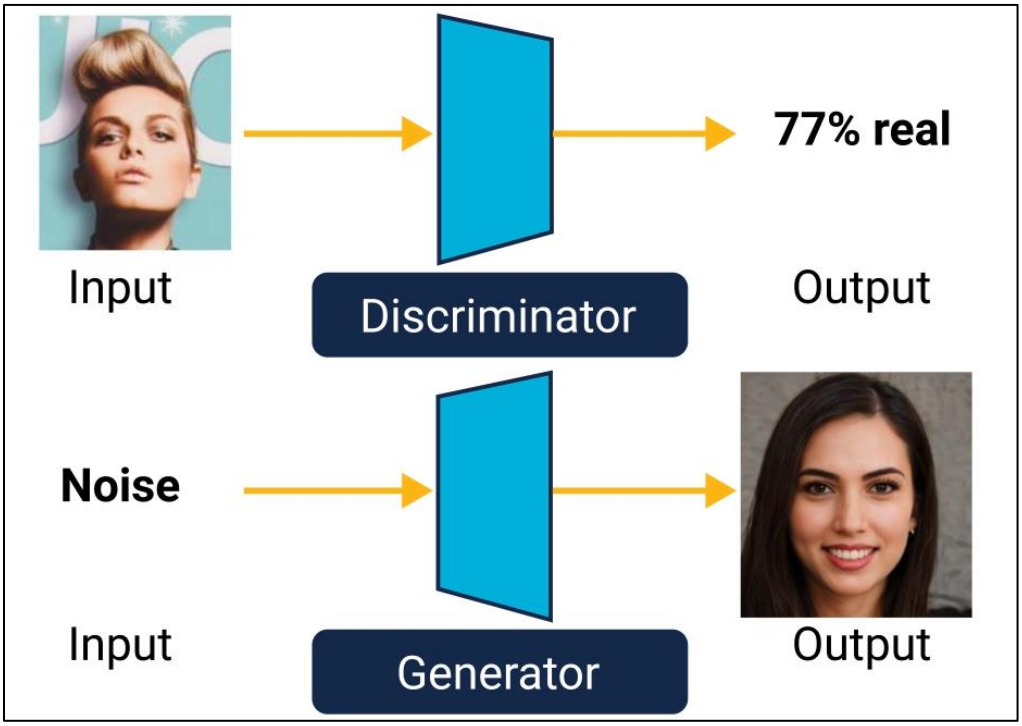


Image Classification Network	Image Generating Network
• Trains a model to classify images as faces and outputs a vector of probabilities for each know class • Requires identifying whether an image belongs to a specific class	• Trains a model to generate images by producing millions of numerical values representing pixel color values to form a realistic image • Requires generating a believable human face that aligns with human perception

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 first network is called the discriminator. It takes as input an image and it outputs the probability that the image is of a face. The second neural network is called the generator. It takes its 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, etc., but we

won't talk about any specific architecture for the generator, since that's not the main idea here.

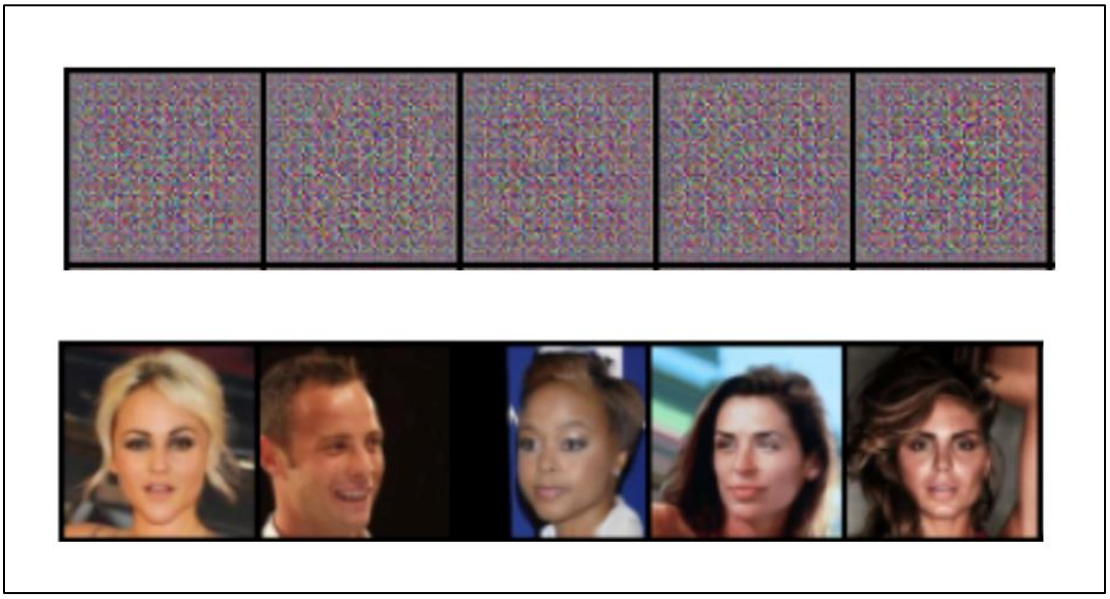
GANs use a training procedure where two networks compete against each other.

- The generator aims to create realistic human faces
- The discriminator aims to distinguish real faces from fake ones

If both networks are trained effectively, the generator will eventually produce faces that can even deceive humans.

GAN: Initial Iteration

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



GAN: Iteration One

- **Dataset selection:** Select a dataset to train the model
- **Training round:** Select N real faces from the dataset and generate N complementary fake faces using the generator
- **Initial generation output:** Observe that the generator initially produces random noise due to lack of learning

Role of Discriminator	Role of Generator
• Learns to differentiate between real and fake faces using cross-entropy loss when provided with a labeled batch of real and fake faces • Takes one step of stochastic gradient descent (SGD) and quickly improves its accuracy	• Uses the discriminator's performance as the generator's loss function, calculated by the negative log of $D(x)$ • Takes on SGD step and slightly improves its images generation

Generator's loss function

$$L(x) = -\log(D(x))$$

Where:

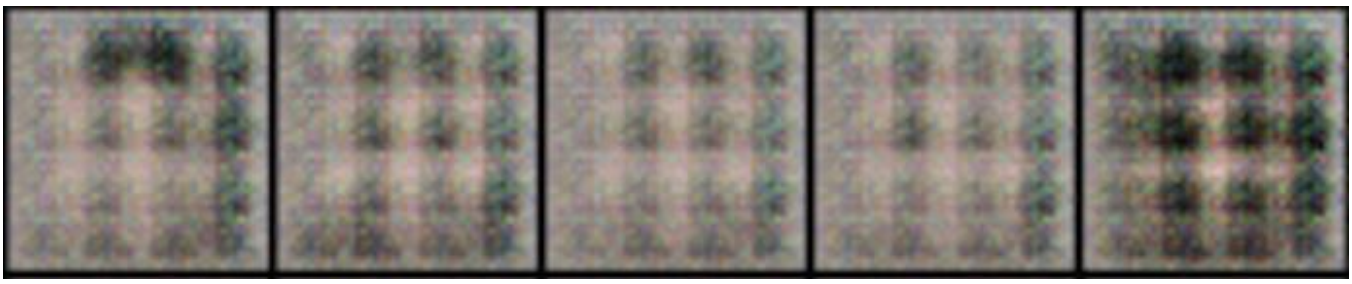
- x = Image generated

- $L(x)$ = Loss function
- $D(x)$ = Probability of the generated image being real

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

GAN: Iteration 100

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 trained was only outputting 64 by 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.

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.

Example:

$$L(x) = -\log\big(D(0)\big)$$

$$= \infty$$

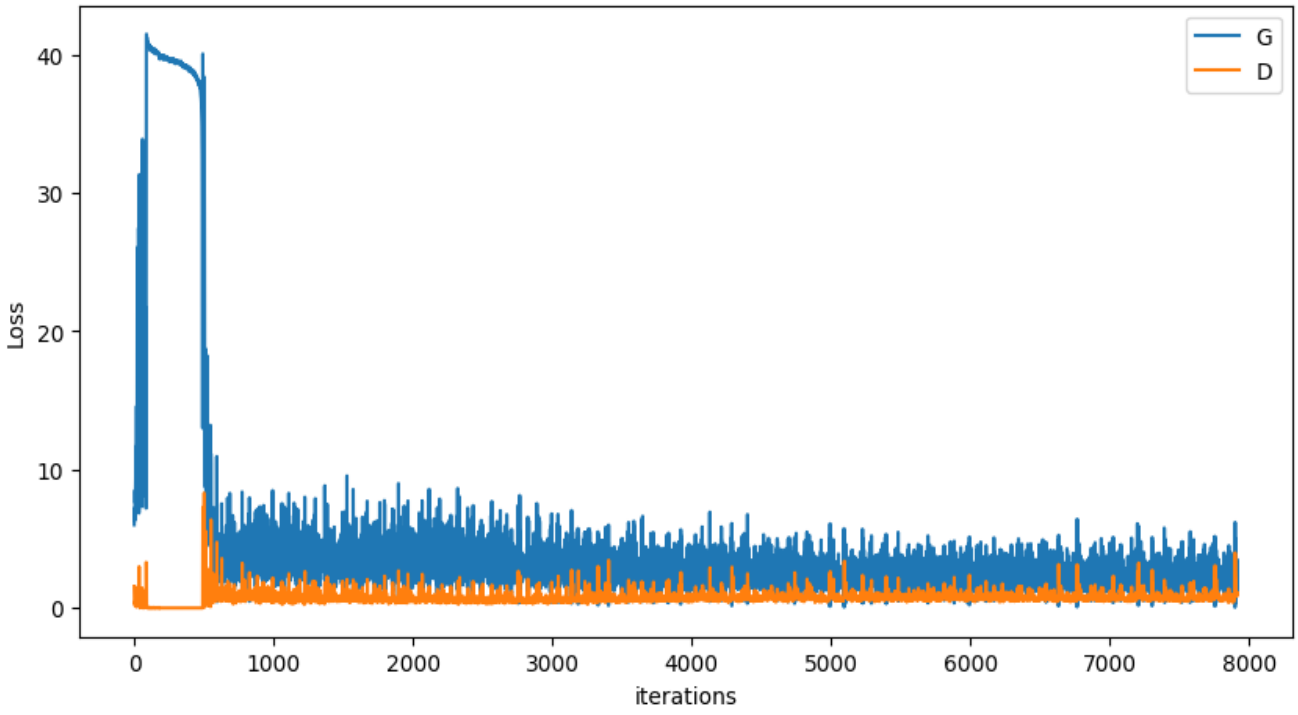
Where:

- x = Image generated
- $D(x)$ = Probability of the generated image being real = 0

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 600, 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.

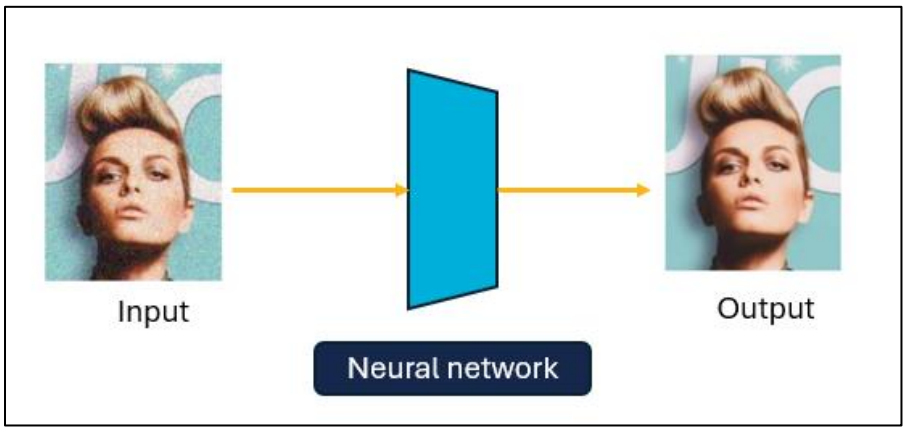
Various tricks exist for these common failure modes, such as adding noise to the input seen by the discriminator, though GAN training remains a significant challenge.

Video 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 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.

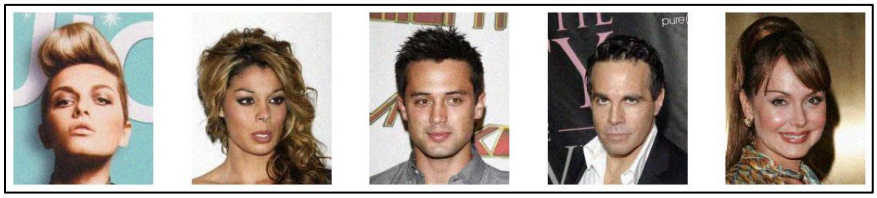
Now suppose we train a neural network to de-noise the images. That is, we provide pairs of images as a supervised learning dataset. The inputs are the noisy images, and the outputs are the original noiseless images. One obvious use for this network is that we could use it to de-noise images that were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



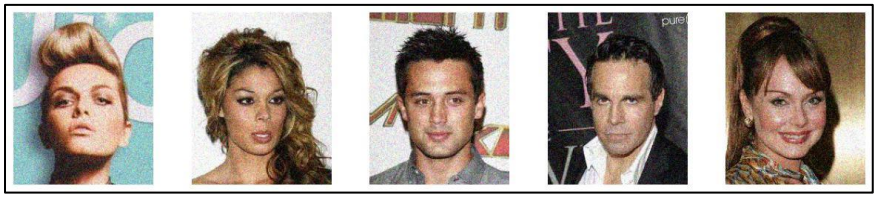
However, we have bigger ambitions. Imagine that instead of just adding noise one time, we add noise over and over, creating a series of increasingly noisy images. Eventually, the noise will dominate the original signal, and we'll be left with an image which is indistinguishable from Gaussian noise.

Level 0

Level 1

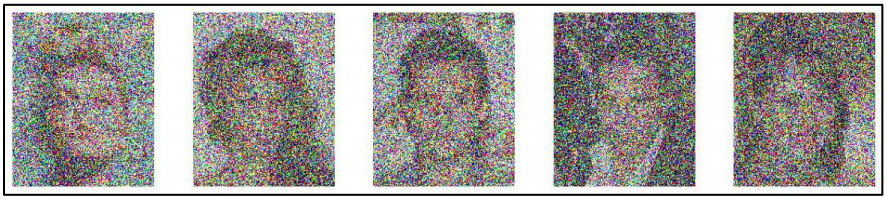


Level 2

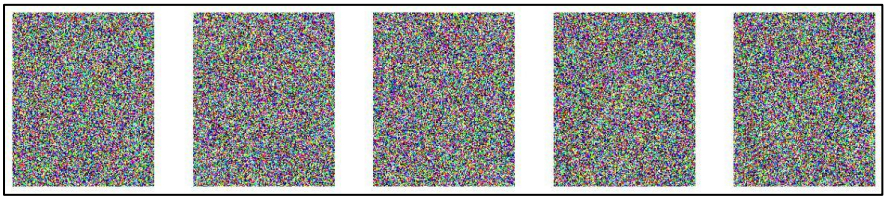


...

...



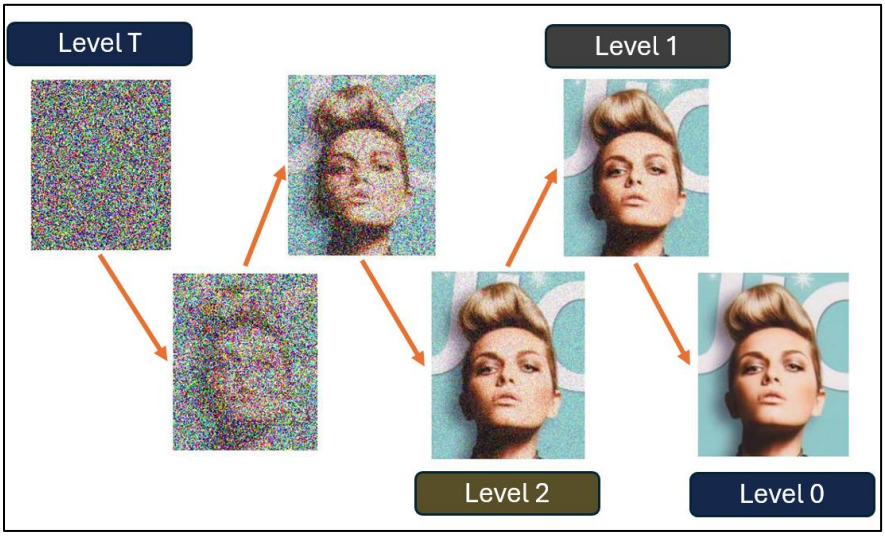
Level T



Let's give a name to these increasingly noisy images as follows:

- Level 0: Original non-noisy images
- Level 1: Images after the first round of noise is added
- Level 2: Images after two rounds of noise are added
- Level T: Images after T rounds of noise, where the image is now indistinguishable from Gaussian noise

Now imagine that we train T neural networks, where the first neural network takes the Level 1 images and learns to output Level 0 images, where the next neural network takes the 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 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 that are then upscaled. Another improvement is that instead of building T different 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 the time it takes to generate images using diffusion is typically much longer than a GAN.

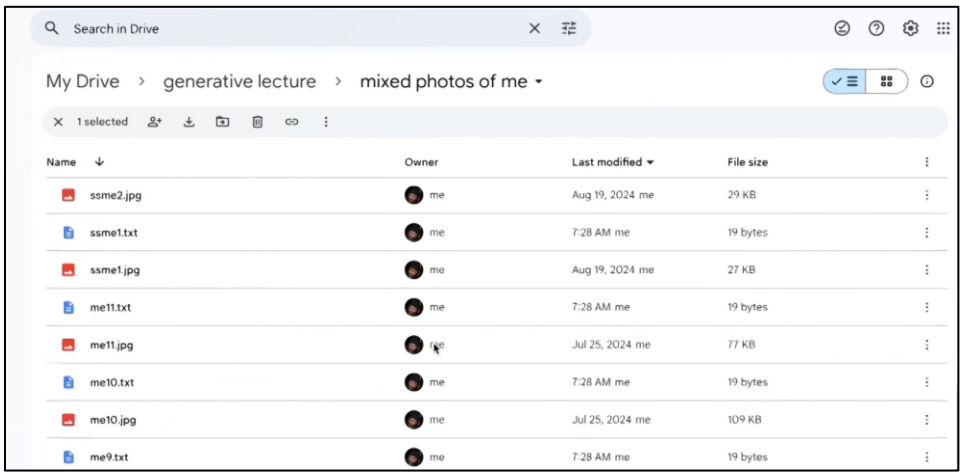
Video 4: Live Diffusion Demo

Behold this image. I generated this using an image generator called Flux and the specific prompt I used was, “joshhug 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 joshhug 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.

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 joshhug” and then the images show some picture of 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 joshhug 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.

But the big picture is that 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 collab. 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 24GB of VRAM is a what's necessary. So, that would be something like the Google collab, has a bunch of different choices. I used an A100.

Categories	Google Colab	A100
Options	Offers various GPU choices	Offers a specific high-end GPU

Cost	Provides relatively low-cost options	Provides expensive rental or ownership options
Pricing	Features different pricing plans	Features usage at a rate of $1 to $2 per hour
Usability	Is user-friendly and adaptable	Is excellent for intensive tasks requiring significant VRAM

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.

```
66        sample:
67          sampler: "flowmatch" # must match train.noise_scheduler
68          sample_every: 200 # sample every this many steps
69          width: 512
70          height: 512
71          prompts:
72            # you can add [trigger] to the prompts here and it will be replaced wi
73            - "joshhug holding a sign that says 'I LOVE PROMPTS!'"
74            - "joshhug with red hair, playing chess at the park, bomb going off in
75            - "joshhug holding a coffee cup, in a beanie, sitting at a cafe"
76            - "joshhug building a log cabin in the snow covered mountains"
77            - "joshhug playing the guitar, on stage, singing a song, laser lights,
78            - "joshhug with a beard, building a chair, in a wood shop"
79            - "photo of joshhug, white background, medium shot, modeling clothing,
80            - "joshhug showing off his cool new t shirt at the beach, a shark is j
81            - "joshhug is a DJ at a night club, fish eye lens, smoke machine, laze
82            - "joshhug is in a recording studio demonstrating on his laptop how to
83          neg: ""  # not used on flux
84          seed: 42
85          walk_seed: true
86          guidance_scale: 4
87          sample_steps: 20
88  # you can add any additional meta info here. [name] is replaced with config name
```

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 joshhug.

Now, this joshhug definitely looks more like me, but much more Rodenty and I think pretty far from what I look like. Iteration four hundreds arguably kind of better, though I do recognise my eye, once I get to iteration 600 it's doing better

800 even better though sort of broii and then by iteration.

But 1000, it's looking pretty good, and I could keep going, but I would believe that this is an image of myself looking.

It's not a monotonic process of necessarily getting better and of course the outputs are random.

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

Video 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 is a Glorified Autocomplete

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.

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 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 three, 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.

Sequence	Frequency
AAA	0
AAB	0
…	
AAL	1
AAM	3
AAN	2
…	
ZZZ	0
_AA	0
_AB	323
…	

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 “_Q”. From our frequency table, we see that “_Q” is followed by “U”, 470 times and by “_” one 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”.

Sequence	Frequency
…	
_QA	0
_QB	0
_QC	0
…	
_QU	470
_QV	0

...	
_Q_	1
_RA	241
...	

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 divided 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". This is basically just gibberish, though it does bear some resemblance to English.

Sequence	Frequency
...	
QUA	149
QUE	501
QUI	174
QUO	84
RAA	1
RAB	88
RAC	136
...	...

There are some obvious modifications we could make. For example, we could select a different N, e.g. for "N=5" we build a table of all 14,348,907 combinations of 5 characters.

Sequence	Frequency
AAAAA	0
AAAAB	0
AAAAC	0

...	

Or we could pick a different vocabulary of symbols, e.g. 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, e.g. “THE WHITE WHALE” appears 38 times, “THE WHITE SHARK” appears 2 times, etc. Then when generating we’d consider sequences of words, e.g. if we started with “THE WHITE”, we’d consider what entire words might come next.

Sequence	Frequency
THE WHITE BEAR	1
THE WHITE BELT	1
THE WHITE BUBBLES	1
...	
THE WHITE SHARK	2
THE WHITE SILENT	1
THE WHITE WHALE	38
THE WHITE WHALES	2

N-Gram models are an interesting conceptual starting point but suffer from severe limitations.

Video 6: Modern Text Generation: Embeddings and Tokenization

Text generation has come a long way since Claude Shannon. Whereas N-gram models do a hard look up, only literally replicating exact sequences of symbols that have been seen before, modern text generation instead performs what we might call a soft look up, 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 Word to vec, Google News 300 that was generated using a corpus of Google News articles. By using the Gen SIM 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.

```
import gensim.downloader as api
word_vectors = api.load("word2vec-google-news-300")
```

For example, I can write code that says Cow vector equals word vectors cow, which gives me back the 300 numbers that represent the word cow,

```
cow_vector = word_vectors["cow"]
```

which are 0.18945312, -0.07519531, -0.15625, and so forth. Of course, these specific 300 numbers mean nothing on their own. What matters is the relationships between embeddings of words.

```
array([ 0.18945312, -0.07519531, -0.15625 , 0.19921875, -0.18457031,
0.20703125, -0.04125977, -0.01428223, 0.00363159, -0.09570312, ...
```

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.

```
cow_vector = word_vectors["cow"]
horse_vector = word_vectors["horse"]
```

```
np.dot(horse_vector, cow_vector)
```

4.74

If I then take the vector for horse and the vector for ocean, I get the dot product 0.68.

```
np.dot(horse_vector, ocean_vector)
```

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.

```
vector_diff = word_vectors["man"] - word_vectors["woman"]

new_vector = word_vectors["mother"] + vector_diff

closest_word = word_vectors.similar_by_vector(new_vector, topn=1)
closest_word
```

```
[('father', 0.8103864789009094)]
```

Or if I add that difference vector to girl and look for the closest word, I get boy. These semantic relationships can get subtle.

```
vector_diff = word_vectors["man"] - word_vectors["woman"]

new_vector = word_vectors["girl"] + vector_diff

# Find the closest word to the new vector
closest_word = word_vectors.similar_by_vector(new_vector, topn=1)
closest_word
```

```
[(boy, 0.8843863606452942)]
```

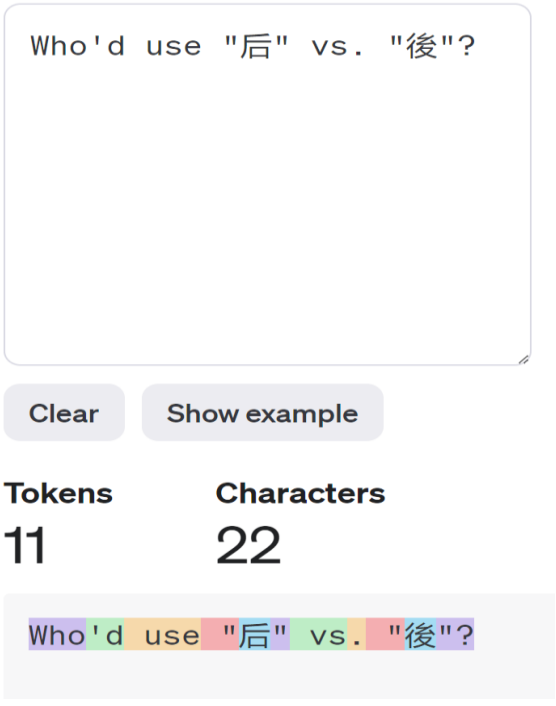
For example, if I subtract “he” from “his”, I am left with a vector that might represent something like possessive inflection. And indeed, if I add this difference vector to you and look for the closest word, I get “your”.

```
vector_diff = word_vectors["his"] - word_vectors["he"]

new_vector = word_vectors["you"] + vector_diff

# Find the closest word to the new vector
closest_word = word_vectors.similar_by_vector(new_vector, topn=1)
closest_word
```

```
[('your', 0.8750520348548889)]
```

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"?”



If we use the tokenizer on ChatGPT's website, we see that ChatGPT would break the sentence into 11 tokens. The 1st token is "Who". The 2nd is "'d". The 3rd is "_use". The 4th is "_"". The 5th is the simplified Chinese character "后". The 6th is """. The 7th is "_vs". The 8th is ".". The 9th is "_"". The 10th is the traditional Chinese character "後", and the 11th is ""?".

Note that tokens are space and case sensitive, so the token "who" all lower case is different than the token capital "Who", and likewise "_use" is different from "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.

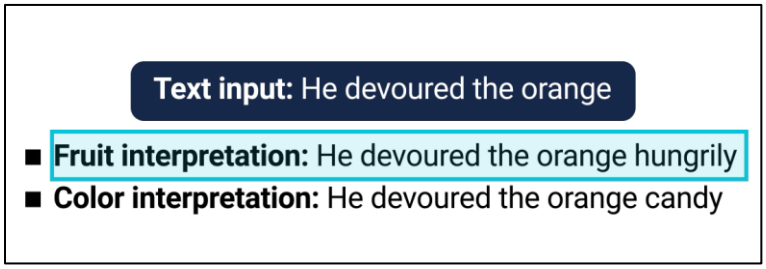
And as noted above, an embedding can be used to represent arbitrary tokens, even fairly abstract tokens like a single quote or a period. Embeddings can also be applied to larger units of text, such as an entire document. For example, short stories about dinosaurs and outer space might all point in a 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 7: Transformer-Based Models

This brings us finally to ChatGPT. ChatGPT is an example of a transformer-based large language model. 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.

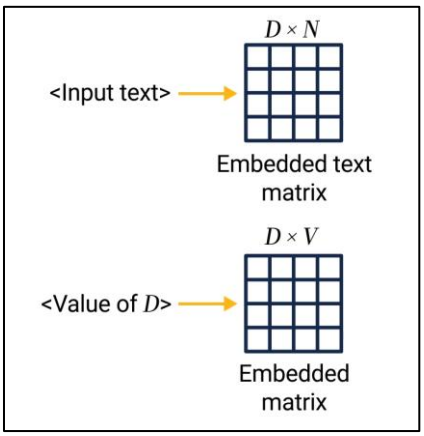
As a running example, imagine trying to complete the sentence, “He devoured the orange”. A human completing this sentence will probably decide that the word orange in this case refers to the fruit and not the color, though both meanings are possible. For example, for the fruit, a completion might be, “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 orange as a fruit, while still allowing for an interpretation that orange could refer to the color.



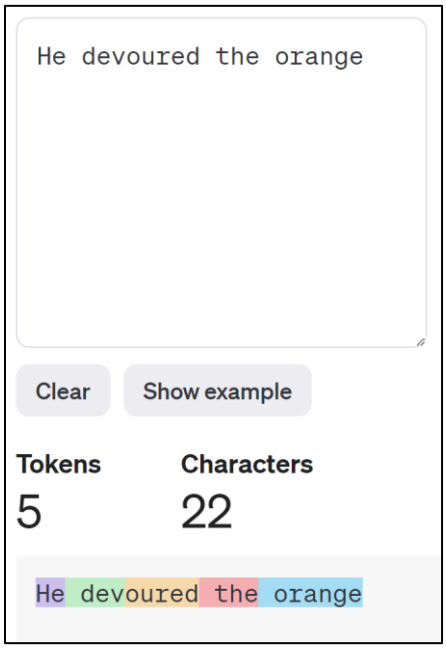
At a high level, transformer-based large language models have t**hree phases** which I'll call **embedding**, **transformation**, and **unembedding**.

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 ith token. Let's call this converted input 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 a 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”, “_dev”, “oured” “_the”, and “_orange”.



We then form a matrix where the leftmost 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 “_dev”, and similarly for the other three tokens. Thus, “He devoured the orange” is ultimately represented by a five by 12,288 embedded text matrix of floating point values with one column per token.

12,288

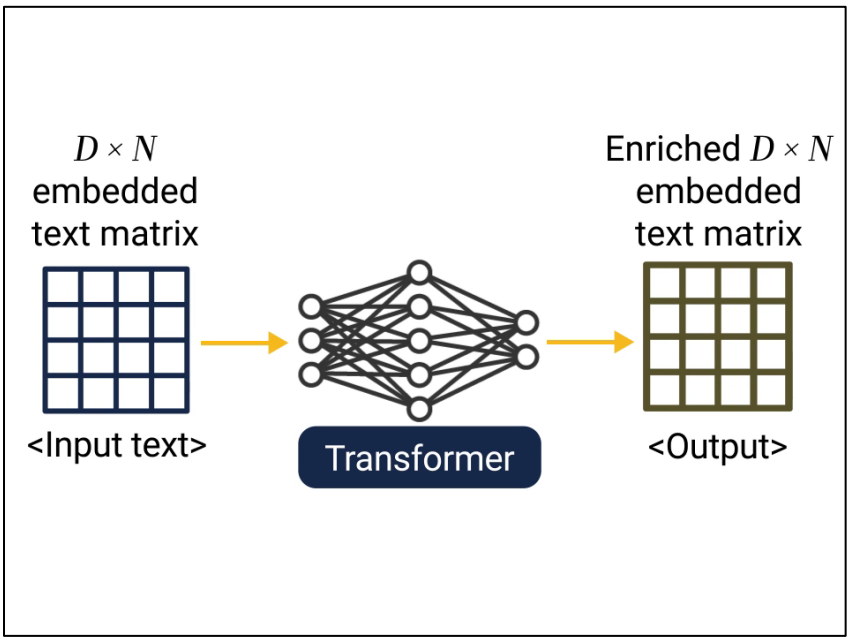
He	_dev	oured	_the	_orange
0.1272	0.1851	0.9916	-0.1555	0.5994
-0.3112	0.2934	0.5122	-0.4717	0.1277
0.1821	-0.1553	-0.1522	0.9187	0.1767
0.2113	0.5818	-0.6812	-0.1223	0.9149
…	…	…	…	…

5

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 It 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 a 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 N known as the

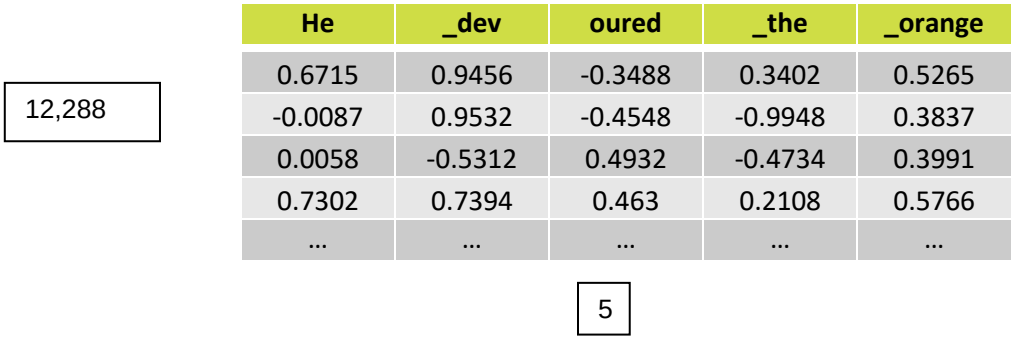
context window. Transformers cannot consider more tokens than this at the same time.



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 adjust 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 "_dev" and "oured" might add something like an edible thing vector to the "_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 a D by N embedded text matrix and outputs a new D by N embedded text matrix that is richer. Just how rich?

Well, 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.

	_orange
12,288	0.5265
	0.3837
	0.3991
	0.5766

	apple	aspa	...	zebra
12,288	0.1882	0.1548	...	-0.8711
	0.3481	-0.9134	...	-0.1161
	-0.1827	0.5171	...	0.9131
	0.6619	-0.0319	...	-0.4712
	...	...	...	...

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 “_in” with a probability that ChatGPT gives me of 13.6%. But there are other possibilities such as “,” at 10.02% “and” at 7.81% “with” at 5.03% and so forth. The first possibility on the list where orange seems to have been interpreted as an adjective is “_juice” at 1.53%.

Next Token	Percent Chance
_in	13.6%
,	10.02%
\n\n	8.34%
_and	7.81%
_with	5.03%
.\n\n	4.91%
.	3.95%
_as	2.14%
_juice	1.53%
_peel	1.31%
...	...

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 “Triton”, “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.

- **Temperature 0:** Always picks the most likely token
- **Temperature 1:** Picks based on the calculated probabilities
- **Temperature > 1:** Flattens probabilities making less common tokens more likely

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 1,000 token Reddit post, we get 1,000 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.

Original post: the undertaker threw mankind off hell in a cell, and plummeted sixteen feet through an announcer's table.

Training samples derived from this post:

the _under

the _under taker

the _under taker _threw

the _under taker _threw mankind

the _under taker _threw mankind _off

the _under taker _threw mankind _off _h

For each of the training examples that we have, the gradient across all of our parameters, which may number in the 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 8: Conclusion

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. 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.

One obvious omission 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.