How is it so good ? (DALL-E Explained Pt. 2)

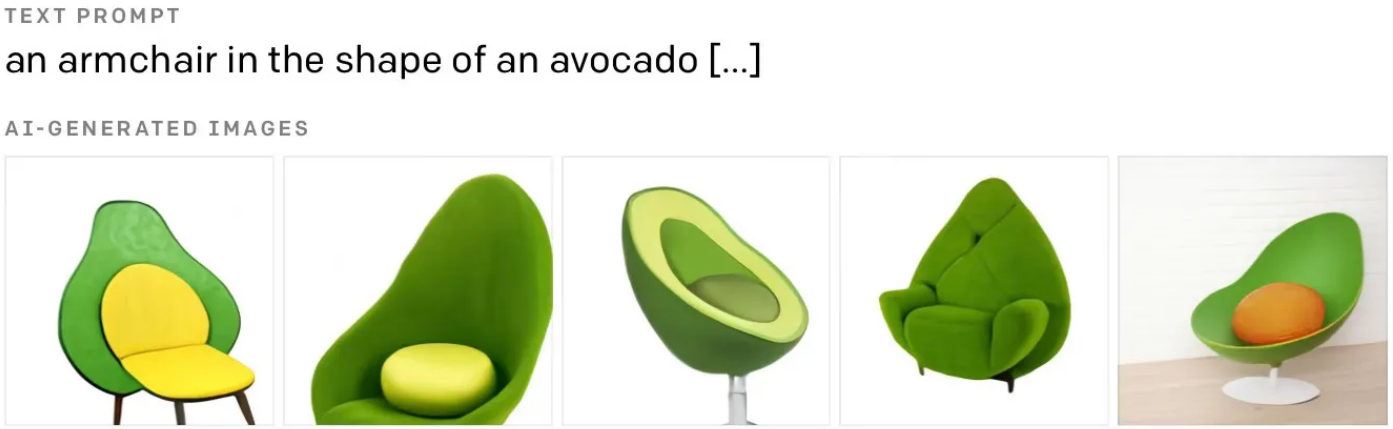
MACHINE LEARNING AT BERKELEY
7 APR 2021

By Charlie Snell

DALL-E consists of two main components. A discrete autoencoder that learns to accurately represent images in a compressed latent space. And a transformer which learns the correlations between language and this discrete image representation.

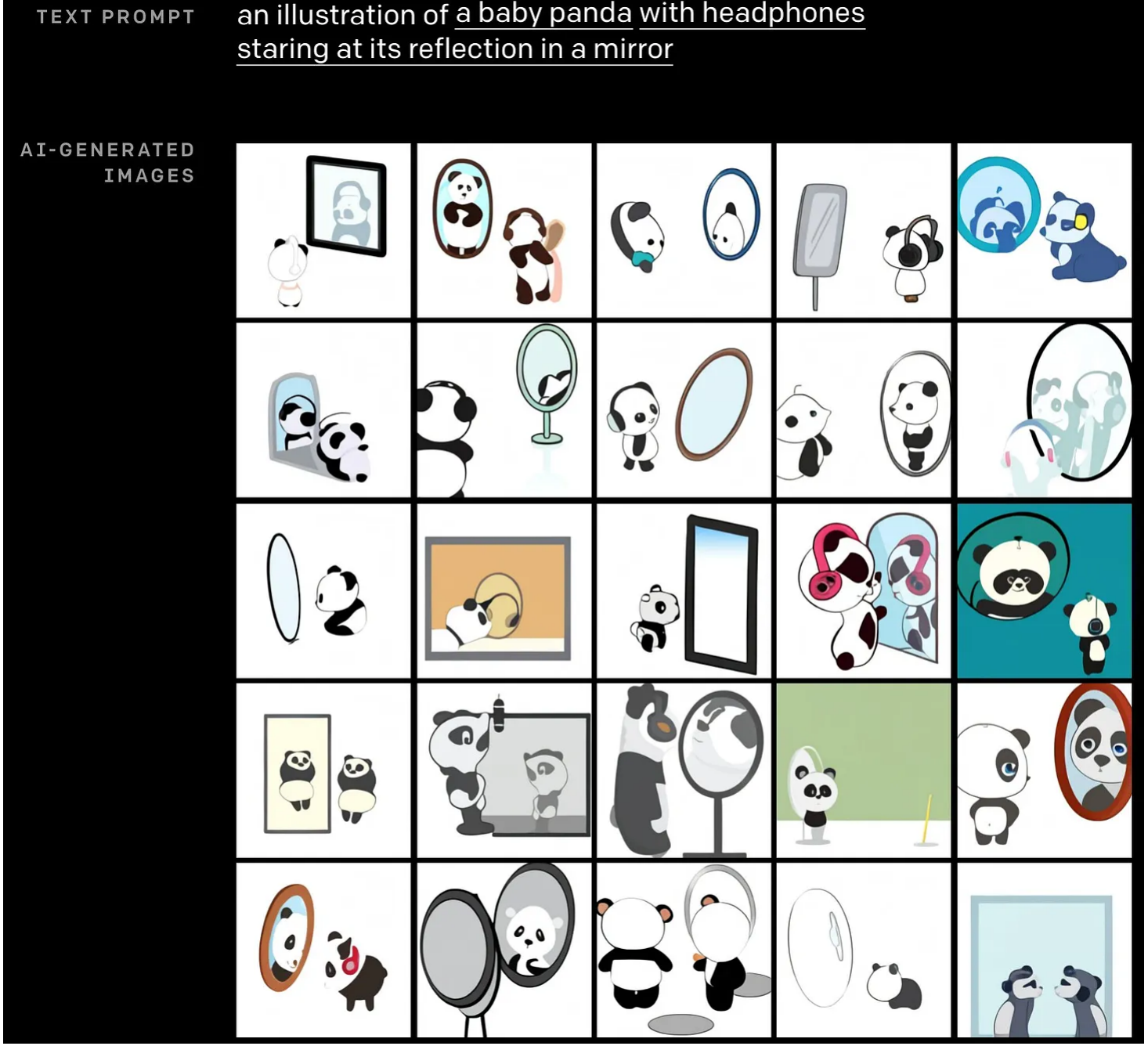
In part one of this series, we focused on understanding the autoencoder. Specifically, we looked at a particularly powerful technique for this called VQ-VAE. According to the now published paper though, DALL-E uses a slightly different method to learn it's discrete representations; they call it dVAE. While the exact techniques are a bit different, the core goal remains the same. Nonetheless, I will briefly explain the differences at the beginning of this post. The rest of this post will focus on DALL-E's transformer.

The transformer is arguably the meat of DALL-E; it is what allows the model to generate new images that accurately fit with a given text prompt. It learns how language and images fit together, so that when the model is asked to generate images of "an armchair in the shape of an avocado", it's able to spit out some super creative designs for Avocado chairs that have probably never been thought of before.



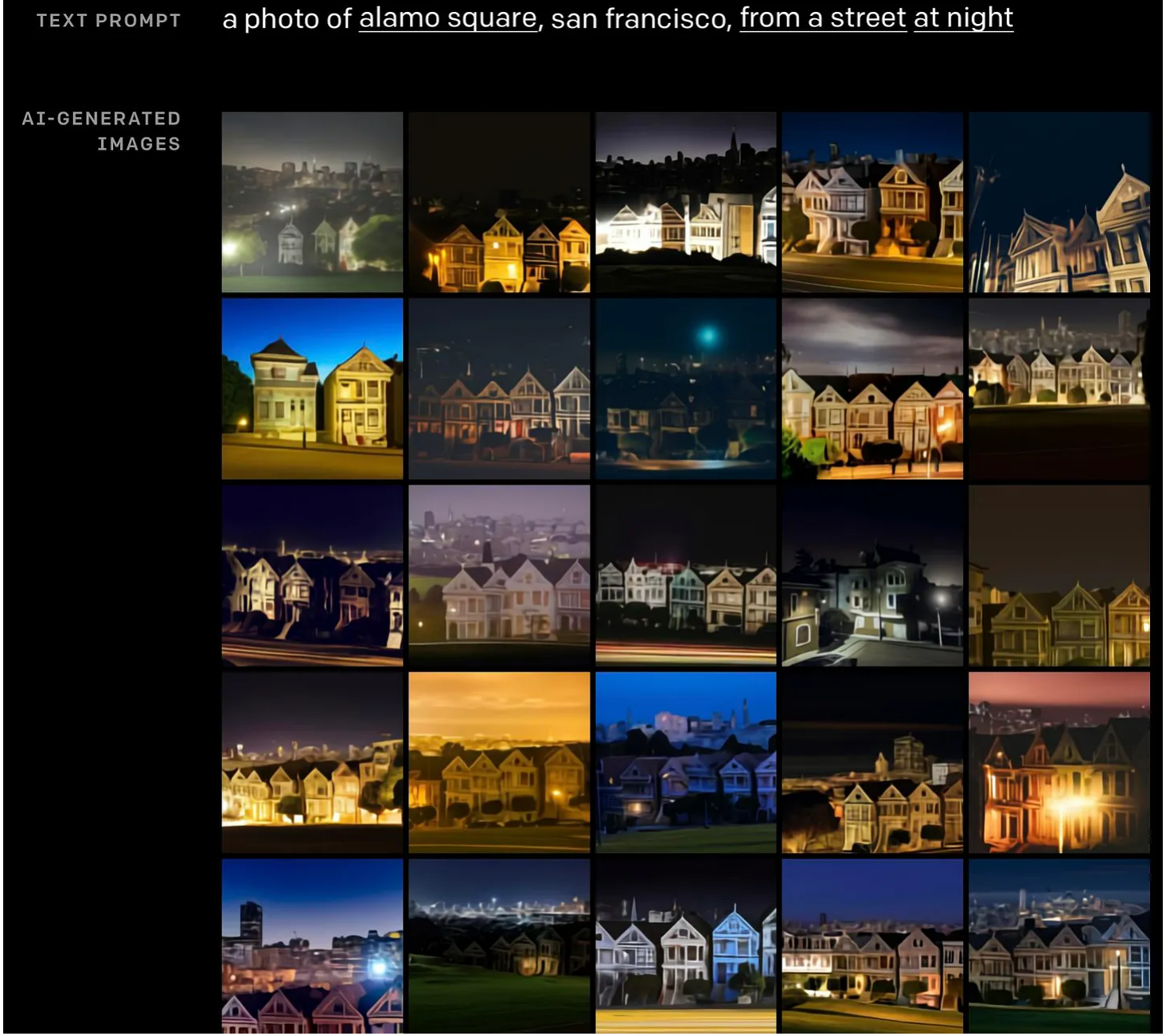
([image source])

Beyond just creative and accurate designs, the transformer also seems to understand some common sense physics. For instance, asking for “an illustration of a baby panda with headphones staring at its reflection in a mirror”, produces roughly this, all the way down to the physical details of the mirror’s reflection.



([image source](#))

The model also seems to be jam packed with factual knowledge about our world; asking for “a photo of Alamo Square, San Francisco, from a street at night” generates exactly that.



([image source](#))

There are so many more impressive examples like this on OpenAI's blog (unfortunately the model itself is not released, so we can't test these examples ourselves; we can only go off of what OpenAI demonstrates in the blog post).

All of this raises the question: how is it so good?

Nobody knows exactly why transformers work so well, or even what they actually learn; there is no fundamental theory for deep learning that can explain all of this, these networks are sort of too big and complicated for us to fully understand currently. Most of what we have are just these crazy empirical results like DALL-E. You train a big model with lots of data and follow a set of mostly empirically derived best practices and suddenly your model can generate images of Avocado chairs on command. No one can fully explain it; it just works.

So really, this question of "how is it so good?" is an open research question. Nonetheless there are some general intuitions that can help with understanding the capabilities and limitations of these types of models.

In order to properly explore the question "how is it so good?", this following blog post is organized into two farily distinct parts:

- The first half will focus on understanding how all the different pieces of DALL-E fit together to generate high quality images from a text prompt. In particular, this section will look at the technical role that the transformer plays in all of this.
- Then, armed with an understanding of how exactly the transformer fits into DALL-E's architecture, the second half will focus on more philosophical questions about the transformer's capabilities, like "how is it so good?" and "why are transformers able to do all of this?"

  In this part I will focus less on the technical details of transformers; there are already so many great blogs that cover this topic in incredible detail. I can recommend Jay Allamar's excellent blogs on this topic, and for a more detailed look, check out Harvard NLP's annotated transformer.

I will also not attempt to answer these questions definitively, since these are open research questions that no one really knows the answer to, instead I will just present some interesting intuitions that give somewhat of a clearer picture on what these models can do and what's really going on here.

Note: this blog post assumes that you have some knowledge of deep learning and Bayesian probability.

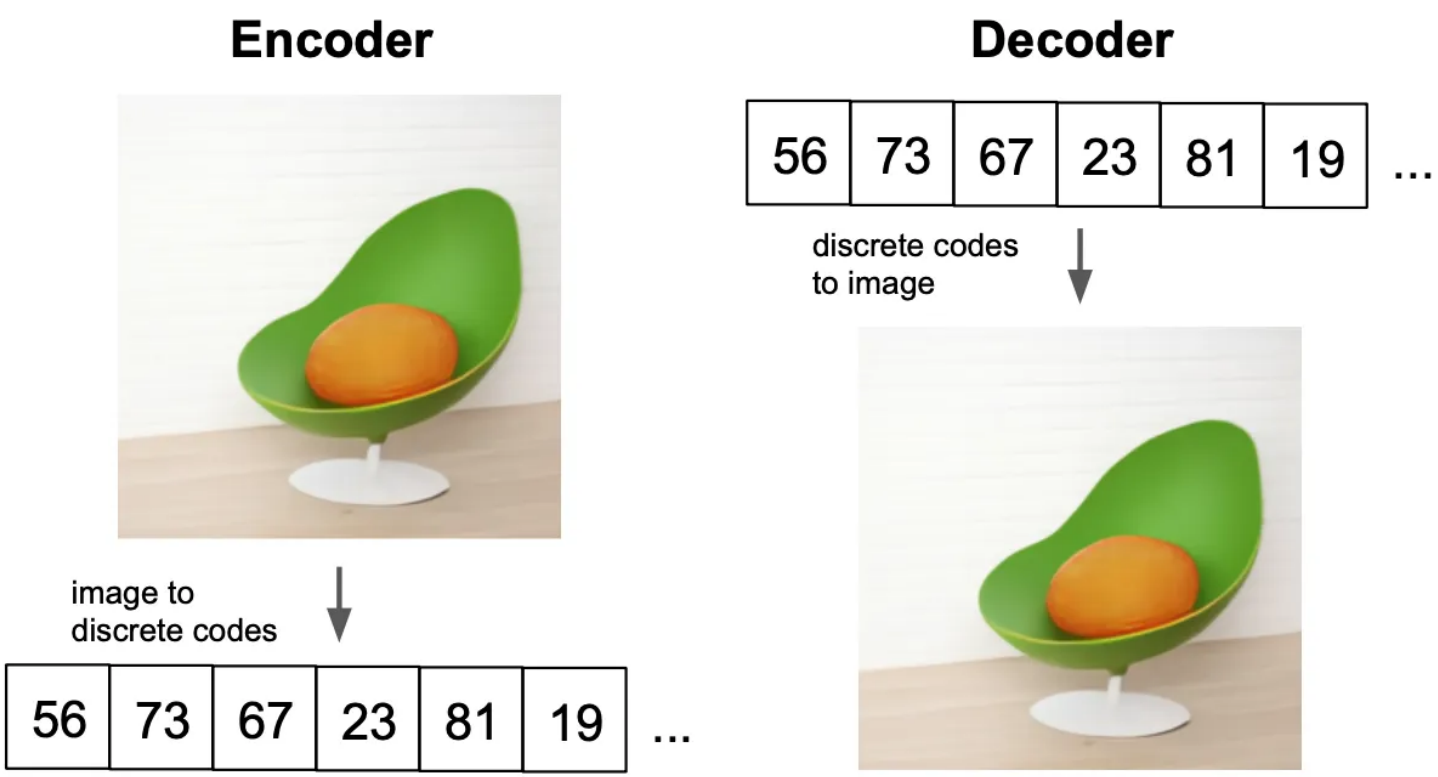
Looking Back at Part 1 (dVAE explained)

Before jumping into the transformer side of DALL-E, I want to briefly correct some of the assumptions made in part one.

When I wrote part one of this series, the official paper for DALL-E hadn't been released yet, so I could only go off of the vague details from OpenAI's blog, which mentioned in a footnote that they used a VQ-VAE-like model to represent the images. Now that the paper has finally been released, we can take a look at the details of their discrete autoencoder, which they call dVAE.

Ultimately, the high level goal of dVAE is the same: it attempts to learn a discrete representation for images. The specifics of how exactly it does this are just a bit different from VQ-VAE.

(Note: this section is somewhat disconnected from the rest of the blog post, so if you are more interested in the transformer stuff, feel free to skip to the next section.)

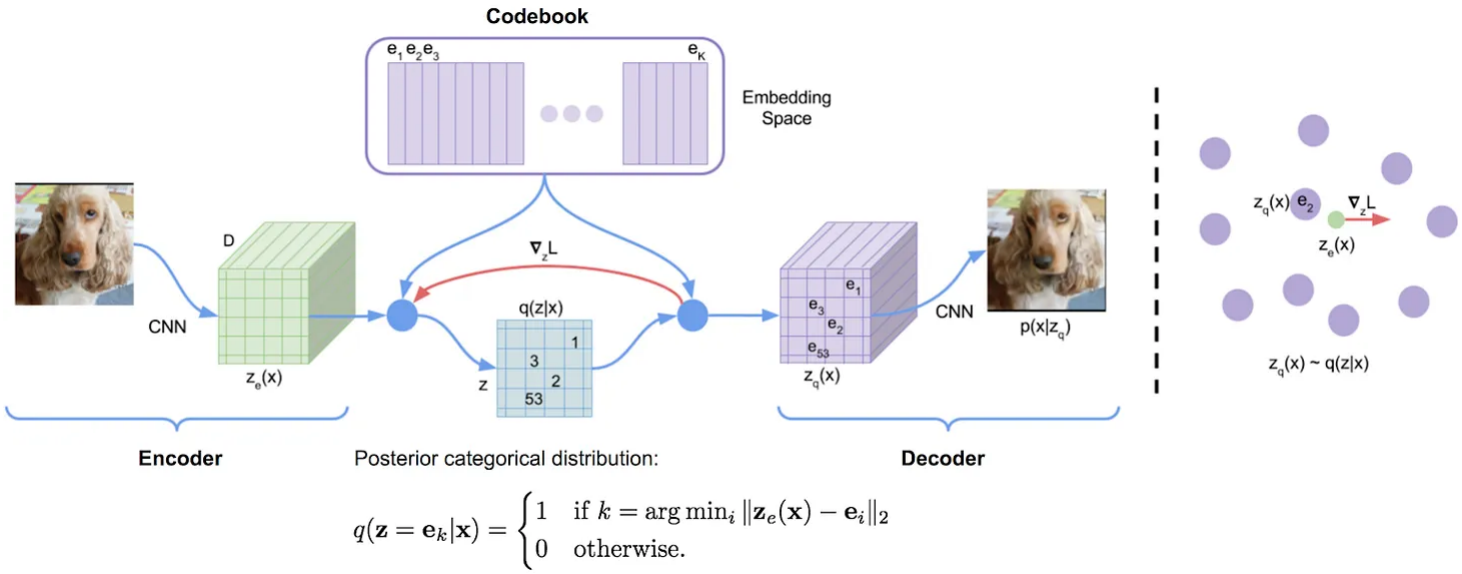


VQ-VAE Review

Recall that VQ-VAE learns a codebook; basically just an indexable lookup table for a finite set of learned vectors. The encoder network is then responsible for taking in an image and outputting a set of vectors, where each one is ideally close to some codebook vector.

These encoder outputs should in total have a much smaller dimension than the original image; the entire goal here is to force the model to compress the image, so that it learns a more fundamental representation for images that highlights only most important features.

After the encoder produces a set of outputs, the VQ-VAE bottleneck then maps each encoder output vector to its nearest codebook vector. And then lastly, these codebook vectors are fed to a decoder network, which is tasked with reconstructing the original image.



$$q(\mathbf{z} = \mathbf{e}_k|\mathbf{x}) = \begin{cases} 1 & \text{if } k = \arg\min_i \|\mathbf{z}_e(\mathbf{x}) - \mathbf{e}_i\|_2 \\ 0 & \text{otherwise.} \end{cases}$$

Now if we just train this model by itself with the standard VAE objective, it will not work. A few additional vector quantization tricks have to be employed in order to backprop through the bottleneck and to properly align encoder vectors with codebook vectors.

Fundamentally, the reason that these tricks are necessary is because of the model's assumption that $q(z|x)=1$ if

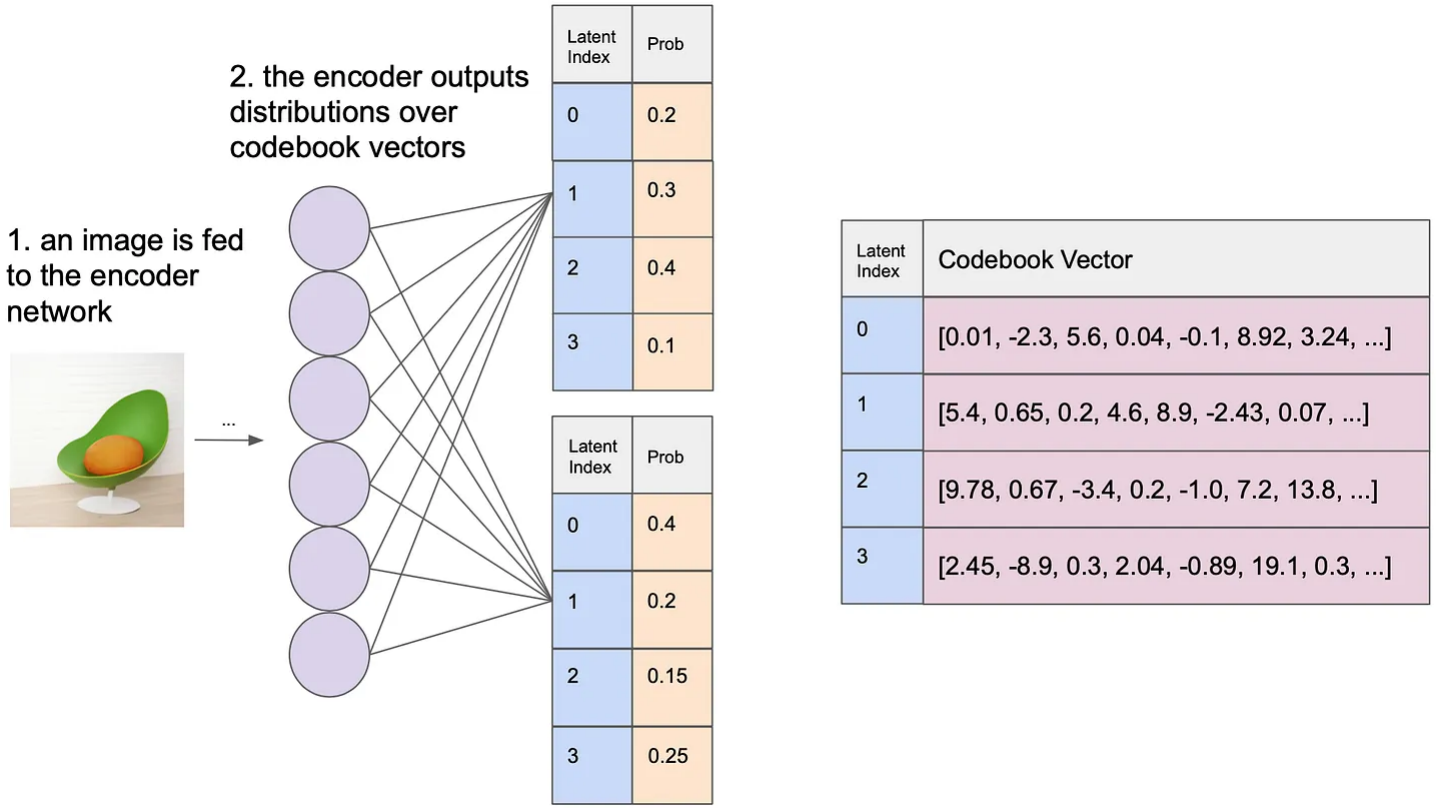
$$z = \text{argmin}_i ||z_e(x) - e_i||_2$$

and 0 otherwise. Here x is the input image, $q(z|x)$ is the distribution of latent vectors given the image, $z_e(x)$ is a vector output by the encoder given the image, and e_i are the set of codebook vectors. Basically this equation is saying that the VAE's posterior distribution is deterministic; it assigns probability 1 to the codebook vector nearest to the encoder's output.

Uncertainty in the Posterior

The posterior distribution does not, in general, need to be deterministic though. In fact, you could imagine situations where the encoder might be somewhat uncertain about which codebook vector to output. VQ-VAE forces the model to pick just one vector in all situations, but it might be better for the encoder to express some uncertainty over latents in the posterior distribution. This is what dVAE does.

The dVAE encoder still outputs some grid of discrete latents for a given image. But instead of producing latents that are each mapped deterministically to a single codebook vector, the dVAE encoder outputs a distribution over codebook vectors for each latent.



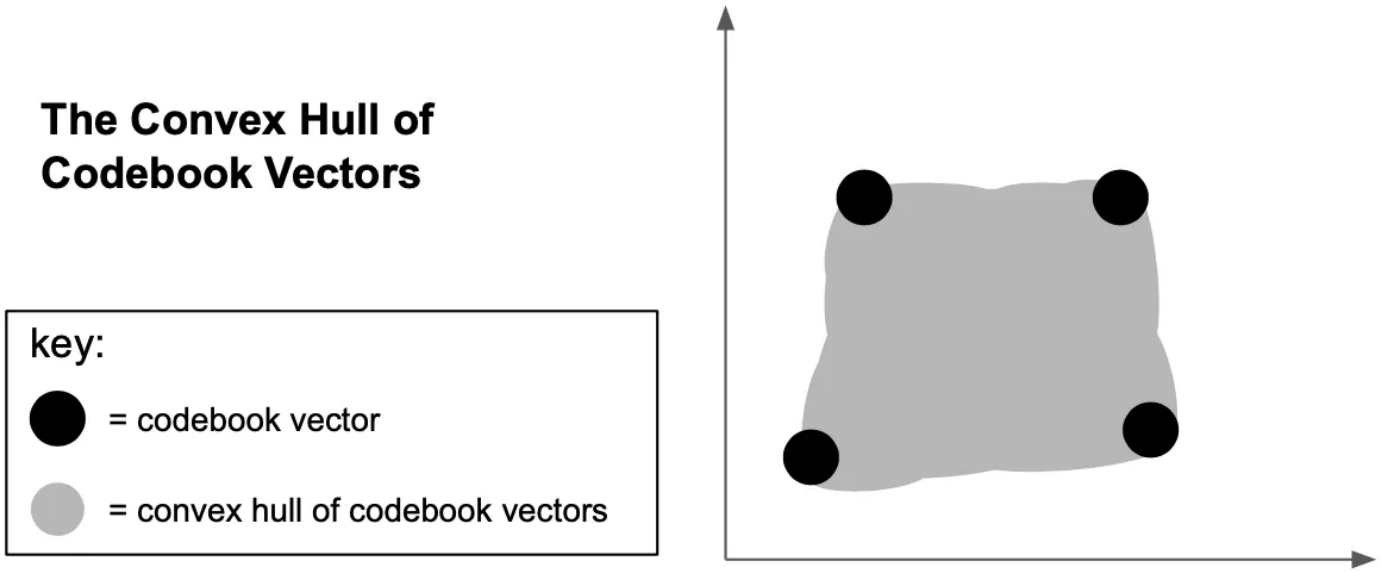
dVAE takes in an image and outputs categorical distributions over the set of codebook vectors for each latent

The Discrete Problem

Now we just need to sample codebook vectors from each of these distributions, and then we simply feed them to the decoder. This is easy, right? Well, there's actually one big problem with this procedure: you can't backpropagate through sampling from a categorical distribution.

In general, deep learning doesn't play too well with discrete bottlenecks. VQ-VAE approached this problem and came up with a nice set of tricks involving vector quantization; dVAE uses a very different set of hacks.

To solve the discrete sampling problem, dVAE relaxes the bottleneck, allowing it to output vectors anywhere in the convex hull of the set of codebook vectors. This relaxation can be tuned by a hyperparameter τ, which approaches discrete sampling in the limit $\tau \to 0$. So by annealing τ over the course of training, the model is able to effectively approach learning from a discrete latent distribution.



a simplified and messy 2d visualization of the convex hull of a set of codebook vectors

The Gumbel Softmax Relaxation

The specific relaxation that dVAE uses is called the Gumbel Softmax Relaxation. I think it's a pretty nice solution to this problem, so let's dive into the details.

Recall, our distribution over the set of k codebook vectors is $q(e_i|x)$, where e_i is the ith codebook vector. One way to sample a latent from this distribution is

$$z = \text{codebook}[\text{argmax}_i[g_i + log(q(e_i|x))]]$$

where each g_i is an identical, independent sample from the Gumbel distribution, and codebook[i] looks up the vector at the ith index in the codebook.

We can not differentiate through that argmax, so the Gumbel softmax relaxation instead replaces the argmax with a softmax. Sampling from this softmaxnow produces a set of weights, y_i, over the set of codebook vectors:

$$y_i = \frac{e^{\frac{g_i + log(q(e_i|x))}{\tau}}}{\sum_{j=1}^{k} e^{\frac{g_j + log(q(e_j|x))}{\tau}}}$$

The sampled latent vector is then just a weighted sum of these codebook vectors:

$$z = \sum_{j=1}^{k} y_j e_j$$

Success! We can now differentiate through the sampling operation with respect to the encoder outputs, $q(e_i|x)$. This is because:

- This new equation is differentiable everywhere.
- The random sampling is coming from g_i: an external variable that is independent of our encoder's output.

This method of re-parameterizing a random sample in terms of an external variable is referred to as the reparameterization trick. The trick doesn't work for discrete distributions, so we had to

convert our original categorical distribution to a similar continuous one in order for this to work.

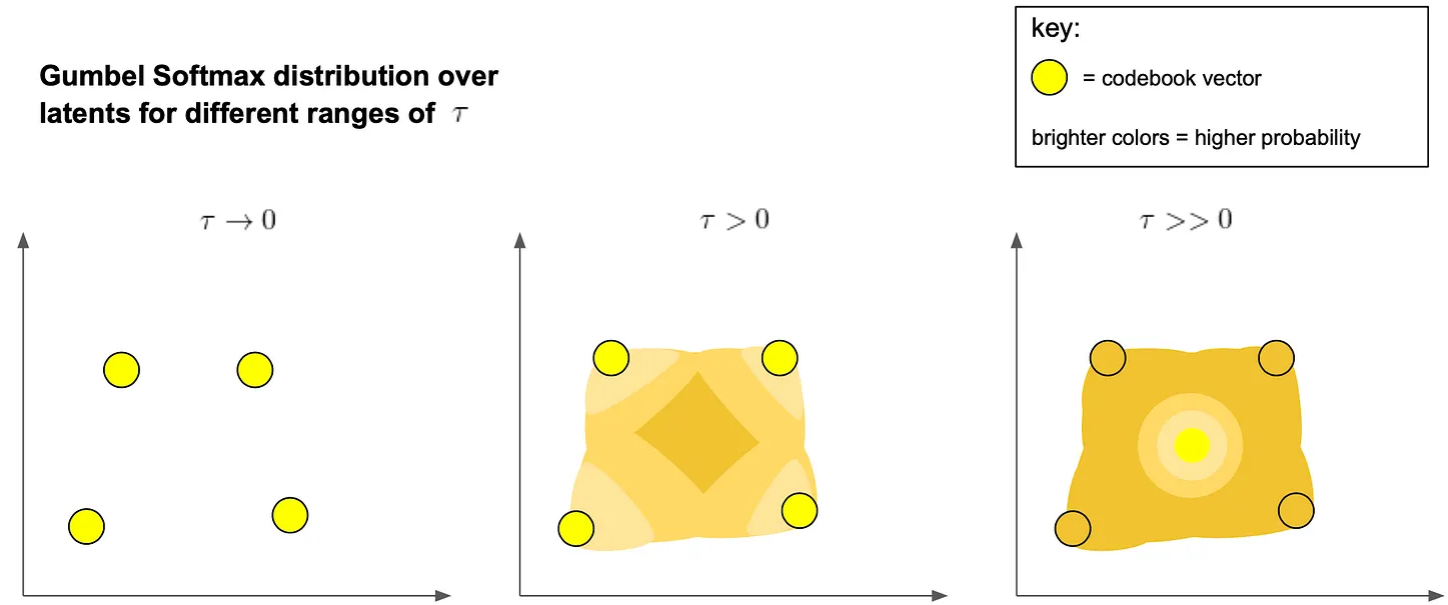
To see how this continuous distribution is similar to our original categorical distribution, notice how the hyperparameter τ functions in the Gumbel Softmax formula. As this parameter approaches zero, the weights y_i become more and more sharply peaked around the maximum

$$g_i + log(q(e_i|x))$$

eventually approaching the set one-hot argmax weights, which would be equivalent to the categorical distribution

$$z = \text{codebook}[\text{argmax}_i[g_i + log(q(e_i|x))]]$$

On the other hand, for large values of τ, the Gumbel softmax will approach a deterministic distribution, which places probability 1 on sampling the centroid of the set of codebook vectors.



Putting It All Together

Now that we can differentiate through the bottleneck of this dVAE, reconstructing an image, involves just two steps:

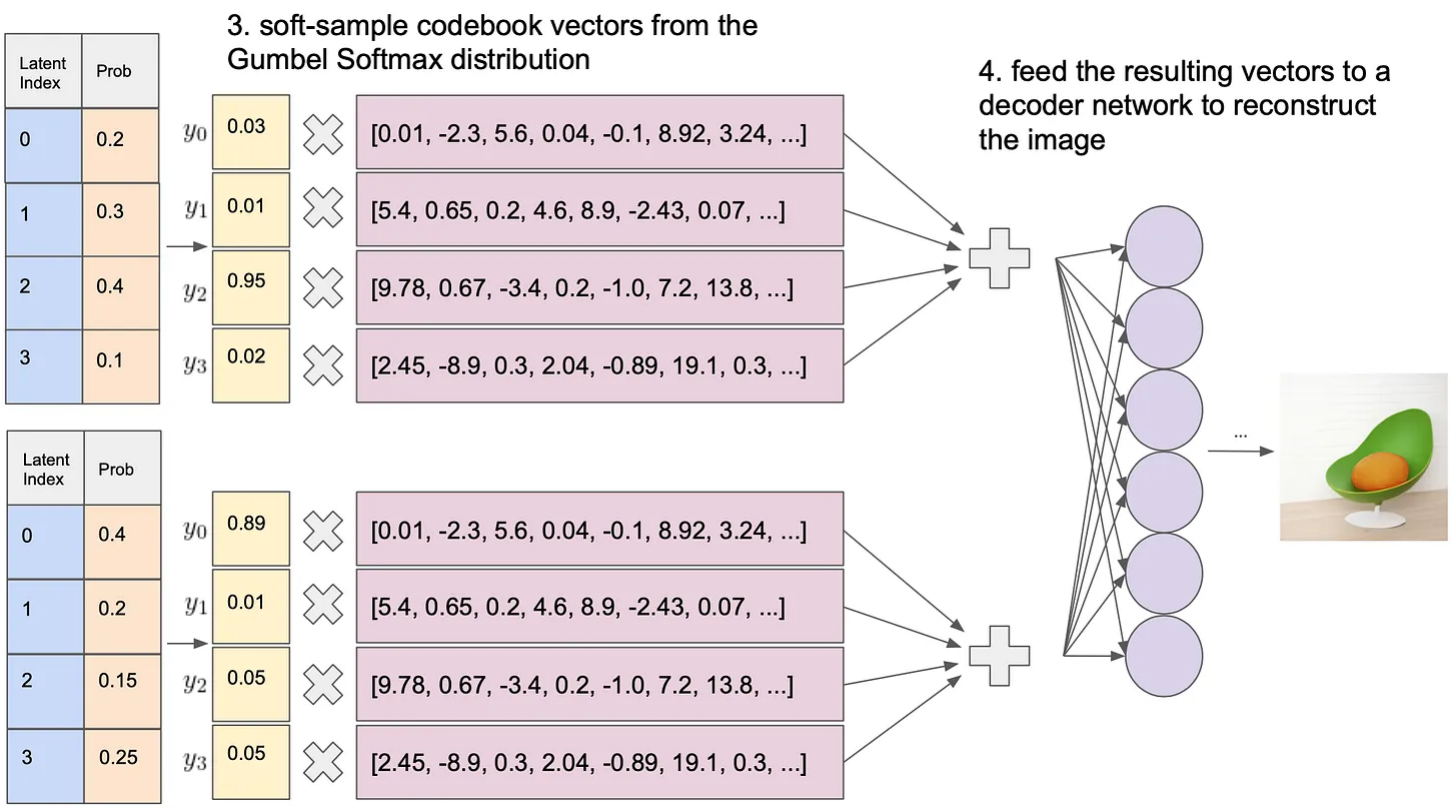
1. Sample codebook vectors from the relaxed posterior.
2. Feed these vectors into the decoder network, which will reconstruct the input as well as it can, just like in VQ-VAE.

And to train the model, we just minimize the standard VAE objective (referred to as the evidence lower bound or ELBO in literature):

$$-E_{z\sim q(z|x)}[\log(p(x|z))] + KL(q(z|x)||p(z))$$

Here $p(x|z)$ is the decoder's reconstruction of the image, and the samples $z\sim q(z|x)$ are taken from the Gumbel softmax relaxation.

Lastly, $p(z)$, the prior on the latents, is just initialized to a uniform distribution over all the codebook vectors. And as we'll see in the next section, the second phase of DALL-E's training involves updating this prior with a transformer model, which ultimately further minimizes this loss function.



sampling codebook vectors from the Gumbel softmax distribution and then passing them to the decoder to reconstruct the original, encoded image

Overall dVAE has the same goal as VQ-VAE: they both try to learn a discrete latent representation for complex data distributions, like the distribution of natural images. Each method approaches the problem in its own unique way though. VQ-VAE uses vector quantization, whereas dVAE relaxes the discrete sampling problem to a continuous approximation. While each technique has its own set of tradeoffs, they both seem like equally valid and equally successful approaches to this problem at the end of the day.

DALL-E: The Big Picture

Now that we understand the specific details behind DALL-E's autoencoder, let's take a brief step back and look at the bigger picture, so that we can better understand the exact role of that DALL-E's transformer plays in its architecture.

fun fact: "DALL-E" is meant to be a combination of "WALL-E" and "Salvador Dalí"

DALL-E's end goal is to learn the conditional distribution of images given some string of text; this is $p(x|y)$ where x is the image and y is the text prompt.

(NOTE: DALL-E actually goes a step further and computes the joint distribution between text and images, but for simplicity we will just consider the conditional distribution here.)

Once we've learned this distribution, we can input some text prompt like "an armchair in the shape of an avocado" and then generate a variety of accurate images by merely sampling from $p(x|y)$.

But how does this sampling really work? And how exactly should we learn $p(x|y)$ in the first place?

Well the first question really depends on what factorization of $p(x/y)$ that we choose to learn. So we can't really answer the first question without forming an approach to the second.

So ... how should we learn $p(x/y)$?

Well first we need a huge dataset of natural text-image pairs; DALL-E specifically was trained on 250 million of these pairs scraped from the internet. Now we just need to train some enormous model to predict images given some text prompt. And tada! Out pops DALL-E!

Practical Problems

It all sounds a little easy, but really there are so many practical challenges in getting this to actually work.

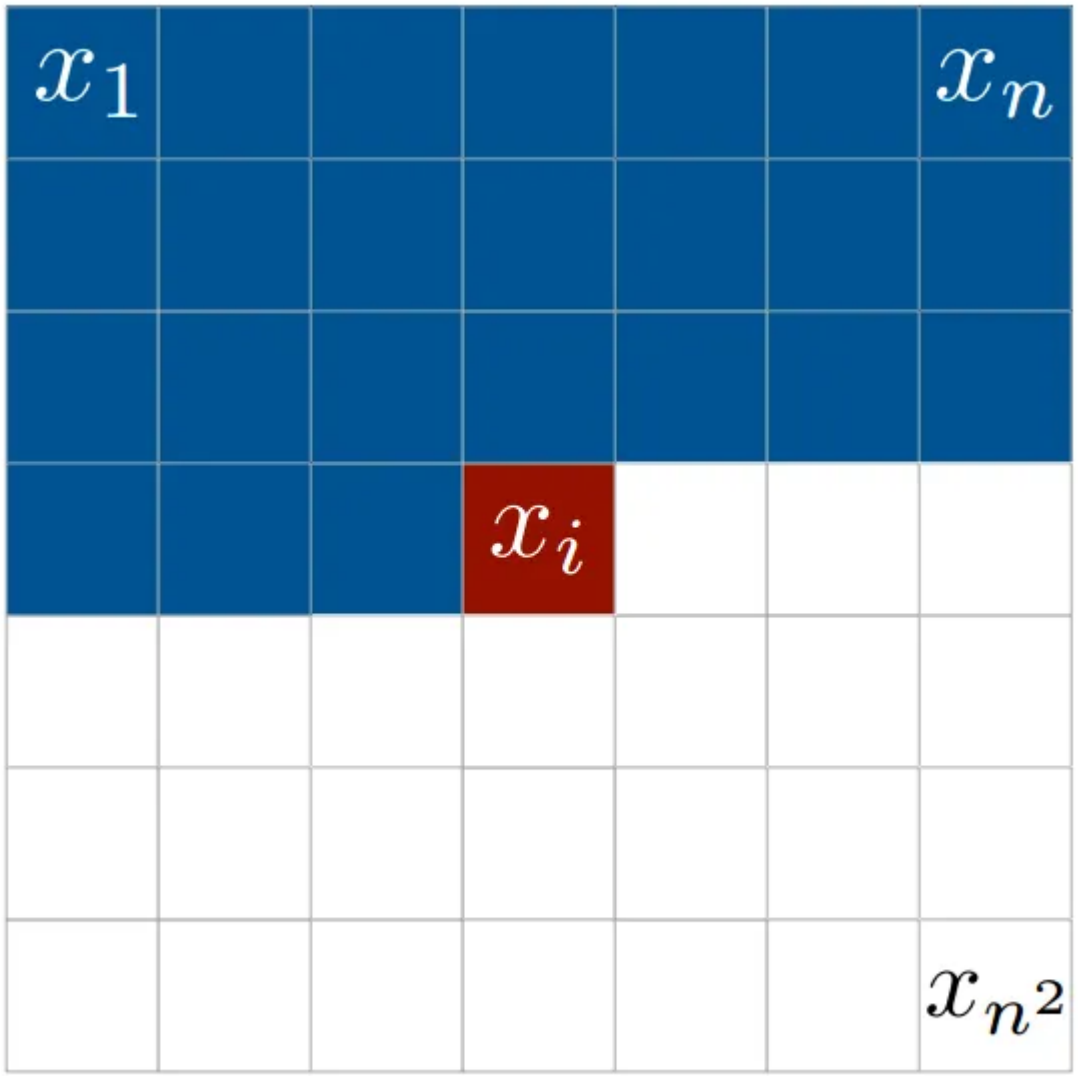
First of all DALL-E has 12 billion parameters; this model is enormous (not GPT-3 large, but still enormous). The compute and scaling infrastructure needed to train a model like this is something that few companies can afford. A good portion of the DALL-E paper is actually devoted to describing all the distributed training tricks they had to use to get this to work; it really sounds like a lot.

Secondly, a simple approach to predicting images from text just isn't practical. Consider the simplest possible approach to this task:

1. Concatenate the set of text tokens with the unrolled set of pixel values in a corresponding image (typically unrolled top left to bottom right).
2. Given this sequence of text and pixel values, we can factor the distribution $p(x/y)$autoregressively:
   $p(x/y)=p(x_1,x_2,x_3,.../y)=p(x_1/y)p(x_2/x_1,y)p(x_3/x_1,x_2,y)...$
   Here x_i is the ith pixel value in the unrolled image.

3. We now estimate $p(x|y)$ by running maximum likelihood estimation on any autoregressive sequence model (e.g. LSTM or Transformer) over each of these $p(x_i|x_{i-1},x_{i-2},...,x_2,x_1,y)$ factors.

   That is to say, we want to train a model to predict the next pixel value in an image, given some text and all previous pixel values.



predicting pixel xix_ixi given the previous (up and to the left) pixels in an image. (image source)

In theory this approach would be elegant: it's so incredibly simple. But what we've just described is actually a very impractical approach because DALL-E works with quality 256x256 RGB images. There are way too many pixels in these images for almost any traditional sequence model to reasonably learn from.

Since DALL-E accepts up to 256 text tokens, the resulting sequence of both text and pixels would total up to 196,864 items long. Any reasonably sized RNN or LSTM would forget most of the early information in this sequence, and if you tried to feed a sequence this long to a transformer, you would most definitely get a cuda out-of-memory error, even on the biggest GPUs out (this is because the $O(n^2)$ memory complexity of attention).

If transformers could actually accept sequences this long, the model might actually do pretty well on this problem, potentially even better than DALL-E, but it would likely require much much more compute than what DALL-E used (which is already a lot).

DALL-E's Factorization

Clearly a more clever solution is needed. Of course we didn't spend the last part of this blog post on nothing: dVAE is the key here. Instead of modeling $p(x/y)$ autoregressively, we factor the distribution in terms of the dVAE's compressed 32x32 grid of discrete image latent variables.

This factorization looks like

$$p(x, z|y) = p(x|z, y)p(z|y)$$

The first factor $p(x/z,y)$ is handled by the dVAE, and the second is modeled autoregressively by a transformer. When unrolled, our grid of latent z's makes for a sequence of length 1024; this is now actually of reasonable length to model with a transformer.

DALL-E does exactly this. It doesn't learn to predict every pixel in the image, rather it predicts a sequence of image latents, which then get turned into a high quality image by the dVAE decoder; this approach makes modeling much more practical.

But it's actually more than just practical, this approach to modeling likely also gives the model some nice inductive biases. By allowing the transformer to model in the latent space, it doesn't have to focus on the small high-frequency details of the image, rather it can focus its powers on the big picture and let the dVAE handle the smaller details.

We can think of the transformer as learning to paint the big picture; it's like the visionary artist. Whereas the dVAE is more just like the image processing software that's tasked with rendering this vision into HD.

> NOTE:
> although this factorization technically results in *p(x,z|y)* and not *p(x|y)*, optimizing for the first one can be shown to be equivalent to optimizing for the second one under the VAE objective:
>
> $$-E_{z\sim q(z|x)}[log(p(x|z))] + KL(q(z|x)||p(z|y))$$
>
> The distributions *q(z|x,y)* and *p(x|y,z)* are modeled by the dVAE's encoder and decoder networks respectively. While the transformer models *p(z|y)*: the one term which directly combines text variables with image variables.
>
> And to contextualize all of this: both the dVAE training and the transformer training minimize this loss. One can see this as a training procedure with two separate phases:
>
> 1. the dVAE is trained to minimize this loss with *p(z|y)* set to a uniform distribution

> 2. the dVAE networks are frozen and $p(z|y)$ is learned via-maximum likelihood estimation with a transformer, further minimizing this loss.
>
> In the DALL-E paper they actually mention running early experiments where they trained both the transformer and dVAE at the same time (i.e. without making explicit uniform assumptions on $p(z|y)$at any point), but they claim that doing this didn't improve performance any.

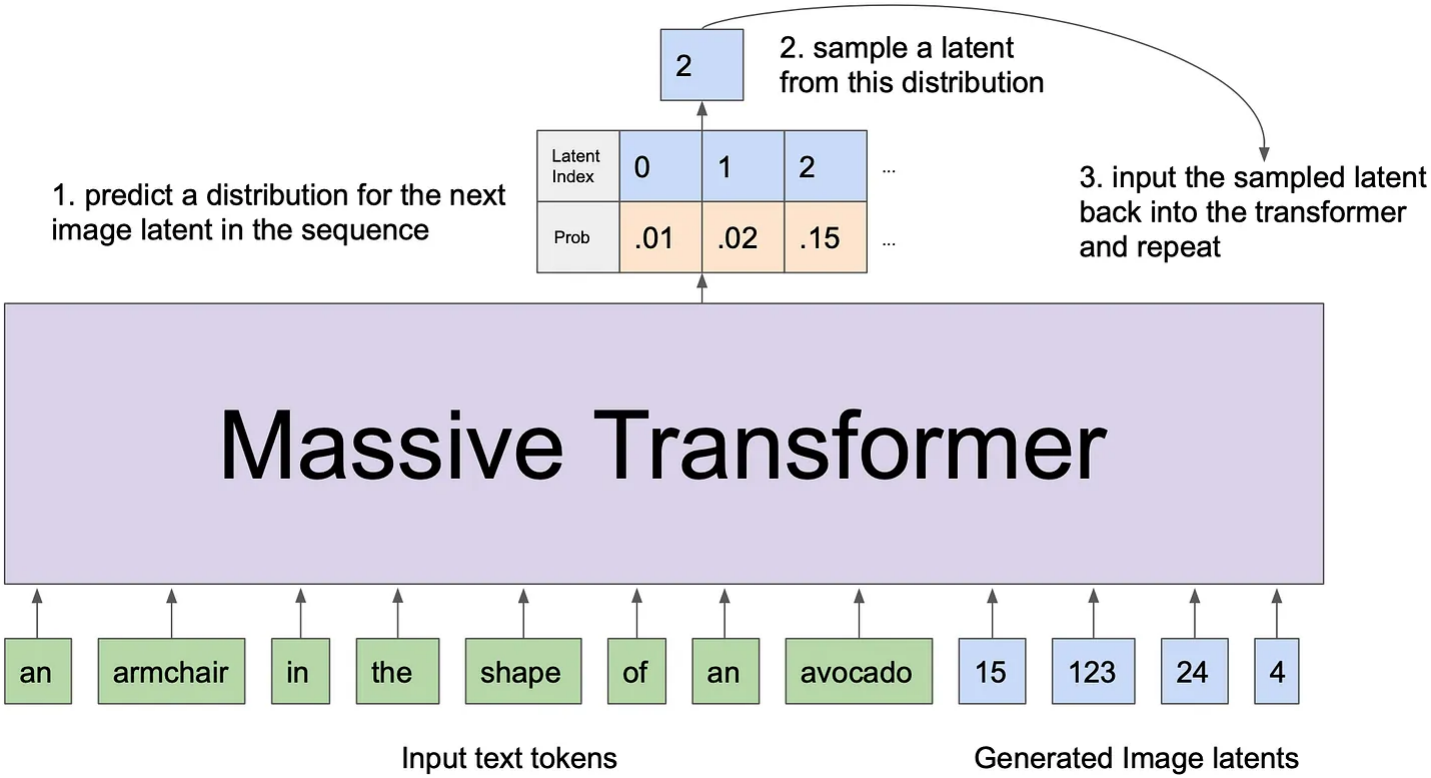
Sampling From a Trained DALL-E

Now that we know what training looks like and how exactly DALL-E factors $p(x|y)$, we can look at how exactly sampling images from a fully trained DALL-E works.

It's quite simple really:

Say we ask the model for images of "an armchair in the shape of an avocado".

First we would input the sequence of words "an armchair in the shape of an avocado" into the transformer, and then we sample a subsequent sequence of image latents from the transformer in an autoregressive fashion. This looks something like:



And to actually render an image from this generated sequence of latents, we just grab the dVAE codebook vectors associated with each discrete latent and feed these to the dVAE decoder. This will render our avocado chair in HD !

5. out pops an image of an avocado chair !

4. feed codebook vectors into the dVAE decoder

look up vectors in dVAE codebook

15 123 24 4 2 6 ...

transformer generated image latents

We can also generate multiple images of avocado chairs by just repeatedly sampling new sequences of latents from the transformer.

a bunch of different avocado chairs sampled from DALL-E (image source)

This basically completes our description of the modeling details behind DALL-E !

So if you happen to have a massive GPU farm in your backyard and access to a dataset of 250 million natural text-image pairs, you can go home and train your own DALL-E model this weekend (or actually, it probably takes a few weeks to train, idk how many GPUs you have at home).

A Transition

We spent the first half of this blog post looking at how DALL-E encodes images into discrete sequences, and then we looked at the big picture of how it is able to learn from these sequences to generate new images from a natural language prompt.

Now that we have a big picture view of how exactly the transformer model fits into DALL-E, we can dig into the more interesting questions of "how is it so good?" and "why is the transformer able to learn so much about language and images?"

The remaining sections of this blog post will be slightly less technical and more exploratory and philosophical than everything that has come before. The goal here is simply to build some intuitions for how exactly a model like DALL-E could come to encode so much rich information about our visual and linguistic worlds.

The remainder of this blog will be structured as follows:

- First we will start from the very basics and look at the history of language modeling. This will help us understand the specific historical problems in sequence modeling that the transformer model solves. We will see why transformers are so much better than previous approaches, and what exactly the transformer's capabilities are. All of this stuff about language modeling is perfectly transferable to DALL-E.
- Next we will look at what happens as transformers get scaled up and up; this should help us understand why DALL-E is able to express so much knowledge about our world.
- In an attempt to make some sense of what's really happening with increasing scale, we will take a detour through lossless compression and see how the transformer's training objective can be directly linked to compression. We will look at what exactly this could mean for DALL-E's learning and what intuitions we can gain from this.

- Lastly, after considering all of this, we will survey the hype around these massive transformer models and also look at some of the theoretical limitations that have been proposed for these models.

A Brief History of Language Modeling

The fundamental problem that DALL-E's transformer is tasked with solving is simply that of predicting the next item in a sequence given all previous items in the sequence.

DALL-E, in particular, operates over sequences of text followed by image latents. But really, we can substitute this specific distribution of sequences for pretty much any other type of discrete sequence and apply the exact same methods to solving the problem.

For example, we could remove the image latents from DALL-E's sequences and just model the text using the same fundamental algorithms. This particular problem of modeling sequences of text is called language modeling, and it has a much longer history in relation to AI and machine learning than DALL-E's multimodal modeling does.

In recent years transformers have completely taken over language modeling. These models are miles better at generating sequences of text than anything that has come before. Just look at some of these insane outputs from OpenAI's recent GPT-3 language model if you don't believe me.

So in order to understand what exactly makes DALL-E's transformer so powerful, we're going to take a brief tour through the history of language modeling. We'll look at the different types of models that have been applied to this problem historically, what issues each of these models faced, and how exactly transformers improve upon all of these previous approaches.

The Two Key Factors for MLE

Simply stated, the goal of language modeling is to compute the probability distribution of language.

And just like any other sequence, this probability distribution can factored autoregressively:

$$p(x_1, x_2, x_3, \dots) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)\dots$$

where x_i is the ith word or token in the sequence of text.

All we need to do is estimate each of these factors using maximum likelihood estimation (MLE) on some large dataset, and suddenly we have a model that can predict sequences.

It is not quite that simple though. The power of MLE really depends on two key factors: the specific model parameterization that we maximize over, and the particular dataset that we use as a proxy for the true distribution that we want to estimate.

In a perfect world, we might have access to infinite computational resources and an infinite supply of data, so we could estimate all events in the distribution exactly, without ever needing to parameterize our estimation in terms of some imperfect model that we optimize over some crappy, finite dataset that only sort of resembles the true distribution of data. Obviously, this is not possible in the real world, especially for complicated data distributions like language or images.

So instead, we just perform MLE over a distribution parameterized by a reasonably expressive model on some dataset that is hopefully big and diverse enough to accurately approximate the real world.

Obviously capturing a good dataset is super important here, but the choice of model is actually just as critical.

For example, if we were working with continuous data (which we are not in the case of DALL-E's transformer or in the case of language modeling, but just consider this for simplicity), a simple model might parameterize the space of gaussians, who's means and variances are just linear functions of the input. Or instead, we could use a much more complicated space of models, like a mixture of gaussians, who's parameters are generated by some complicated, non-linear function of the input data.

The second, more complicated case here would likely be able to model the true distribution much more accurately than the first given enough data.

I wanted to make this point super clear because basically all of the advances in language modeling that I'm about to describe revolve around these two factors: data and model. Each revolution in language modeling can be described by an improvement in the model used and also usually an increase in the dataset size and diversity. This is especially true of the transformer revolution, which as we will see, can be effectively described as a big improvement in model architecture alongside huge increases in dataset size.

N-Grams

Let's begin by looking at the simplest possible approach to language modeling: N-grams.

Training N-grams is quite simple:

- Get a large corpus of text and count how many times each unique sequence of N consecutive words occurs in the corpus.
- Use these frequencies to compute the conditional probability of the Nth word in a sequence given all previous N-1 words.

Computing probabilities for a 2-gram model

words	counts	probability of the second word conditioned on the first
The dog	341	0.39
The cat	543	0.61
I'm running	187	0.17
I'm eating	890	0.83

$$p(dog|the) = \frac{p(the\ dog)}{p(the)} = \frac{341}{884} = 0.39$$

$$p(cat|the) = \frac{p(the\ cat)}{p(the)} = \frac{543}{884} = 0.61$$

Of course there are some edge cases, like what to do when you want to get probabilities for sequences of length < N words, but this is basically all there is to training an N-gram model.

Now that we know how it works, let's look at some example sentences generated by an N-gram model trained on Shakespeare's writing:

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, ’tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv’d in; –It cannot be but so.

outputs from N-gram models trained on Shakespeare’s writing (image source).

As you can see, the 1 and 2 grams hardly make any sense, but the 3 and 4 grams actually have some brief moments of coherence and sound a little bit like Shakespeare. Overall, none of these outputs are great though.

To make sense of why this isn’t working so well, let’s look at the assumptions that this model is making.

First of all, N-grams, by their nature, assume that each word is independent of all words N or more back in the sequence. Of course, this is a naive assumption because often in language, a word can depend on stuff that happened paragraphs ago. So for small N this assumption is quite limiting.

There are also big problems with using too large of a value for N. Obviously increasing N, gives the model the ability to consider more context, therefore resulting in more coherent outputs. But this comes at a big cost: as we increase N, there are more and more possible sequences of grammatically correct words and it is likely that many of these sequences only appear in our training data just once or twice if even at all. With fewer counts for each sequence, we are less

reliably able to estimate probabilities and may even assign probability 0 to perfectly valid language. This is a huge modeling problem.

For years, people got around this by using various probability smoothing hacks or falling back to 2 or 3 grams when the higher order grams didn't have enough data. But none of these tricks really solve the fundamental problem: N-grams treat each sequence of words as its own independent entity. It doesn't have any notion of similarity between sequences; each sentence is merely an index in a table, on its own island, completely disconnected from all other sequences.

For example, N-grams wouldn't consider any sort of similarity between the sentences "my cat had pancakes for breakfast this morning" and "my cat ate pancakes for breakfast this morning". Each sequence would be counted separately in its own table entry, even though these sentences are effectively saying the same thing. Clearly something is wrong with this approach, a good model should be able to take advantage of the fact that some sentences are similar or contain similar information.

Truthfully, this simple N-gram counting approach would be fine if we had infinite data. In fact, this method of counting frequencies is a perfectly unbiased MLE estimator of the true distribution, under the infinite data limit.

But sadly this is the real world, and we don't have an infinite supply of data to work with, so we need to be more efficient. A simple way to improve efficiency is to take advantage of some structural information in the data (e.g. representing similarities between sentences and not treating every single sequence of words as its own island).

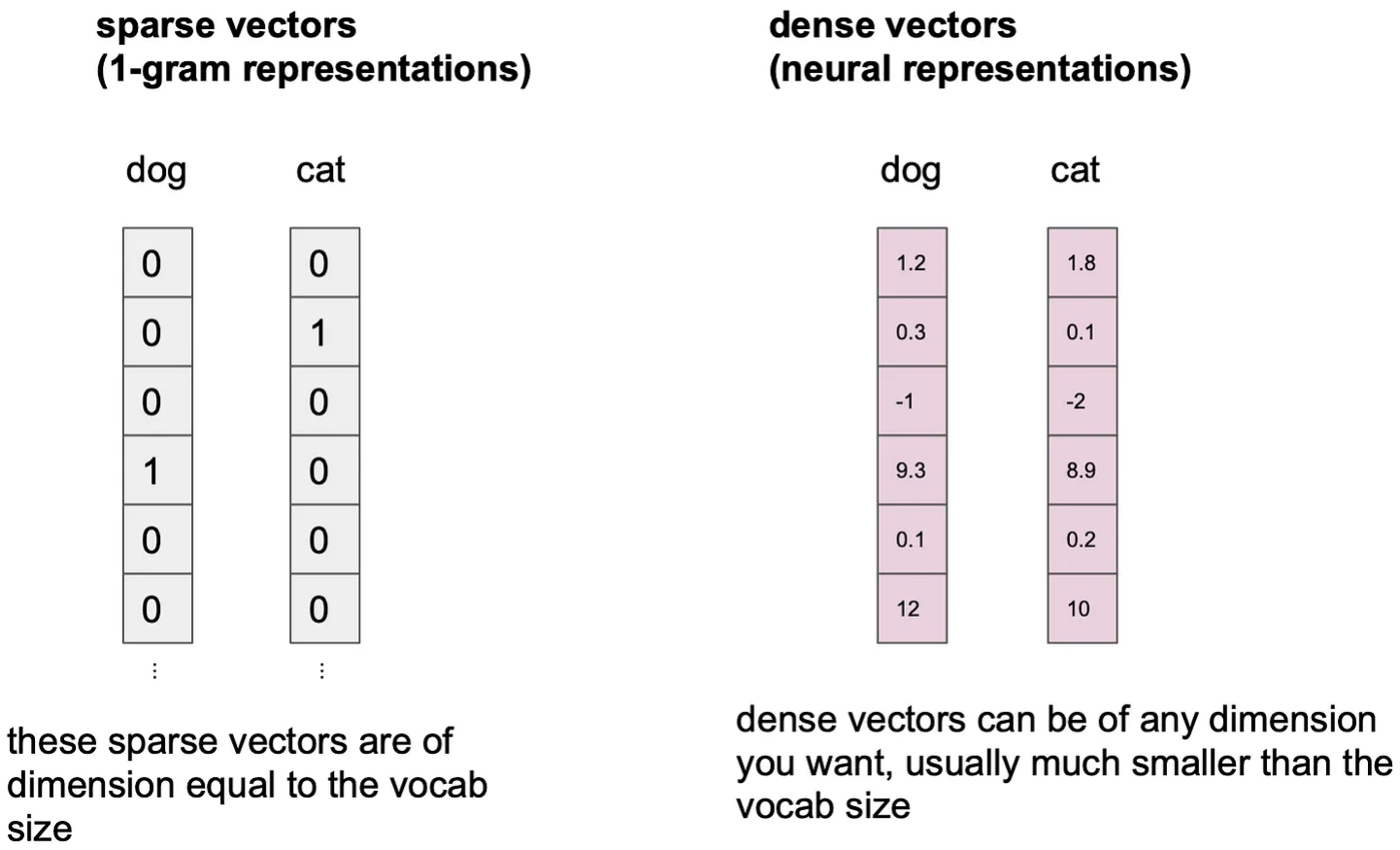
From Sparse to Dense Representations

Clearly N-grams are not it. A good way to think about the problem with this model is to consider each N-gram as its own sparse, 1-hot vector. Each sequence of words in this model can be seen

as a huge 1-hot vector, of dimension equal to the number of possible sequences of N words, with a 1 in the index corresponding to a given sequence and 0s everywhere else.

With this representation, the distributions output by the model can be seen as a simple linear function of the elements in a given N-gram vector. This is like the simplest parameterization possible!

So why limit ourselves to representing linear functions of giant one-hot vectors? Maybe instead we should parameterize our model in terms of much smaller, more dense vectors for each word or phrase. Perhaps this would allow the model to express richer correlations between words and phrases by placing similar things next to each other in the vector space. This might even allow the model to generalize to longer sequences of words or sequences it hasn't even seen before. This is exactly what neural language models do.

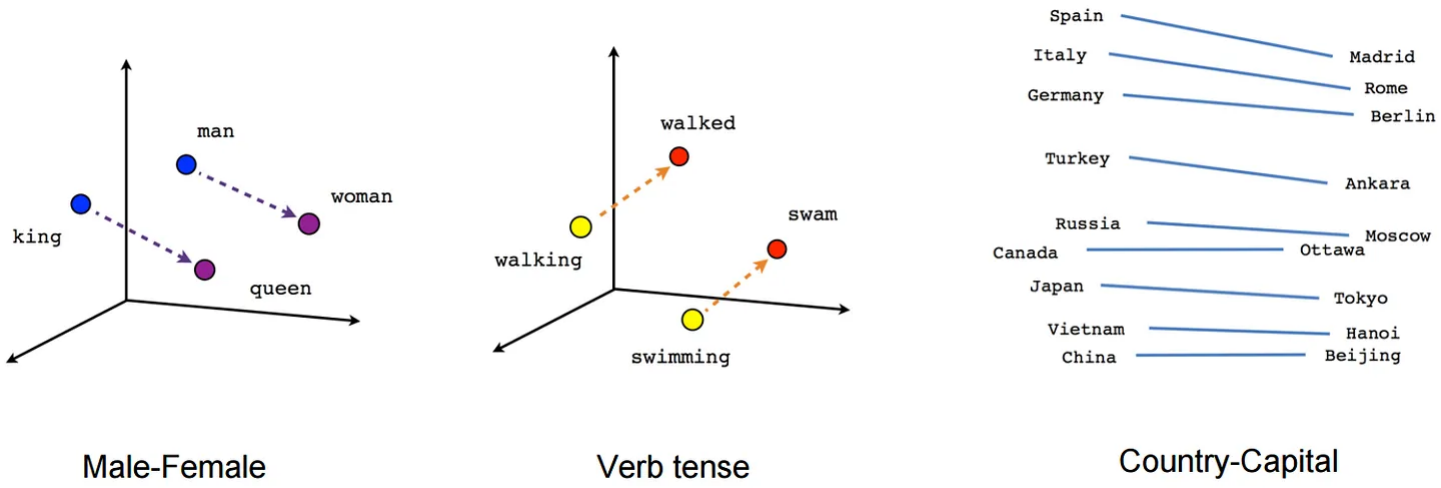


by representing words densely, neural language models can express similarities between words and phrases, making the models more expressive than N-grams.

For example, we could learn a 64 dimensional vector for each word in a vocabulary and then use these vectors to help us predict the next word in a given sequence. To do this, we can just take sequences of N words, concatenate their word vectors and then pass this concatenated vector through an output layer to predict a distribution over the next word in a sequence. We can then train these dense vectors with stochastic gradient descent on an MLE loss function. We've just described how neural N-gram models work!

The word vectors learned by these types of models, can actually have some pretty surprising properties. Not only will the model learn to place similar words next to each other, but it can also represent some non-trivial relationships between words in the differences between vectors. For

example, famous word analogy problems like, `king - man + woman = queen`, have been found in these types of dense neural representation vectors.



some examples of word relationships that can be found in neural word vectors (image source)

So by simply allowing the model to represent dense vectors for words, we have gained so much ground over simple N-grams! Sequences are no longer on their own islands; we can now represent similarities between words and phrases. We can also generalize to unseen sequences; we don't have to assign 0 probability to stuff we haven't seen before.

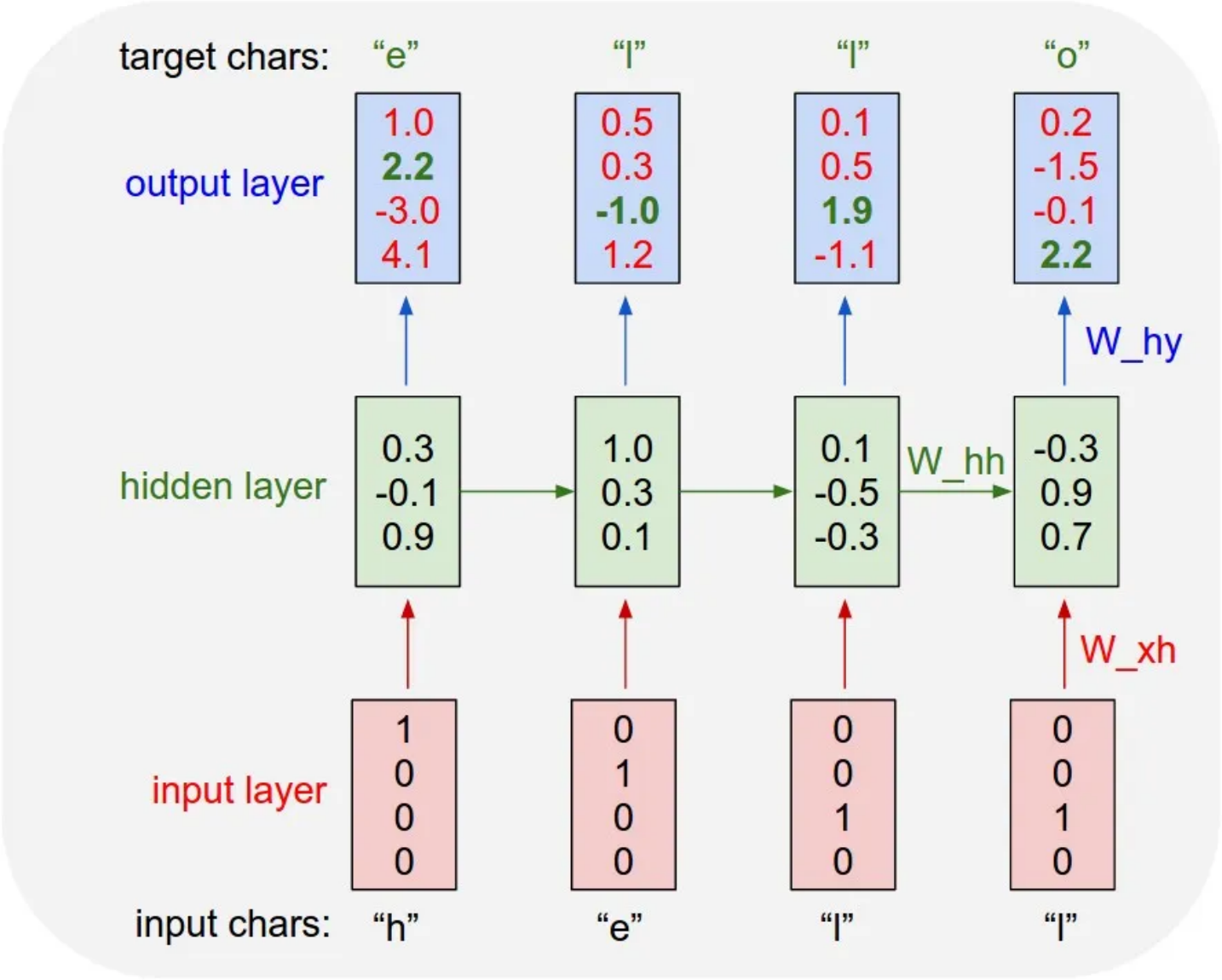
We've solved many of the original problems with naive N-gram models, but some still remain, like the independence problem. We're still treating words as if they were independent of other words that occur much earlier in a text.

Some new problems have also arisen with this approach. For example, by only representing a single vector for each word, it's hard to express different meanings for words in different contexts. Like the word "mouse" could be used to refer to either the animal or a computer mouse, which one all depends on the context.

This particular problem didn't exist in our previous, naive N-gram approach because our model didn't make any assumptions about the structure of the data; now we are making too many structural assumptions.

RNNs

We can actually solve this contextual problem by not only learning a dense vector for each word in our vocabulary, but also by having our model learn to generate a context vector for every word in a given sentence. Context vectors should ideally represent the meaning of a word within its context. These contextual representations could then be used to help our model better predict the next word in a sequence. This is exactly what recurrent neural network (RNN) language models aim to do.



an illustration of an RNN operating on sequences of characters (image source)

I won’t go into all the details of RNNs, but essentially these models take in dense vectors representing the words in a sentence one by one. And for each new word that it processes, the model updates a context vector with information from the current word. In theory, this context vector contains information about everything the model has seen so far in the sequence, allowing it to consider all the context when making a prediction.

In addition to keeping a context vector, RNNs are just generally more expressive than the neural N-gram models we described earlier. The distributions produced by RNNs are usually parameterized by complicated nonlinear functions of the inputs, this is in contrast to the simple linear parameterization of our neural N-grams.

The ability to consider words in context and represent complicated nonlinear functions of the inputs, makes for an extremely effective combination (some might even say unreasonably effective). Just to see how much better this is than the N-grams we saw earlier, lets look at some examples of text produced by an RNN trained on the works of Shakespeare:

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

outputs from an RNN trained on Shakespeare's writing (image source).

While the model still makes some obvious mistakes, this actually looks like Shakespeare. It definitely nails the style, and is reasonably coherent. This is a huge step up!

In general, RNNs make fewer assumptions about the structure of the data than neural N-grams, but there are still some problems with these models. Most notably, we haven't completely solved the N-gram independence issue yet. While RNNs are not explicitly tied to any particular sequence length, and can theoretically consider sequences of arbitrary length, in practice these models tend to forget most information that's about 100 to 500 steps back in the sequence (the exact number really depends on the size and type of RNN; LSTMs are much better than vanilla RNNs, but they still usually top out at around sequences of length 500). This is still big progress, as our earlier N-gram models would likely have trouble with sequences of length much smaller than 100.

The reason that RNNs forget stuff stems from the fact that it has to compress all context into a single vector. This is somewhat of an informational bottleneck. And as we'll see next, transformers are able to improve on RNNs by fixing this very issue.

Transformers and Efficient Information Passing

How do transformers solve the RNN information bottleneck problem? And how much does this really improve things?

So I won't describe all of the details of transformers here, but you can think of a transformer as a model that holds onto and updates a separate vector representing each word in its context. Unlike RNNs, this is not just one vector that is constantly getting overridden at each time step, rather transformers keep an entierly seperate vector for each word.

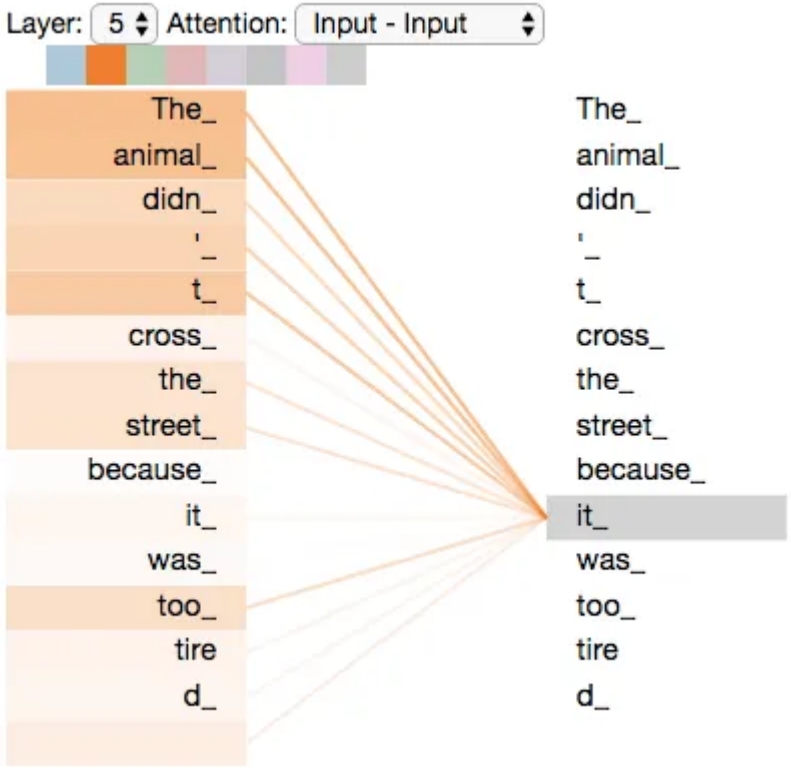
Specifically, at every layer of the transformer, each word vector passes through two processes:

1. the word's context vector is scored for similarity against all the other word-context-vectors, producing a distribution of "attention" weights over the set of words. Using these attention weights, a weighted sum over context vectors is taken, and this weighted sum becomes the new, updated context vector for the word.
2. The new context vector is then passed through a couple feedforward layers with non-linearities between them

The second step here is important because it allows the model to parameterize complicated nonlinear functions of the input, just like RNNs.

But the first step is what really sets transformers apart; it solves the RNN bottleneck problem. The model doesn't have to compress all contextual information into just 1 vector anymore, rather it can spread the information out across the sequence. In fact, the model can look back arbitrarily far in the sequence without ever having to lose any older information.

The figure below is a visualization of the attention weights learned by a transformer:



a visualization of transformer attention weights. Darker colors indicate stronger attention weights. (image source)

Here the model has compared the vector for “it” to all the other words’ vectors in the sentence to determine how much each word is relevant to the contextual meaning of “it”. And as you can see, the model attends strongly to “the animal”, which is exactly what “it” is referencing in this sentence. Clearly the transformer has learned something interesting about how to incorporate context in its representations.

Ok but how much better is this than an RNN really? Well, here’s an output from OpenAI’s very powerful GPT-2 model:

SYSTEM PROMPT (HUMAN-WRITTEN)	*In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.*
MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)	The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez. Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns. While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English. Pérez stated, "We can see, for example, that they have a common 'language,' something like a dialect or dialectic."

(image source)

I think this output is actually pretty impressive. The model produced an incredibly coherent story about unicorns in the Andes mountains. It creates characters and writes interactions between them. The model also seems to be very good at incorporating context from several paragraphs before. This is another huge step up from the RNN Shakespeare example we saw before; clearly something is working here!

We can see that transformers are better than all the other approaches described. In addition to the improvements that RNNs and neural methods made on N-grams, the transformer's attention mechanism basically fixes the single biggest issue with RNNs: the information bottleneck.

But these models are not perfect. Although they can incorporate arbitrary context, it can quickly become computationally infeasible to use transformers on longer sequence lengths because computing the attention weights costs O(n2)\mathcal{O}(n^2)O(n2) time and memory, where nnn is the sequence length. There is lots of recent research that tries to fix this exact problem though, so it may be possible to alleviate some of these computational issues.

To summarize what we've learned about transformers through this historical tour:

- By using dense vector representations, transformers are able to express rich relationships between words, phrases, and contexts.
- With attention the transformer is able to consider context arbitrarily far back in the sequence when making a prediction.
- With repeated attention layers followed by nonlinear feedforward layers, transformers are able to express incredibly complex functions of the inputs. This allows the model to accurately estimate complicated real-world data distributions like language or images.
- In total, transformers make for an incredibly powerful class of models that we can preform MLE over on any distribution of discrete sequences, whether it be natural language or multimodal text-to-image sequences.

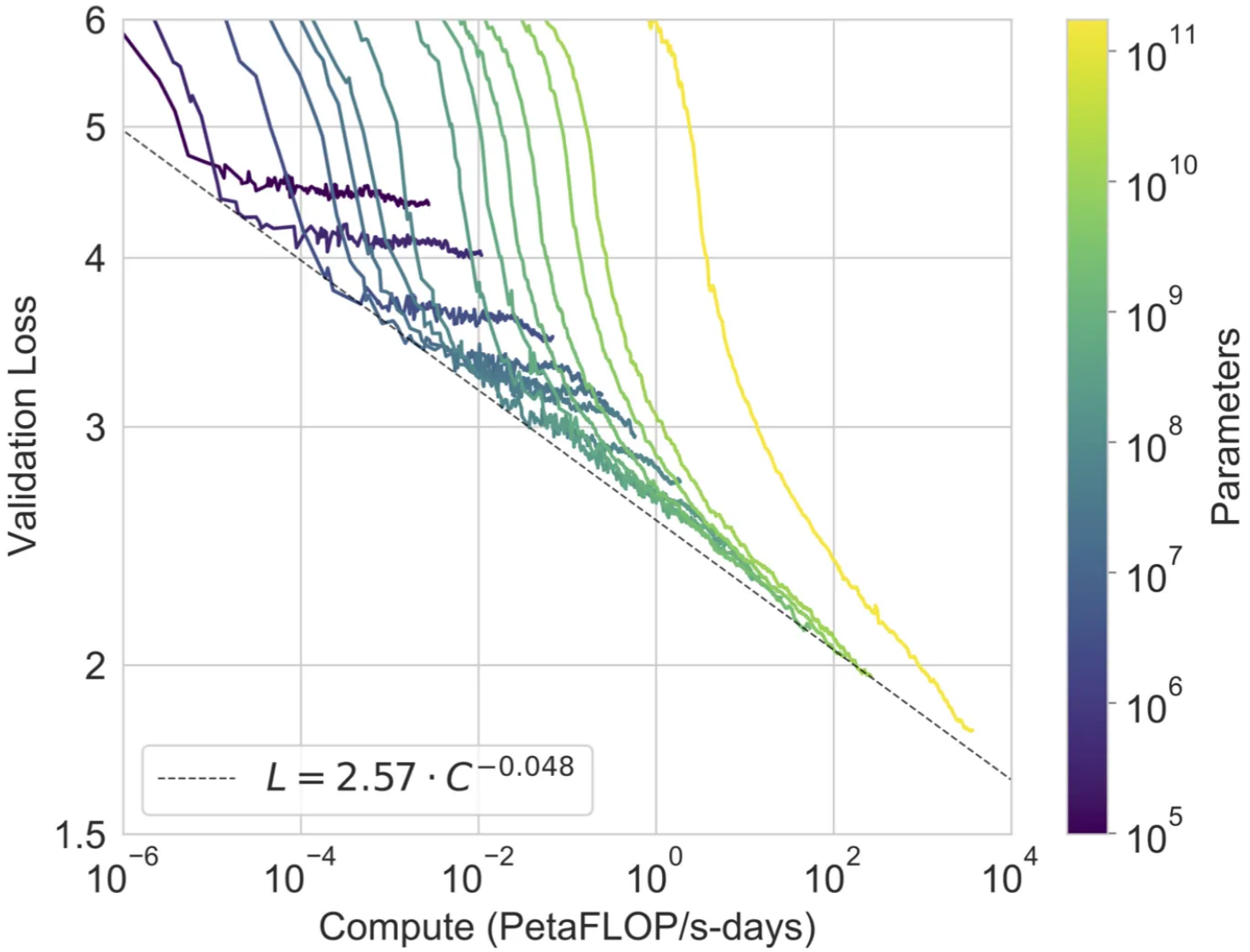
Scaling Laws: Take it to the Moon

In a way, comparing the previous GPT-2 transformer example to the Shakespeare RNN was a bit unfair. GPT-2 is a much much bigger model, in terms of parameter count, than the Shakespeare RNN. The transformer was also trained on orders of magnitude more data and for much longer. In fact, GPT-2 was, at the time, the largest neural network model ever trained.

But the fact that this transformer worked so well at such a large scale is kind of the point. Scaling up RNNs by that much, wouldn't give such dramatically improved results. Transformers scale incredibly well and this is sort of their secret sauce.

In particular, by scaling I mean, things like increasing the number of layers, the size of the layers, and increasing the number of training sequences. Really just anything to make the compute bill go up.

There has actually been a lot of research on looking at how these models scale. And it turns out that there are some simple power laws that can accurately predict how much better a transformer language model will get as you increase compute and data. The paper Scaling Laws for Neural Language Models describes this phenomenon in great detail. In fact, GPT-3 can be seen as an expression of the power of these scaling laws. With GPT-3, OpenAI significantly scaled up the models and datasets from GPT-2 and got consistently better results. The plot below shows GPT-3's validation loss as a function of compute:



(image source)

As you can see, OpenAI trained models of sizes across several orders of magnitude, and they saw the same consistent, predictable trends in the loss. This is actually a bit surprising. These models really do scale! Typically in deep learning, scaling tops out at some point, but it seems we haven't even found the limits to scaling transformers yet.

In fact, alongside all these quantitative gains, came arguably more impressive qualitative gains. OpenAI found that their largest models, especially their big 175 billion parameter model, are

capable of not only generating incredibly realistic text, but also meta-learning from the prompt. By just giving the model a few examples of some question answering or classification task in the prompt, it can then learn to perform this task with accuracy that is often times competitive with some SOTA fine-tuned models (tbh there is some debate about whether it is just pattern matching or really meta learning, no one knows definitively).

Before GPT-3, NLP models had to be fine tuned on reasonably large datasets to perform a specific task well. So the ability to generalize from just a few examples in the prompt, has been a game changer.

To see just how much things can improve with scale, lets look at some examples:

Here I tried to get GPT-3 to write a script for the upcoming Spiderman movie. The bold is the prompt I wrote, and the rest was completed by the model:

Script for Spiderman Far From Home:
Spider Man swung down from the building and landed on the ground with a thud. It was a dark night, but the light from the street lamps and the moon made it easy to see. He looked around, trying to figure out where he was. He knew it was New York, but he wasn't sure where. He saw a sign that said "Central Park West", and a few blocks down was another that said "Columbus Circle". He was in the middle of Manhattan. He was about to start walking when he heard a scream. He turned around and saw a woman being mugged. He ran over and punched the mugger in the face, and the mugger fell to the ground. The woman was shocked at what had happened, but she was grateful. "Thank you so much! I was so scared!" she exclaimed. "It's no problem, ma'am. I'm Spiderman. If you ever need me, just call." Spiderman said. "Thank you. I will." The woman said. Spiderman walked away. He was looking for a place to stay for the night. He decided to swing to the nearest building. He climbed up the side of the building and walked on the ledge. He walked across the ledge to a window. He opened the window and climbed inside. He walked over to the bed and lay down. He fell asleep quickly.

I mean, it's pretty good… it accurately describes the signs in Manhattan, and it details an interaction involving spider man stopping a mugger. It seems to roughly understand the context that I was imagining when I wrote this very small, simple prompt.

I tried an even simpler prompt, with less context, and this is what I got:

Script for Spiderman Far From Home:

The film is directed by Jon Watts and produced by Kevin Feige and Amy Pascal. The film is written by Chris McKenna, Erik Sommers, and Erik Jendresen. The movie will be released on July 5, 2019.

Also Read: Avengers Endgame: Captain America and Iron Man's friendship

Spiderman Far From Home Cast:

Tom Holland as Spiderman:

Tom Holland is an English actor. He is best known for his role as the Marvel superhero Spiderman. He is also known for his role in The Impossible, In the Heart of the Sea, and The Lost City of Z. He will reprise his role as Spiderman in Spiderman Far From Home.

Zendaya as Michelle:

Zendaya is an American actress and singer. She is best known for her role as Rocky Blue on Shake It Up and as K.C. Cooper in the

Obviously there is a bit more ambiguity here; without the first line of the script in the prompt, the model is not being guided as much. And it seems that the model thinks it's writing a Wikipedia page or something. As you can see, it spits out a shocking amount of factual information.

After looking some of these names up, it actually did make up a couple people, but most of this is factually correct. At the end of the day, from a first glance, this output looks completely correct, and that's sort of the point. These models are not trained to output the right answer, they just try to output what they "think" will come next, and oftentimes these two things intersect but not always.

And one last example of the model's in prompt learning capabilities:

The following is a list of companies and the categories they fall into

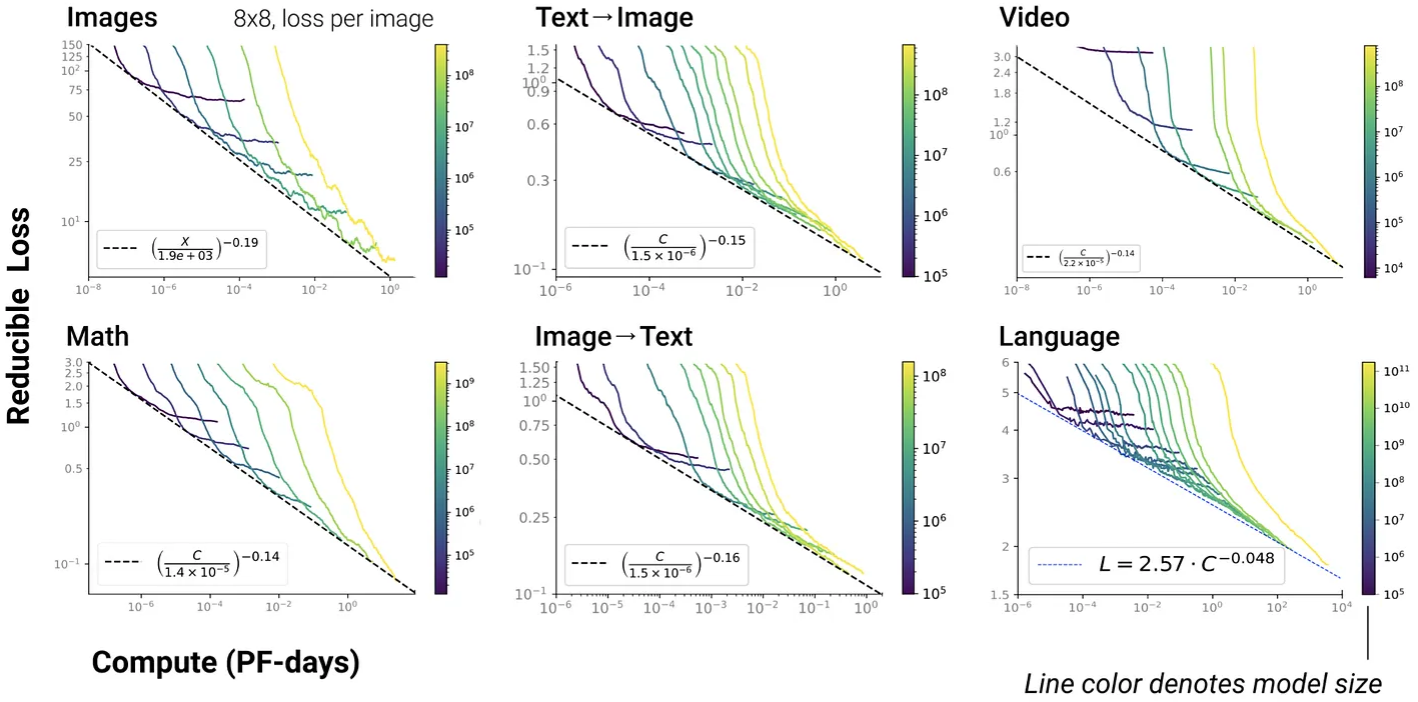
Facebook: Social media, Technology
LinkedIn: Social media, Technology, Enterprise, Careers
Uber: Transportation, Technology, Marketplace
Unilever: Conglomerate, Consumer Goods
Mcdonalds: Food, Fast Food, Logistics, Restaurants
FedEx: Logistics, Transportation
Google: Technology, Search, Enterprise,

As you can see, given just a few examples in the prompt, GPT-3 is able to understand the task at hand and make accurate predictions.

All of these capabilities come from scale. With more scale, the model can hold more factual information in its weights, it can learn more from the prompt, and it can better extrapolate when given little context.

The moral of the story is: as these models get bigger, they become more capable and can represent more and more complicated concepts.

In fact these scaling properties are not exclusive to language modeling: it has been shown that transformers for autoregressive generative modeling, on basically any modality (text, images, vidoes, math, text-to-image, image-to-text), exhibit similar scaling laws. The paper Scaling Laws for Autoregressive Generative Modeling explores this idea in depth. As you can see from the plots below, these scaling laws look very similar across modalities.



([image source](#))

This means that the same type of scaling laws that we've seen with language modeling also apply to DALL-E's transformer!

Intuitively by scaling up DALL-E's transformer, it is able to incorporate more and more facts about things like how mirrors work or what Alamo Square in San Francisco looks like. And all of this knowledge likely combines to enhance DALL-E ability to generate creative things like avocado chairs. The fact that transformers scale so well really is the key to DALL-E's incredible outputs.

A Perspective on Lossless Compression

What is really happening at an increased scale? How is it consistently getting better?

I can't answer these questions exactly, but by taking a closer look at the loss function that these models are trained with, and comparing it with concepts from lossless compression, we can gain some pretty interesting intuitions on these questions.

Specifically, we will first review the basics of lossless compression and information theory. Then we will see how DALL-E's autoregressive loss function can be viewed as a measure of the model's ability to compress information. Lastly, we will look at what this could mean and what intuitions we can gain from this.

Compression?

So how is predicting the next part of an image or the next word in a sequence related to lossless compression?

Well, let's start from the basics. Lossless compression is rooted in the idea that if data comes from a certain distribution, you can take advantage of the structure of this distribution when representing the data in bits.

For example, suppose we have a vocabulary of 4 words "the", "cat", "sat", and "mat", and suppose we collect some information about how often these words occur naturally. The table below shows this data that we've collected:

word:	the	cat	sat	mat
probability:	0.5	0.3	0.15	0.05

Now since, the word "the" is super common, maybe we could represent it with fewer bits than the much rarer word "mat". Under expectation, this would reduce the number of bits we need to represent sentences from this vocabulary.

Basically, we just want to re-map all these words to sequences of bits, such that, under expectation the number of bits per word is as low as possible.

Entropy

In fact, there is a theoretical lower bound on how low we can get our bits per symbol rate without losing any information. This lower bound is the entropy of the distribution. Entropy is just:

$$-\sum_i p_i log_2(p_i)$$

Here we are enumerating over the probability of all events in the distribution p_i.

Just to give some intuition, the entropy is at maximum when the distribution is uniform. This makes sense; in this case, we can't really use any structure from the distribution to compress anything, all symbols are equally likely. On the other hand if all the probability mass is centered on one symbol, then the entropy is 0; theoretically we don't need any bits to represent this because we already know what's going to happen.

In the "cat", "sat", "mat" example above. The entropy is:

$$-0.5 * log_2(0.5) - 0.3 * log_2(0.3) - 0.15 * log_2(0.15) - 0.05 * log_2(0.05) = 1.65$$

So if we are compressing language from the vocabulary above, we can theoretically compress our data to no more than 1.65 bits per word on average without losing information.

Huffman Coding

Now we just need to figure out a way to assign sequences of bits to individual words in a way that comes as close as possible to this limit, while still being able to recover the original data completely.

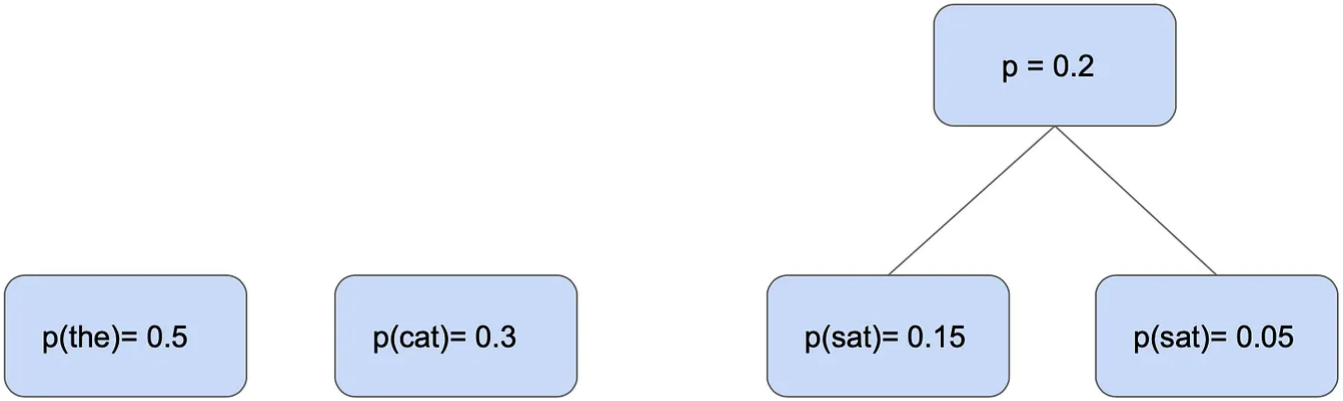
As it turns out, there are a number of algorithms that can take in a probability distribution and determine a near optimal way of assigning bits to symbols under the distribution. Huffman coding and Arithmetic coding are two common approaches to this. Both of these algorithms have nice provable bounds on how close they can get to the entropy lower bound: Huffman coding comes within 1 bit per symbol of the entropy at worst, and arithmetic coding can come arbitrarily close to the optimal entropy. Arithmetic coding is a bit more complicated than huffman coding, so I'll just explain huffman coding for now.

The Huffman algorithm is really quite simple. The main idea is to build a binary tree that describes the sequence of bits needed to generate a given symbol. This binary tree approach has the nice property that no symbol's bit representation will be a prefix of another's, removing all ambiguity from the final bit sequence, making it possible to losslessly encode and decode the bits. This prefix free method of compression is generally called "prefix coding".

We start by building this tree up from the bottom. The leaf nodes of our tree are represented by the set of vocabulary symbols and their corresponding probabilities.

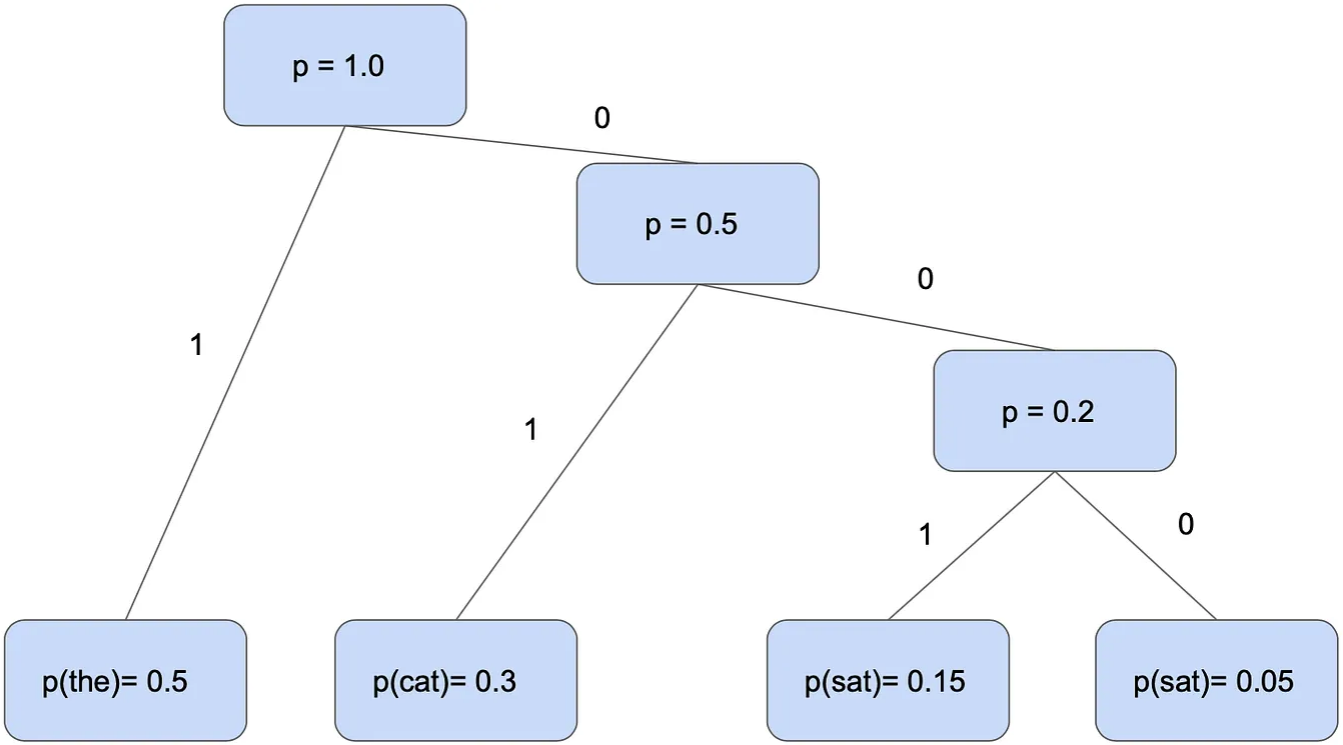
the beginning of Huffman coding, start with the symbols at leaf nodes.

Next we find the two lowest probability symbols and combine them by attaching a parent node that connects them. This parent node now gets a probability that is the sum of its two childrens' probabilities.



the first step of Huffman coding. Combine the two lowest probability nodes, “sat” and “cat” by attaching a parent node.

We then continue this process of combining the two lowest probability nodes until the entire tree is built.



Given this tree, we assign 1's and 0's to each of the edges between nodes. Each symbol is now represented by the bit sequence corresponding to the path from the root node to its leaf node. For out specific example we get the following bit sequences for each symbol:

word:	the	cat	sat	mat
probability:	0.5	0.3	0.15	0.05
Huffman bits:	1	01	001	000

As you can see, "the" is represented by just 1 bit, and less common words like "sat" and "mat" are represented by more bits. This is exactly what we expected: the algorithm is using fewer bits for more common symbols! It is taking advantage of the structure of the probability distribution to compress information.

That's Huffman coding in a nutshell! Now we can use this algorithm to losslessly compress any data source given a probability distribution over the symbols in the data.

Connection to Autoregressive Modeling

Ok, now that we have a handle on the basics of lossless compression, we can dig into how this applies to DALL-E.

As we've seen, our ability to compress is limited by the specific distribution that the data comes from. And typically for complicated real world data, like text and images, we have no idea what that distribution really is, so we have to approximate it with some model. And if our model is good, and aligns closely with the true distribution, we'll be able to compress the data very well.

This is exactly what DALL-E and GPT-3 are: they are big, expensive models that have learned to approximate these types of complicated distributions. Therefore we can actually use these

models to compress data!

Cross Entropy Loss

In fact, the training objective for GPT-3 or DALL-E can be interpreted as forcing the model to minimize the optimal expected bits-per-symbol of the training data under its distribution. This just is a long-winded way of saying that "the model is trained with cross entropy loss".

Cross entropy loss (with a slight change of bases) is:

$$\frac{1}{|y|} \sum_i -log_2(p(y_i))$$

where y is just a sequence of ground truth symbols $\{y_1,y_2,y_3,...\}$.

The term

$$-log_2(p(y_i))$$

can be interpreted as the number of bits needed to represent y_i under the optimal coding for the model's distribution. Therefore by minimizing this quantity over some training data, we are directly forcing the model to get better at compressing the data.

Empirical Experiments

In fact, this objective of compression is not just theoretical; you can actually use these autoregressive models to compress sequences.

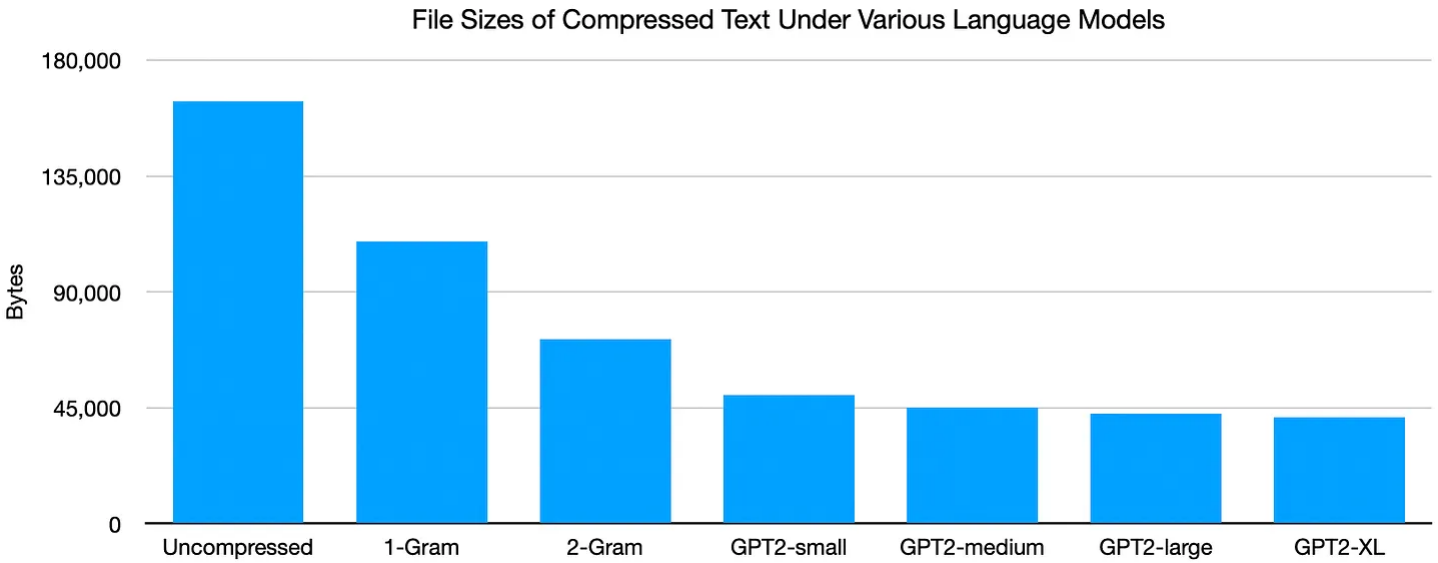
Compressing data with an autoregressive transformer like DALL-E or GPT-3 is actually quite simple:

Encoding:
At each step of the transformer, the model outputs a distribution over what the next image latent

or next word could be. We can just run huffman coding on this distribution and grab the sequence of bits corresponding to what the actual next word or image latent is.

Decoding:
First, query the probability distribution for the next token from the model, then form a huffman tree on this distribution. Follow the path down this tree corresponding to the encoded bit sequence until you reach a leaf symbol. Input this new symbol into the model to get the next distribution and then repeat.

I actually went ahead and implemented this compression algorithm. I ran it on different N-gram models and on different sizes of GPT-2, just to see how the compression ability empirically differs with each of these models.



empirical compression results on the first 81920 tokens of the Wikitext-2 validation corpus. I used the GPT-2 tokenizer on all models. The N-gram models were trained on the Wikitext-2 training corpus, and the GPT-2 models were not fine tuned at all.

All my code is in this Colab notebook, if you wanna see exactly how this works or run some experiments yourself.

As you can see, with each bigger, better model, the ability to compress real data increases. Now there do appear to be some diminishing returns with the larger GPT-2 models, and I actually think this is more due to the limits of huffman coding than the model's actual compression ability. For instance, the theoretical bits-per-symbol for GPT2-XL is 3.87, whereas empirically huffman coding only gets to 4.04 bits-per-symbol. This distinction really adds up to a big difference in bytes when compressing over a hundred kilobytes of text.

Additionally, as the compression rate improves and approaches the true entropy of the distribution, small improvements in the compression rate become more and more significant. So the smaller returns between the large GPT-2 models are probably still quite meaningful qualitatively.

Overall, the moral of the story is: the loss function that all of these models are trained on is directly related to compression.

Intuitions From Compression

Once a large autoregressive transformer is trained, it will probably be very good at compressing the training data. But hopefully it is also good at compressing data it hasn't seen before. This measure of how well the model can compress unseen data is exactly what those scaling law curves from the previous section were measuring. As the model gets bigger, it gets consistently better at compressing data it hasn't seen before.

How can we interpret this? What intuitions about big transformers can we gain from this perspective?

Well you can think of the model's weights as the space that the training data gets compressed into. And usually, the training data is much larger in terms of file size than the model's weights. The model is therefore forced to compress as much of the training data as it can into this smaller

space of dense vectors and weights. To do this, the model likely ends up learning lots of general things about the structure of the data that can be applied in multiple situations.

For example, the most compressed way to learn how to add numbers is not to memorize tables of addition, but rather to implement the general algorithm for addition. That's not to say that these models would necessarily learn to add exactly in this way, but rather this is more just an example to show that these models likely learn lots of general functions and algorithms that can help them operate on the data efficiently. And the general structures that these models learn likely generalize to unseen data as well.

This compression story is pretty simple really: the model is trained to compress lots of data, and it generalizes somewhat because of that. And the larger the model you train and the more data you use, the more likely it is that the model will find non-trivial structures in the data that it can use to compress information and ultimately generalize with.

Even with this perspective, I'm still somewhat surprised that the objective of compression is capable of giving rise to such non-trivial behaviors, like GPT-3's meta learning or DALL-E's incredibly creative image outputs.

Some people would not find this surprising at all though; they would say that this is exactly what they expected. In fact, there are are mathematical theories linking the ability to compress information with an agent's ability to act intelligently in its environment. There is actually a contest, aimed at encouraging AI research called the Hutter Prize, which challenges participants to losslessly compress 1GB of text as much as possible. And if you beat the current compression record you get a nice prize of 500k euros. This idea of compression is not exactly new in the field of AI.

So even if all of this is a little vague, it's clear that there is something very deep about the ability to compress information, and hopefully this compression story gives you some intuitions for how DALL-E is so good at generating images conditioned on text.

Hype and Limitations

The Hype is Sky High

GPT-3 and DALL-E have shown incredibly impressive results and demonstrated seemingly unbounded scaling (technically bounded by the true entropy of the data), so naturally, there has been lots of hype around ideas like "AGI" recently. Some people have even suggested that GPT-3 is already a primitive version of AGI or that with just a couple changes and more scale it will become true AGI. As philosopher David Chalmers said in an interview:

> "What fascinates me about GPT-3 is that it suggests a potential mindless path to artificial general intelligence (or AGI). GPT-3's training is mindless. It is just analyzing statistics of language. But to do this really well, some capacities of general intelligence are needed, and GPT-3 develops glimmers of them." - David Chalmers

Others like VC Delian Asparouhov have talked about how models like GPT-3 are a game changer for society, saying things like:

> "The iPhone put the world's knowledge into your pocket, but GPT-3 provides 10,000 PhDs that are willing to converse with you on those topics." - Delian Asparouhov

or

> "30 years ago, Steve Jobs described computers as 'bicycles for the mind.' I'd argue that, even in its current form, GPT-3 is 'a racecar for the mind.'" - Delian Asparouhov

No one really knows what these models are actually doing under-the-hood, but the hype is sky high. Of course, you can choose to buy into this hype of not. But even if these types of models are nowhere near AGI, they will still likely be incredibly impactful on society.

Is this True Understanding?

I've spent a lot of time in this blog post discussing the incredible capabilities of models like GPT-3 and DALL-E and how well they scale, but I think it's also important to briefly look at some of the theoretical limitations of these models.

In addition to all the hype, there are also many pessimists about these technologies. For GPT-3 and large language models in particular, some have argued that models which only interact with the raw form of language can never gain true understanding. These arguments typically suggest that true understanding is some abstract state that exists in our brains and that language is just a compressed, communicative latent of this more complicated thing. This viewpoint naturally leads to the argument it would be impossible to recover this more complicated mental state from language alone. Many believe that recovering true understanding would, at minium, require models to connect language with concepts from our visual word.

So from this line of argument, DALL-E is actually a big step towards true understanding because it directly connects language with the visual world.

Still others, like professor Garry Marcus, would argue that merely scaling transformers will never be enough for true understanding and especially AGI. Professor Marcus, in particular, is a big advocate for neural-symbolic approaches to AI. These are systems that incorporate formal algebra systems and symbolic manipulation explicitly into the model. The main idea behind this is that deep learning is really uninterpretable and untrustworthy, and also really bad at expressing things like compositional reasoning and causality: two key components for intelligence. Therefore incorporating symbolic manipulation into models could provide a strong prior for this type of reasoning and also improve interpretability.

This is just one definition of symbolic AI though, really the term "symbolic AI" is quite vague and is often used in highly differing ways. But most of these definitions still lead to critical perspectives on models like DALL-E or GPT-3.

In particular, a recent paper from Deepmind explores a new definition for symbolic AI. They define symbols as concepts that are attached to some physical substrate (a word on a screen, the sounds that make up a spoken word or phrase, the visual image of an object), and whose meaning is determined by some societal convention. Specifically Deepmind's paper argues that not only is this type symbolic understanding critical for AGI, but also that in order for a model to exhibit true symbolic behavior, it must be aware of the fact that the meaning of a symbol is arbitrary in itself and is merely set by convention. They also argue that true symbolic behavior requires the ability to interact within a system of symbols; the agent should be able to change the meaning attached to any given symbol whenever a certain context requires it.

They argue that while models like DALL-E seem to be receptive to the meanings of individual symbols, these models are incapable of these other more interactive behaviors that are needed for a true symbolic system.

If these ideas interest you, I'd recommend checking out Deepmind's paper on this topic, it's a really good read, and it goes much more in depth into this.

These are just a sampling of some of the critical perspectives that I'm familiar with, but there are definitely many more. In AI and related fields, everyone has a slightly different take, and some takes are more extreme than others. There are so many different ideas in this field; I think it's really healthy, and it keeps things interesting.

Conclusion

Wow we've covered a lot of ground in this blog post!

- First we looked at the details behind DALL-E's dVAE autoencoder.
- Then we saw how all the pieces of DALL-E come together to generate images from text, in particular we learned about the precise role that the transformer plays in all of this.

- In order to understand why transformers are so good, we looked back at the history of language modeling to gain some intuitions
- We then saw how autoregressive transformer models consistently improve with scale.
- We gained some intuitions about this scaling behavior by looking at an interesting connection with lossless compression.
- And lastly, we surveyed the hype surrounding these models, and also looked at several theoretical limitations that have been proposed.

Hopefully, after reading parts 1 and 2 of this blog series, you now have a clearer idea of the technical innerworkings behind DALL-E. And I also hope that, along the way, you gained some good intuitions for why these models work so well at scale and what some of their limitations might be.

I had a lot of fun writing this series of blog posts. Thank you for reading !

References

- DALL-E blog post
- DALL-E paper
- Gumbel Softmax paper
- Stanford text on N-gram Language models
- Stanford text on vector semantics
- Andrej Karpathy's blog on RNNs
- Transformer Paper
- Jay Allamar's Transformer Blog
- Language Model scaling laws paper

- Generative Model scaling laws paper
- Gwern blog on scaling laws
- Philosophers on GPT-3
- Delian Asparouhov blog on GPT-3
- Paper on language understanding: “Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data”
- Deepmind paper on symbolic AI
- GPT-3 Paper
- GPT-2 Paper
- Wikitext-2 dataset

3 Likes

1 Comment

Write a comment...

guigui Apr 7, 2023

nice article , clear information about all , and I also have ever thought concept is a kind of compression , but with nice structure. now our big model has been nice , but still coarse in formulating the structure of concept .

LIKE REPLY SHARE