[SQUEAKING]

[RUSTLING]

[CLICKING]

PROFESSOR: All right. So today, we actually come to the last lecture of the class, because Wednesday is going to be project presentations. And so I want to talk to you about diffusion models today, which is an incredibly exciting area, which I don't think gets the same amount of attention, in some ways, compared to large language models. But it's got enormous potential. So I'm very excited to talk to you about it.

So just for kicks last night, I asked ChatGPT to create a photorealistic image of graduate students in class, in a class on deep learning, and this is what it came back with. There is a noticeable absence of an instructor.

[LAUGHS]

Plus, various students are facing in various directions. But apart from that, it's not bad. And here is an example of a Midjourney text-to-image diffusion model, which produces the amazing picture from this prompt. A quaint Italian seaside village with colorful buildings, blah blah, blah, blah, blah, rendered in the style of Claude Monet and so on and so forth. And that's what you get. It's pretty unbelievable. And I'm sure you folks have played around with these things, and you have your favorite pictures and prompts and whatnot.

Now, February 15, OpenAI released a text-to-video model called Sora, which you folks may have seen, which I find frankly, just stunning what it can do. It can produce a one minute video from a text prompt. And so if you actually give it this prompt, "In an ornate historical hall, a massive tidal wave peaks and begins to crash, and two surfers, seizing the moment, skillfully navigate the wave."

I think we can all agree that such a thing has never happened in history, and therefore, it was not in the training data. And then, you get this picture, this video. And then, some random person is coming back in a completely dry hall. So anyway, but it's pretty amazing. I think you'd agree.

So if you actually look at the OpenAI's Sora technical report, you actually find this opening paragraph where they say that we train text conditional diffusion models, blah, blah, blah, using a transformer architecture. So now, we know what a transformer architecture is. You've been working with it. You're quite familiar with it at this point. So today's class is really about text conditional diffusion models, the other building block.

OK, so let's get to it. What I'm going to do is I'm going to divide this into two parts. The first part is, I'm just going to talk about, how do you get a model to just generate an image for you? If you wanted to generate an image from a class of potential images, how can you just generate an image? And then, next, we talk about, OK, great.

Now, that you can do that, how do you actually control or steer the model to do an image based on whatever prompting you give it? How do you condition it? How do you control it? Those are all the words. How do you steer it? You'll find all these synonyms being used heavily in the literature. That's basically what they mean. How do you give it a prompt, and then steer what gets produced?

All right. So let's say you want to build a model that can be used to generate images of stately college buildings. Obviously, our very own Killian Court is the finest example of such a thing. But let's say you want to do that.

So what you do is you, as we always do with machine learning, we collect a bunch of data. In this particular case, we collect a whole bunch of images of stately college buildings. And what you see here is literally me just doing a Google image search with the query stately college buildings. So this is the kind of stuff you get.

So you have your training data at your disposal. It's ready to go. Now, the question is, if you have such a model, let's say. And obviously, we talk about how to build such a model very soon. But let's say you have such a model, and every time you sample this model, every time you ask the model, hey, give me an image, you obviously want it to give a different image, otherwise it's kind of boring.

Maybe you want the Killian Court, maybe you want the rotunda from University of Virginia. Any UVA alums here? Nobody? OK. So the question is, how can we actually get it to randomly give us different images? But they all have to be stately college buildings. It can't be just some random stuff. So how do you do that? And the way we do that, and I still find it really astonishing that this approach actually works. The way we do that is that we actually give it noise. And I will define very precisely what I mean by noise in just a bit.

Basically, assume an image in which all the pixel values are randomly picked. So every time you generate a random image and you give it to the model, it will use that random starting point and then create an image for you. And because, by definition, if you choose noise randomly, they are obviously going to be different each time. It's hopefully going to generate a different image. But if the model is trained on stately college buildings, it will produce images of stately college buildings. It's not going to produce a picture of a Labrador Retriever.

So that's basically what we're going to do. Now, if you look at something like this, the first question, of course, is that how can we train a model to generate an image from pure noise? This just sounds ridiculous. You basically give it a bunch of random numbers and say, give me clean code. It feels really ridiculous. And at that point, folks can come to a stop and say, all right, this approach is probably not going to take me anywhere. It's a bit of a dead end.

But then, some clever people had this very interesting idea. They said, it's not clear how to do this. Just a quick aside, there's this really amazing book, which was published maybe 50 years ago, maybe earlier than that, called *How to Solve It,* by George Polya. George Polya was an eminent mathematician, and he wrote this small book called *How to Solve It,* and it lists a whole bunch of heuristics that mathematicians use when they solve problems.

And perhaps the most commonly used heuristic is just reverse the question. Just reverse the question and see if anything comes out of it. Most of the time, nothing will come out of it. But maybe some other times, something amazing comes out. This is a great example of that heuristic at work. We don't know how to do this. So the question is, can we do the reverse? If I give you Killian Court, can you produce noise out of it for me? And the answer is yeah, of course, we can do that.

Given an image, we can easily create a noisy version of it. So you can take the original image, you can add some noise to it to get this. And you keep on adding a lot of noise. And finally, you'll get something that's basically, you can't tell that is Killian Court anymore. This process, the reverse process, is actually very easy to do. By the way, for folks who may not be very familiar with this notion of adding noise to an image or making an image noisy, let me just show you in a Colab, just a minute, how easy it is.

All right. So let's say we import a bunch of these things. As usual, we have NumPy. And so there is this thing called the Python Imaging Library, PIL, which is very handy for image manipulations. So we import that. And then, I just literally read this image in. I uploaded it before class. Let's just make sure it's here. OK, good. killian.png. So I read this image. OK. And then, once I read it, I convert it into a NumPy array.

And then, remember, in any color image, you have three tables of numbers. There is a number for each pixel for red, blue, and green. And then, each number is between 0 and 255. And so here, what we do is we divide everything by 255 just to normalize it so it's all between 0 and 1. And we have done this in the past. I do that here. So let me just read this back in, convert it. And then, if you look at the shape, it's basically a 411 by 583 times 3. Three channels, as we have seen before. And then, let's just show it. All right, that's the picture.

So now, what we want to do is we want to add noise to this picture. All we have to do for each pixel, we basically, randomly pick a normal variable, a normal distribution, normally distributed random variable with a mean of 0 and a small standard deviation. So it's like a small number. And then, we just literally add that number to every pixel.

But for every pixel we sample, every pixel we sample, it's not like we sample once an array of all the pixels, we sample for every pixel. And so the way you do that is basically, literally np.random.normal. And then, this 0.3 here is a standard deviation. And we tell it, generate as many of these things as the shape of the image that I gave you. And then, add each one of these numbers to the original image. You get this noisy image.

So if this is the original image, these are all the values between 0 and 1. And then, you add do this noisy image. You can see the numbers have become different. The 0.23 has become 0.18, 0.15 has become -0.17, and so on and so forth. You've just added a small number random to everything. But as you can see here, now you have some negative numbers. You may have some numbers that's greater than 1. And we do want everything to be between 0 and 1.

So all we do is we do this thing called clip it, where essentially, values smaller than 0 are set to 0, values greater than 1 are set to 1. And so we'll just do that. That's it. Everything over 1, squashed to 1. Everything under 0, set to 0. Others, leave it unchanged. Now, it's again, well-behaved between 0 and 1. And we can just plot it, and you get this. That's it. That's all it takes to actually add noise to an image. One line of NumPy.

Obviously, you can just put this whole thing in a loop, and keep increasing that standard deviation number from 0.3, 0.4, 0.5, so on and so forth. And when you do that, you get this nice sequence of clean code, and all the way to some very, very noisy version of clean code. That's it. So that's the basic idea of adding noise. Any questions on the mechanics? OK, good.

So we can add random numbers. And we can, by increasing the magnitude of the standard deviation of these normal random variables, we can make the image noisier. So that suggests a really interesting idea. What idea would that be? Yeah.

STUDENT: Doing the opposite.

PROFESSOR: Could you please, microphone please.

STUDENT: By doing the opposite. Like, recreating the image from the noise.

PROFESSOR: So we are trying to create the image from the noise, but that feels a little hard. So what exactly can we do? Be a little bit more specific. So here, we have the ability to take any image and add any amount of noise to it. That's the data we have that is Killian Court. And there's various noisy versions of Killian Court, like that for the rotunda, the University of Virginia, and so on and so forth.

STUDENT: I would assume you would do some kind of loss function for the final image that you get, and compare it with the original image that you trained it on. And then, fine tune as you go.

PROFESSOR: OK, you're on the right track. Any other proposals?

STUDENT: I think we could try to train the neural network to reconstruct the image going from the noisy one, that would have all the set with images, find their noise counterpart, and train them to do the opposite, the neural network to do the opposite task.

PROFESSOR: Yup, that's definitely on the right track. That's definitely on the right track. Yeah, good ideas. So what we do more concretely, is we can take each image in the training data and create noisy versions of it, as we have seen before. And then, what we do is that we say we can create x, y training data pairs, input output pairs, from all these images.

So specifically, what we do is we take the slightly noisy version of Killian Court and call it the input, and we take the nice version of Killian Court and call it the output. That's the y1, x1 pair. And then, we get y2, x2, we get y3, x3 and all the way. So at any point in time, the relationship between x and y, what's the relationship between x and y? If you set it up like this, as the input and the output.

STUDENT: It's the set of standard deviations and the values which we obtain for these pixels. Those are, like, weights, which correspond to another feature.

PROFESSOR: Right, or maybe I was looking for something simpler, which was that-- that's correct. What I was looking for is really, the relationship between x and y. X is an image, any image, and y happens to be a slightly less noisy version of the image. The slightly less noisy is really, really important. You're not going from Killian Court, you're not going from the image to full noise. That's an impossible leap. You're going from the image to a slightly noisy version of that image. It is that slightly that allows all the magic to happen.

So that's what we have. And so here, what we can do with these x, y pairs, when you have an-- so here's the thing. This is like a larger comment about machine learning and deep learning. Basically, what machine learning, deep learning are, are really, it's like this black box where if you can find interesting input output pairs, you can learn a function to go from the input to the output. That's it.

But this sounds kind of simple when I describe it like that. But there are some incredibly non-obvious ways of applying this idea. So for example, a few years ago, Google had this thing, which may actually be in production in Google Sheets now, where whenever you choose a bunch of numbers, a range of numbers in a spreadsheet and then go into another cell, it will immediately suggest informative for you. Where is that coming from? It's because all the Google Sheets users all over the world, they have been creating all these numbers with formulas.

So someone says, look, wait a second. We have all this data on people choosing a range of numbers, and then entering a formula. So let's imagine the range as the input and the formula as the output. And let's just give a million examples of this pair and see if anything comes out of it. And boom, you get that feature.

So similarly here, x is an image, less noisy version of the image. What that means is that we can build a denoising network. We can take an image and we can build a network using all these x, y pairs to slightly denoise it. And so how do we do it? We just run stochastic gradient descent on the data. We have a network, which has x and y, and then y is slightly less noisy version. And then, boom, boom, boom, boom, boom.

You're just a network. It has a bunch of weights. We have the right answer in terms of what the images need to be. We can do stochastic gradient descent or Adam or something. And before you know it, if you have enough data, you have a network which can denoise anything you give it. OK. You had a question.

STUDENT: Why slightly?

PROFESSOR: Why slightly? We'll come back to that question. The reason is that in general, you have to do what you can to help the model. And this is the proverbial, there is an old adage, you can't cross a ditch in two jumps. It's too big, so you can't do it. So what you do is you create a bridge to go from here to there. And so what you do is, if you can slightly denoise something really well, well, I can actually denoise anything you want really well using that fundamental capability, as you will see in a second.

STUDENT: Just to follow up. So if you go back the last slide, I could have created the same thing as, that is my x1 and that is my y1. Then, the second one is x2 and still, this is the y. So there is effectively, there is a learning there that it could have taken from those pairs and come back with, OK, this is also a possibility. This is also a possibility.

PROFESSOR: Right.

STUDENT: And found out that noise matrix that it can subtract.

PROFESSOR: Yeah, so the thing is, you want to make sure that each time the amount of learning it has to do is as bounded and small as possible. If you give it some starting point and an ending point and keep moving this ending point, the gap is still really high for the first several of those starting points. That's the problem.

OK. So to come back to this. So we can build a denoising model. We can do this. And now, once you have built such a thing you give it some noisy thing, and then it will give you a slightly less noisy version of it. The resolution is going to go up slightly if you do that. This, of course, suggests the obvious way in which you would use it, which is that once you train it, we can solve this problem. And how can we solve this problem? So what you do is you start with pure noise, and then repeatedly denoise it.

OK. You get that, you get that. And then, before you know it, Killian Court has emerged from the fog. It's pretty insane that it actually works, this idea. So the model will generate a sequence of less noisy images, and the final one you have is the answer. Now, there's a whole bunch of detail here, which I'm glossing over, about, OK, how many times must we run this loop to get to a really good picture? The short answer is initially, it was like you have to run it like 1,000 times.

Each denoising step was like a baby step. You have to do it a thousand times to get a really good answer. Again, research has been very active in the area, continues to be very active. Now, we can, I think, do it with 50 steps or 100 steps. But diffusion models, like this, they tend to take more time than a large language model. Which is why if you give a prompt to one of these models like Midjourney, it will take some time for it to come back with an image. And the reason for the delay is because it's going through this incremental denoising loop. Yeah.

STUDENT: From this, we understand that each, the final noise output sample would be very particular to each image in the beginning. So, I mean, say if you take two images, finally we are getting a start image. And after, when we start noising it, and the final output we get with the noise sample, it will be two distinct for each of them, right?

PROFESSOR: Correct.

STUDENT: But when we are picking up image to generate a diffusion model and we work backwards, we may not have the exact thing available to us, what was there initially.

PROFESSOR: No, no, the thing is, we don't want to necessarily regenerate images that were in the training data. That's kind of pointless. We want to generate new images. And for new images, we just use noise as a starting point. The fact that Killian Court was here, and then the fully noised version of Killian Court is here, that is used for training. And once you use it for training, you don't need it anymore because you're not trying to recreate Killian Court again.

You want to create new images which belong to the category of stately college buildings. And for that, you just grab noise, send it in. It gives you a stately college building. End of story. And because noise by definition is different each time you pick it, it's going to come up with a different stately college building.

So the way you think about it is that-- all right. So you can think of it as this. So when you sample, think of this as the noise distribution. Each time you sample, there is a little point you pick from here. Another time you sample, maybe you get a point here. Each is just a nice distribution. That's it. What actually these things are doing is they are mapping it to the distribution of stately college buildings, which might be in a strange, crazy distribution.

So each time you sample, you just go from here, and you land at a point here. And when you go from here, you land at a point there. So what you have done is when you take the training data, you basically created points here, and then found the matching noise here, and then flipped it for training, as we have seen before. And once you're done with it, you basically have a mechanism for transforming any entry in this distribution of images to an entry in this distribution of images. So it's a way to transform one distribution to another distribution. That's what's going on. All right. So there was a question, yeah, and then we'll go.

STUDENT: I understand going from noise to image and back, how you call how the training works. So my question is, in some of these models today, you have, when you give it the noise now to generate an image, for example, it could generate a human with four fingers or stuff like that. So is it that the model, that the training data is not just quite enough to or as robust enough to generate that kind of detail? Can you talk through what's going on with that?

PROFESSOR: Yeah. So fundamentally, what it is doing is it actually does not understand the notion of fingers and things like that, because we haven't injected any domain knowledge into this whole process, by saying that, hey, you need to generate a human body, and here are the semantics of what the human body is. It's got five fingers and all the anatomical stuff. We're not giving anything. We're literally giving it pixel values, a bunch of pictures. So everything you're seeing is basically just coming out of that very blind statistical transformation process.

So you would expect that macro level details would probably get it right because there are so many right answers. So imagine it's actually it's creating the roof of a house. There could be all kinds of variations in the roof of the house and you would still think it's a roof of a house because there are many possible right answers. But when it comes to five fingers, there aren't many possible right answers, which is why you notice the error very quickly.

As far as the model is concerned, it doesn't know. It's just producing a statistically plausible sample from that distribution. And since we haven't forced it to obey constraints like five fingers and so on and so forth, it's not going to do any of that stuff. It's an unconstrained process. Now, over time, these things have gotten better and better, and that's because the data has gotten better, to your point. But I think our approach to doing these things is also getting better.

There are lots of ways to now steer it and control it so it behaves the right way, and that is actually part of what's happening as well. So when we talk about, how do you actually give a text prompt and have it build the image for that particular prompt? We will revisit this question. OK. There were more questions. Yeah.

STUDENT: Is there some randomness in the model itself? So if you gave it the same noise image twice, will it actually produce the same final image or will it come up with something similar?

PROFESSOR: Yeah, there is randomness, in the process as well.

STUDENT: In the process as well?

PROFESSOR: In the process.

STUDENT: Yeah.

PROFESSOR: Exactly. So to actually, that's a really good point. But now, I'm afraid to open my laptop or my iPad. One second. All right. So what's going on here is that if you go to this thing, so I talked about we are transforming from here, to some crazy distribution here. So what happens is that let's say that this is the starting point for the noise input. This is your noise input. And then, what you actually do is you go here, and then you take this point. And then, you do a small sample next to it.

So you use this as the mean value and then sample around it, and that's actually what gets published in the user interface. That's where the randomness comes in. OK. So back to this. Was there another question somewhere? Yeah.

STUDENT: Not sure.

PROFESSOR: It's OK.

STUDENT: I was wondering about, when training on a clear picture to go to a noisy image, to display where a random sample with a random drop, the sample probably is pseudorandom. I'm just wondering if it's like learning relationships that are dependent on pseudorandomness. And so when it goes from a noisy image back to a pure image, is it dependent on that, or if that matters at all?

PROFESSOR: Oh, I see. So if I understand your question, what you're saying is that it's pseudorandom, not actually random.

STUDENT: Yeah.

PROFESSOR: And so therefore, there is some signal in the supposedly random generation.

STUDENT: Right.

PROFESSOR: Is it actually glomming onto that signal, right, is the question?

STUDENT: Yeah.

PROFESSOR: Theoretically, it's probably possible, but in practice, it really doesn't matter. Because we basically say pseudorandom is good enough for our purposes. And in fact, in practice, you will see it's not an issue. Oh yeah, go ahead.

STUDENT: I just had a quick question. When you're doing text-to-text, let's say you're tokenizing the input. But here, you somehow have to identify that this is Killian Court and like a stately home. And this is just going from pixel to image or decoding a pixel image. Where does the tag or tokenization of columns or fingernails, or like--

PROFESSOR: There's nothing. It's learning everything from the pixel values.

STUDENT: Everything's from the pixel values?

PROFESSOR: Yeah, and this is what I was, when Ike asked the question about the four fingers, five fingers thing, it has no idea of fingers. It has zero knowledge about any of these things. All it's seeing is a bunch of photographs.

STUDENT: So when you type in, you say, I want to see a hand with green fingers--

PROFESSOR: Oh, I see. So we haven't yet come to the stage of, OK, how do you actually steer this image using your text prompt? It's coming.

STUDENT: OK.

PROFESSOR: Right now, all we are saying is that, look, I'm going to give you a bunch of photographs of a particular kind of thing, stately college buildings. And I want to have a model, which at the end of the day, I just poke it. Every time I poke it, it gives me a stately college. That's it. Now, I'm going to actually start giving it text and saying, OK, create the thing I'm just telling you about. That's coming. And that's some additional magic is going on to get that done.

OK. So this is what we have. And this is called a diffusion model. And this is the original paper that figured this out. And the process of actually creating-- if taking an image and creating noisy versions of it to create a training data is called the forward process, and then what we did in reverse is called the reverse process. Check out the paper. It's actually really well written and I recommend it.

Now, in practice, some other researchers came along shortly after this and made a small improvement. It turns out to be actually a big improvement in practice in terms of improving the quality of what's being produced. And so what they said is, hey, instead of training the model to predict the less noisy version of the image, we actually ask it to predict just the noise in the input. And then, we will just simply subtract the noise from the input to get the image.

So instead of saying x is an image, y is the noisy image, we actually tell it, here's the image. Here is the noise that we added to x to get the noisy version. And then, just predict the noise for me. And then, once I get it, I just do x minus noise and I get the less noisy version of the image. This feels arithmetically equivalent, but in practice, it ends up generating much higher quality images. And there is some very interesting theory as to why that works and so on and so forth. And you can read this paper if you're interested.

So if you actually look at what's going on in most diffusion models today, they're basically using an approach like this. They're actually predicting each time they predict noise and take it away, subtract it. So iterative subtracting of predicted noise. That's what's going on. So all right, that's what we have.

Now, at this point, you might be wondering, OK, so far in the semester, we have actually learned how to take an image and then classify it into one of 20 things, 10 things whatever. We've also taken text and figure out what to do with things with it. We haven't yet talked about how do you actually take an image, and how can we get the output also to be another image? We haven't done that yet.

So we have actually not done image-to-image. How do you actually build a neural network to do images? And in the interest of time, you're not going to get into it massively, but I want to give you a quick idea of how it works. So the most, I would say, the dominant architecture to take an image as an input and produce an image as an output is called the U-Net. And that's architecture we see here.

So fundamentally, if you look at the left half. So there's a left half to the network and a right half to the network, hence, the U. If you look at the left half of the network, it's a good, old convolutional neural network, like the kind we know and love and the kind that we are very familiar with. So you take an input image, and then you run it through a bunch of convolutional blocks, and then we do some max pooling, and then we keep on doing it. And at some point, it becomes smaller and smaller, and we get something like this, which we are very familiar with.

The big image with the three channels gets smaller and smaller and smaller, but the number of channels gets wider and wider. It becomes much smaller but much deeper. It becomes like a 3D volume. And we have seen that again and again. The left part is just a good, old convolutional with pooling layers. And then, you come to the middle. And then, from this point on, what we do is we take whatever this thing here, and then we essentially reverse the process.

We go from the small things, which are really deep, to slightly bigger things that are a little less deep, and so on and so forth, till we get the original size back again. And we do that using an inverse of the convolution layer called an up convolution or deconvolution layer. And you can check out 9.2 in the textbook to understand how it's done. It's also called conv 2D transpose.

It's a very similar idea. And I'm not going to get into the details here, but you essentially do the inverse of a convolution operation to get the size to come back to the bigger size. And you do it gradually, till the output you have matches the size of the input that came in. So image gets smaller and smaller into a thing, and then you just blow it back up again to get an image back. So that is the U-Net.

Now, there's really one very important thing that happens in the U-Net, which is you see these connections. Basically, what they do is at every step, when you're coming back up in the right half, you actually attach whatever was in the mirror image of the original input as we processed on the left side, we attach it to this side as well.

Remember, I talked about this whole notion of a residual connection, back many classes ago, where I said, when an input goes through each layer of a neural network, at one point, let's say you're in the 10th layer, you're only seeing what is the ninth layer has produced for you. That's all you're working with. But wouldn't it be nice if the 10th layer actually had access to the eighth layer, the seventh layer, the sixth layer, the fifth layer? Heck, why not the input?

Because the more information it has, the more able it's probably to do whatever it can with the inputs given. Why restrict it to only the input of the output of the previous layer? Why can't we give it everything that has came before it? Now, giving everything is too much, but we can be selective in what we give it.

So what these folks decided, I'm sure, after much experimentation, is that if they actually attach whatever was coming out of this layer to this layer before it goes through the output, it really helped. Similarly, this thing gets attached, and so on and so forth. And it kind of makes sense. Why force it to figure out everything it has to figure out just from this thing that came in? Let's give this that. Let's also give a little here, a little here.

So these residual connections are a huge building block for why these things work as well as they do. And in general, giving a layer as much information as we can give it is always a good idea, but you can't go nuts because then you have much more parameters and all kinds of stuff happens. So there is a bit of a balance you have to strike, and this was the balance struck by these researchers. And so this thing was originally invented for some medical segmentation use cases, but it is just heavily used for everything now. It's a really powerful architecture. Questions?

STUDENT: Can we have an example of in what scenarios would you use to spin up a network?

PROFESSOR: Anytime you have an image-to-image.

STUDENT: What kind of conversion do you have [INAUDIBLE]

PROFESSOR: Oh, like, what kind of examples of use cases? Let's say that, for example, you want to take an image, like a black and white image, and you want to colorize it, for instance. Boom, you U-Net. You want to take an image and make it a higher resolution image, U-Net.

You want to take an image and for every pixel in the image, you want to classify it into one of 10 things. So anytime when you want the output shape, the shape of the output, to be basically the same shape as the input but with other data, you need to use this. Yeah.

STUDENT: This logic of having access to all the previous iterations--

PROFESSOR: Not iterations.

STUDENT: All the previous layers.

PROFESSOR: Correct, outputs of the previous layers.

STUDENT: Layers. But this would also help for clean up and give better categorization. Does it always have to be an image-to-image?

PROFESSOR: No, no. In fact, if you look at ResNet, ResNet is the one, in fact, that pioneered the idea of the residual connection. So we use it for ResNet. We actually use the transformer stack. If you remember, it goes through the self-attention layer. It comes out the other end. And then, we add the input back to it. And then, we send it through LayerNorm.

So you will see that this residual connection is sitting in two different places in a single transformer block. So it's extremely heavily used. There is something called Deep and Wide Network, if I remember, or DenseNet, which uses the same trick. In fact, when you're working with structured data, good, old, say, linear regression, and you've looked at your data and you've come up with all kinds of very clever features.

I'm going to look at price per square foot. You do a bunch of feature engineering and you have a bunch of new features. Well, you should take your old features and your new features and send both in. Why send only the new stuff that you have concocted? Why can't you send everything in? That's the idea.

All right, so let's come back here. Now, we have seen how to generate a good image. Now, let's figure out how to steer it or condition it with a text prompt because that's, sort of, the Holy Grail. So we want to take, so here is some intuition. We want to take the text prompt into account and obviously generating the image. Now, imagine if we had like a rough image that corresponds to that text prompt. Just imagine.

So the text prompt is cute Labrador Retriever. And you have a very noisy image of a Labrador Retriever that just happens to be handy. You have it. Well, now, you're in good shape because you just feed that in, and your system will denoise it for you. You can get a better image. That's pretty easy. But obviously, in reality, you don't have a rough image. In fact, you're trying to create one of those things in the first place. We don't.

But what if we had an embedding for the prompt that's close to the embeddings of all the images that correspond to the prompt? So let's take a prompt, and let's imagine all the images in the universe that correspond to that prompt.

And now, further imagine, because everything is a vector, everything is embedding in our world, that the image has an embedding. Sorry, the text prompt has an embedding. Every image has an embedding. And we have somehow calculated these embeddings so that the text prompt's embedding is smack where all the image embeddings are.

We will get to how we actually do it in just a moment. But conceptually, imagine if we had an embedding, if you could calculate embeddings for text and embeddings for images, so they all live in the same space So if we feed this embedding to a denoising model, because that text embedding is sitting in the same space as all the image embeddings that it corresponds to, maybe our model can just denoise that embedding and give you what you want.

So since this embedding is already close to the embeddings of the things we want to generate, maybe we'll just get it done. So ultimately, we want to generate an image. And if we had an embedding for that image, we could generate the image from the embedding. And we use the text. So we go from text to embedding, which happens to live in the same space as all the embeddings of the images we care about. And then, from that image embedding, we go to the final image. OK.

This is a bunch of me talking and hand waving. It will all become very clear. But that's the rough intuition. OK. So what we'll do is we'll describe an approach to calculate an embedding for any text, any piece of text, that is close to the embeddings of the images that correspond to that piece of text. So this is the problem you're going to solve. There's a bunch of text.

Conceptually, there are a whole bunch of images that describe the text. And we're going to now create embeddings so that that is close to all the embeddings of those images. It feels, kind of, almost impossible that you can actually do something like this. But there's a very clever idea that OpenAI came up with that tells you how to do it.

So here's what we're going to do. So let's say we have an image and a caption. So here is an image. Here's a caption. And we need some way to take that piece of text and run it through some network and create a nice embedding from it. Similarly, we want to take this image, run it through some network and create an embedding from it.

Now, first question. How can we compute embeddings from a piece of text? First question, how can we compute an embedding from a piece of text? You know the answer. Run it through a transformer. Piece of cake. We know how to do that. In particular, you can do something like BERT. And for an image encoder, you just run it through something like ResNet. The penultimate layer, one of the final layers, the ResNet is going to be a very good representation of that image. You get another embedding.

So using the building blocks, we already know, we can create embeddings very quickly from these things. But if you just take a piece of text and run it through a BERT, and you take an image and run it through ResNet, you're going to get some embeddings. But why the heck should they be related? They were not trained together, so there's no basis for them to be related. They would just be some two embeddings. Maybe they are kind of similar. Maybe they are not. We don't know. There's no reason to expect that they're going to be similar. They just do embeddings.

Now, what we want to do is, once we have these, we need to make sure the embeddings that comes out of these two things satisfy two very important requirements. We want to make sure that if you give it an image and a caption that describes that image. So you have an image and a subcaption that describes that image, we want to make sure that the embeddings that come out of these two boxes, they are as close to each other as possible. Given an image and a caption that describes it, that's the connection. They have to be close to each other.

And conversely, if you have an image and a caption, that's totally irrelevant, a train rounding a bend to the beautiful fall foliage all around, clearly irrelevant, those embeddings should be far apart. That intuitively makes sense. Pairs of related things should be together. Irrelevant things should be far apart. So if you can find embeddings that satisfy these two criteria, maybe we will be in the game.

So now, this ensures that the text embedding and the image embedding are referring to the same underlying concept. These requirements will enforce that. And so the embedding for any text prompt is close to the embeddings though, for all the images that correspond to that prompt. So the question is, how do we do this? First of all, how can we tell how close two embeddings are? You know the answer to this. What's the answer?

STUDENT: Cosine similarity.

PROFESSOR: Correct, cosine similarity. We use the cosine similarity of the embeddings. So we know how to measure closeness So the question is, how can we compute embeddings that satisfy the two requirements. And OpenAI built a model called CLIP, which is very famous, to solve this problem. It stands for Contrastive Language-Image Pretaining. And this forms the basis for a whole bunch of models that have sprung up after this called BLIP and BLIP 2 and so on and so forth, but this is the fundamental idea.

So this is how clip works. What they did is they took a 12-block, 12-layer, 8-head Transformer Causal Encoder Stack as a text encoder. Now, you understand this, right? That's what it is. 8-head, 12-layer Transformer Causal Encoder, TCE Stack. And that's a text encoder. So we send any piece of text through it. You get the next word prediction embedding. And that's embedding you're going to use.

And they took ResNet-50 and made it the image encoder. They took ResNet-50, chopped off the top. And whatever was left is the image encoder. OK. Then, they initialized with random weights, these things, and then they grab a batch of image caption pairs. So in this example, let's say that we have these three images, and I have captions to go with these images. We have these three things. And this is the key step.

They run the images through the image encoder and the captions through the text encoder and get these embeddings. It's a forward pass. You send it through this network, you get to embeddings. And then, this is what they do. With these embeddings, they calculate the cosine similarity for every image caption pair. OK. And so imagine something like this.

So you have these three captions. You have these three images. And those are the embeddings. And then, they calculate the cosine similarity for every one of those things. It took me like five minutes or 10 minutes to do this PowerPoint. You're welcome.

[LAUGHS]

Particularly, trying to get this comma to line was as a real pain in the neck. So all right. So we have this here OK, now what we want to do is we want these scores to be as high as possible because the scores in the diagonal are the ones for the matching picture and caption. Those are the scores for the matching pairs of embeddings. We want them to be as high as possible.

So we want to maximize the sum of the green cells. These are the green cells, the diagonal. So if you want to write it as a loss function, because the loss function is always minimization, we basically say minimize the negative sum of the green cells. So the question is, will this loss function do the trick? It seems reasonable. You want to make sure that related things are really close together. So you want to maximize.

STUDENT: If that was the only part of the loss function, wouldn't it just kind of squish everything to the same spot in space?

PROFESSOR: Correct. What it's going to do is it's going to basically ignore the input. The optimizer can simply ignore the input, make all the embeddings the same. For example, it can just make all the embeddings 0. That's it. And then, now, we have a perfect cosine similarity for everything. For any pair of image and captions, the cosine similarity is going to be 1. It's perfect. So clearly, that's not enough. This is, by the way, is called model collapse. So to prevent it from doing that, we need to do one more thing to the loss function. Any guesses? Yeah.

STUDENT: Make the images that aren't related not have a cosine similarity.

PROFESSOR: Exactly right. Exactly right. So what we want to do is we want the scores of the red stuff to be as small as possible. We want the green stuff to be as much as possible and the red stuff to be as small as possible. Together, it'll get the job done. So we want to maximize the sum of the green cells and minimize the sum of the red cells, so the equivalent loss function is minimize the sum of the red cells and the negative sum of the green cells. That's it.

So all CLIP does is that it just grabs a batch of image caption pairs, runs it through the networks, calculates the embeddings and calculates some of the stuff here. And that is your loss. And then, back propagate through the network. Boom. Batch, batch, batch. Do it a whole bunch of times. And OpenAI did this with, oh, this is the official picture from the paper, which is worth reading, by the way. It comes in, text encoder. You get these embedding vectors, image encoder. And then, boom, the diagonal is maximized and the off diagonals are minimized.

And they did it with 400 million image caption pairs scraped from the internet. 400 million. By the way, folks who work in this space may know this really well, but one very easy way to get a caption for an image. We see images, but where do you think the captions come from? Where did they get those captions? Obviously, they didn't ask people to manually label each image with the caption. Where do you think they got it from?

STUDENT: Google Search?

PROFESSOR: Google Search can help, but why does Google Search actually find the caption? How does it, because Google Search is not creating the caption.

STUDENT: Take it from the alt text on the images.

PROFESSOR: Correct, alt text. So a lot of folks, for accessibility reasons, they have alt text on the images they create. A lot of people have alt text in their images that they have published on the web, and that's what we use. And the alt text actually, ends up being a more verbose description of the image than a typical caption, which tends to be much briefer. And for us, more verbose, longer the better, because there's more stuff for the model to learn from. So that's why they built CLIP.

And so now, what we do is we can use CLIP's text encoder by itself. We can send in any text and get an embedding that is close to the embedding of any image that's described by the text. OK. Now, by the way, CLIP can be used for zero-shot image classification. And what I mean by zero-shot image classification, and I'll walk through the picture in a second, is that typically when you want to build an image classifier, you get a whole bunch of training data of images and their labels, and then we train them.

Maybe you take something like ResNet, chop off the top, attach our own output head and train, train, train. Boom. You have a classifier. But the only problem with that is, let's say that tomorrow-- so today, for example, you had five classes in your problem. And tomorrow, somebody comes along and says, oh, actually, we have a sixth category. What do you do then?

Well, you have to go back to the drawing board and retrain the whole thing with six labels now, not five, because your problem has changed. Wouldn't it be great if you had a classifier where you just come to it and say, here's an image, and here are the six possible labels I want you to pick from. Pick one for me. And you want to be able to give it a different set of labels each time. And it will just use the labels you're giving it, and the image, and figures out which label corresponds to the image you just fed it.

That would be an insanely flexible image classification system. And that's what I mean by zero-shot image classification. And you can use CLIP to do zero-shot image classification. Now, how you do it is actually in the picture, though not very clearly done. Anyone wants to guess? How can you use CLIP to build an infinitely flexible image classifier?

STUDENT: I mean, the text input was trained BERT, right? So in the same way BERT can handle words it's never seen before, does it essentially do that?

PROFESSOR: Sorry, say that again, the second part.

STUDENT: You're saying it sees a text input with something it's never seen before, right?

PROFESSOR: Yeah.

STUDENT: OK. So in the BERT model, which is where it came from, in the text encoding. In the BERT model, I think we talked about when it sees a word it doesn't know, it's never seen before, it can use the context words around it to try to--

PROFESSOR: Right track. But here, just to be clear, I want you to use CLIP that we just built. And assume CLIP knows all the words because it's been trained on a big vocabulary. You can give it any text you want. It will create an embedding from it. That's the key capability.

STUDENT: So it creates a text embedding for a text compute.

PROFESSOR: Yeah.

STUDENT: But because there are things, and then for image [INAUDIBLE]. So for comparing the similarity scores between the two, image is complete but the text is not complete. So there will be missing spaces. And then, it makes a prediction using this method.

PROFESSOR: But why is there a missing piece in the text?

STUDENT: Because the image, the text, the text does not contain the class. And then, well, for the image, the way it was trained, it was trained with pairs with class included.

PROFESSOR: Right, but we actually know the class now. So the use case is that I come to you with an image and I say, here are the seven possible labels for this image. And each label is a piece of text. So you actually have seven pieces of text and an image. And all I want CLIP to do is to tell me, OK, the seventh, the fourth label is the right one for this image, but you're on the right track. Once you see how it's done, you'll be like, yeah, of course.

STUDENT: I may not be understanding something, but wouldn't you just pick the embedding that's the closest to the, like, the text embedding that's the closest to the image embedding?

PROFESSOR: Correct. You're not missing anything. That's the right answer. Well done. [LAUGHS] Come on, people. Can we applaud our fellow peer?

[APPLAUSE]

You folks are hard to impress. That's exactly what we do. So here, the key thing to remember, the key thing to keep in your head is that when a label is just text, dog, cat, it's just text. So you can just imagine taking each label, which in this case is plane, car, dog, whatever. For each one of them, you create an embedding. You get T1 through whatever, if you have n labels.

For the image, you just have one embedding, I. And then, you just create the cosines, calculate the cosine similarity. And whichever is the highest number, you say, OK, it's a dog. That's it. It's super, just imagine the level of flexibility here. So that's a side use of CLIP unrelated to diffusion models, but I just thought it was really clever, so I wanted to share that. OK, good.

Now, let's see how we can actually use this entire capability to go to solve the original problem we set out to solve, which is can we steer the diffusion model to create an image based on a particular prompt we give it? So now, remember, if we go back to how we did it, we created all these training pairs of x and y based on denoising the image. X is the image, y is the less noisy version of the image.

So what we can simply do is we can actually change the input so it becomes the image, and then the clip text embedding of the caption for that image. So you have an image and you have a caption. You take the caption, run it through CLIP. You get an embedding. By definition, that embedding lives in the same space as all the images that correspond to that caption.

So you just attach, you concatenate the embedding of the CLIP output of a caption along with the image. You make that the new input now. Why continues to be the less noisy version of the image? Or as we saw earlier, it could be just the noise component of that image. OK, this is the new x, y pair that we have. And so now, the model is you send the CLIP text embedding, the image x. Send it through this noisy version of the image. And you keep on training it for a while.

Once your model is trained for when you want to use it for inference, for a new prompt, you just give it, Killian Court at MIT during the spring time, along with a bunch of noise. Goes in, it starts denoising it. But because this embedding of this thing, thanks to CLIP, lives in the same image space as all Killian Court embeddings, Killian Court images, you keep on doing it for a while. At some point, you will get Killian Court.

That's how they do it. That's how they steer the image. It's a two-step process. You create all these clip embeddings, which CLIP was a breakthrough, in my opinion, because it was one of the, maybe the first example. I don't know if it's the very first but one of the early examples of saying we have different kinds of data, we have images, we have captions, we have text. How do we create embeddings for every one of these very different data types that all happen to live in the same space, the same concept space?

That was the key idea. And if you look at the modern multimodal large language models, they are all based on the same exact idea. So it's very powerful, this approach.

STUDENT: Yeah, now I understand this for images, but for video generation models like Sora, do they have some sort of underlying physics structure, or do they learn the physical representations?

PROFESSOR: There's a lot of debate on the internet about this stuff. They haven't published the results of the full technical report yet, so we don't know for sure. But the consensus seems to be no, they are not using a physics engine. What they have done, and again, this may be wrong. Once the report comes out, we'll know for sure.

But what people are saying, computer vision experts, is that it has been trained on a lot of video game data, along with actual videos and so on. And the corpus of training is so massive that it has basically learned to mimic certain physics aspects to it just as a side effect. Much like LLMs, you train them on a large amount of text data, they begin to start to do things which you didn't anticipate that they'll do.

So for example, I read this, I thought it was a really great example of what is surprising about large language models is not that you train them on a bunch of high school math problems, and then you give it a new high school math problem. It can actually solve it. That's not surprising. You give it a whole bunch of high school math problems in English, then you ask it to read a bunch of French literature, and then you give it French high school math problems to solve it. That is the new news.

So similarly here, I think the expectation is that it's not actually using a physics engine under the hood. It may have used a physics engine to actually come up with the videos and renderings, but there are no physics constraints in the model itself. It just comes out of the training process. That's the current view. Once the technical report comes out, we'll know for sure what they actually did.

STUDENT: So one quick question about stability. It's claiming to be a little bit more real time in our image generation.

PROFESSOR: You mean Stable Diffusion?

STUDENT: Yeah.

PROFESSOR: Yeah, yeah.

STUDENT: For Stable Diffusion. So are they jumping through the noise more quickly, or are they kind of preprompting it, and kind of trick--

PROFESSOR: Very good question, and there's a very key trick. It's coming. You had a question.

STUDENT: So here, the example of the noise is normal distribution.

PROFESSOR: Yeah.

STUDENT: However, if we have changed the noise distribution, would it change the result?

PROFESSOR: Oh, you mean if you change it to a Poisson or some other distribution? It will definitely change the results. Because if you look at the underlying math of why this works, it heavily depends on the Gaussian assumption. Yeah. There was another question somewhere here.

STUDENT: You may not know the answer because the technical report is not out, but could it be in terms of video generation that's analogous to going from one noisy image to another, like you're almost doing a series of still images and learning how to--

PROFESSOR: No, that, I think, people are sure is how it's done. So basically, you think of the video as just a series of frames, and each frame is an image. And there is a sequentiality to it, which is where the transformer stack will come in because it handles sequentiality. So in general, video stuff typically operates on frame by frame, which is just an image. So that is definitely there.

What we don't know is if they also used some understanding of the fact that, for example, that if an object is dropped, it has to fall to the Earth in a certain rate. Or if an object goes behind another object, you can't see the object anymore. Things like that, which we take for granted. The question is, are they using it?

And the consensus seems to be, in the absence of an actual technical report, that no, they're not doing it. Because there are lots of examples on Twitter where people will show a Sora video in which it's not obeying the laws of physics.

So you take like a beach chair, and then put it in the sand. You see the sand come through the base of the beach chair. Or you take an object and put it behind an object, you can still see the object, even though the original object is opaque. So you'll be seeing some evidence that no, no, it's not obeying the laws of physics. What you're seeing is just an amazing-- you just learn to fake the laws of physics in some context. Much like you can fake five fingers without knowing that it has to be only five fingers.

OK. All right. So let's keep going. Now, so there was another paper afterwards, and this is the original paper, which took that idea of the diffusion model. And then, diffusion models are very slow, as Olivia, you pointed out. So the question is, can we make it much faster? So what they did, and I'm not going to get into this whole thing here. I just want to highlight a couple of things.

The first one is that, first of all, notice that you see U-Net here. So they are using a U-Net, to go from image-to-image. The second thing is that the CLIP embedding of the text prompt is basically, is woven, meaning it's incorporated into the U-Net through an attention mechanism, a transformer mechanism. And you can see the QKV business here, which should be familiar at this point. So it is integrated into the transformer stack directly that input the CLIP embedding, the second thing I want to point out.

And then, thirdly, and this is where the speedup comes. So what you do is instead of taking the image, running it through the whole network and creating a slightly less noisy version of the image, here, what you do is you take the image, you run it through an image encoder, you get an embedding. And now, you only work with the embedding. You take the embedding and create a slightly less noisy version embedding. Keep on doing it. And these embeddings are much smaller than images, therefore, they are much faster to process.

And once you've done it 1,000 times, you get a very, sort of, almost pure noiseless version of the embedding. Now, you run it through an image decoder to get the image out. So the idea here is that you operate in the latent space, meaning the embedding space. And hence, it's called the latent diffusion model.

So that's where the speedup comes. But research continues to be very strong, to make it even faster. Because for a lot of consumer applications, people are obviously not going to wait around for, I mean, who wants to wait for ten seconds, right? And so there's a lot of pressure to make it even faster.

All right. So that's what we have. Obviously, these models are transforming everything. And by the way, this site here, lexica.art, you can go check it out. It has a whole bunch of very interesting images and prompts that created the images. So if you're working in this space, it gives you a lot of interesting ideas. But it's not just for consumer fun applications. These models are being used to actually-- you know, AlphaFold, if you'll recall, if you give it an amino acid sequence, it actually create the 3D structure.

So that's an example of, I don't think they use a diffusion model, but you can imagine using a diffusion model to create these complicated objects. Meaning, the objects you create don't have to be images. They can be arbitrarily complicated things. As long as you have enough data about such things to use for training, and the notion of noising the input is meaningful, you can create some very interesting structures.

You can create 3D things and protein structures. And there's a whole bunch of very interesting applications in biomedical sciences. So this is really just the tip of the iceberg. And now, there are these things, there are ways in which you can use diffusion models to do large language modeling as well. So there's a lot of overlap and blending and so on going on in the space.

So I'm going to do a quick demo. If you look at Hugging Face, there is something called the diffusers library, which is like, as the name suggests, it's a library for a lot of diffusion models. And let's take a quick look. All right, so the diffusers library has a whole bunch of diffusion models. We are going to work with Stable Diffusion, which is one of the better known models. So let's install diffusers.

You will recall, when I did the quick lightning tour of the Hugging Face ecosystem for language, Hugging Face has a whole bunch of capabilities built out of the box. And you use this thing called the pipeline function to very quickly use any model you want. The same exact philosophy applies here. You still use the pipeline. So I'm going to import a bunch of stuff.

All right, so. Oh, I see, I to do this thing. OK. Great. Hugging Face. All right, now that we have here. So you remember that when we worked with text, we would grab a pretrained model and then we actually run it through a pipeline. And we can do all the inference we want on it. The same exact philosophy applies here. And this is very similar to what we did in lecture 8 for NLP.

So what we're going to do is, we use this command, the StableDiffusionPipeline from pretrained. And we use this version, 1.4 Stable Diffusion model. So let's just create the pipeline. And obviously, we have used TensorFlow, not PyTorch here, but a lot of these models, unfortunately, happen to be in PyTorch. So knowing a little bit of PyTorch is actually very helpful to be able to work with these things.

And what we're doing here, while it's downloading, we are using this FP16 storage format for the model weights, because it's going to be a little smaller than using 32 bits. So it will download faster. So that's what's happening here. So, all right. It's downloaded fine. So now, we just give it a prompt. And this is actually one of the original famous meme prompts, a photograph of an astronaut riding a horse.

And so once we have the pipeline set up, I'll just set a seed for reproducibility. And then, literally, I do pipe a prompt. And then, it's actually, you can see here, 50. So it's going through 50 denoising steps. OK. And you come up with an astronaut riding a horse. So that's that. You can actually change the seed, and you can get a different-- the seed is basically, sets the random starting point for the image. So therefore, you would expect a different astronaut. Yep. This is an astronaut riding another horse.

So I think people came up with these kinds of fun examples because it's guaranteed not to be in the training data. So whatever the model is doing, it's not regurgitating what it has already seen. All right. Give me a prompt. Prompts, anyone? Wow.

STUDENT: MIT professor riding a horse.

PROFESSOR: [LAUGHS] Ooh. OK. That might be a-- all right, riding a horse.

[LAUGHS]

All right.

[LAUGHS]

There are two of them. And clearly, MIT professors don't have, really--

[LAUGHS]

Yeah. Moving on. So by the way, you should spend some time with the diffusers library. They have a bunch of tutorials, which are really interesting. Because this core capability of giving a prompt and getting an image out can actually be manipulated for all sorts of very interesting use cases.

So for example, there is this thing called negative prompting. And the idea of negative prompting is that you can give it two prompts and say, create an image which embodies the first prompt but not the second prompt. Essentially, subtract the second prompt from the first one. That's called negative prompting. And you might be wondering, what use is that? There are lots of fun uses.

So here, the prompt is going to be, Labrador in the style of Vermeer. That's the first prompt. 50 steps. Look at that. Amazing, right? But maybe you don't care for the blue scarf. So you basically give it a negative prompt. And you basically, the negative prompt is blue, meaning remove everything that's blue. I don't like this. Otherwise, keep the Labrador thing going. So you run it. Look at that. The blue is gone. Negative prompting. OK. Yeah?

STUDENT: If you change that from 50 to 1,000, will it become less pixelated or will it eventually just keep going and iterating?

PROFESSOR: No, typically, if you do more of these things, it gets better. The quality is much better. Because each step will denoise it very slightly, so errors won't accumulate and things like that. And the diffuse library gives you lots of controls for fiddling around with all these things. OK, so that's what we had. 9:49. OK, so check out this tutorial if you're curious about how this stuff works.

And I'm going to do one other thing, because I didn't get to do it earlier on. So we spent some time with the HuggingFace Hub, and I walked you through a few use cases for text where you can take a text model and use it for classification, things like that, summarization and so on and so forth. You can do the same thing for computer vision models. So if you have a computer vision problem that just maps to a standard computer vision task, you can just use the HuggingFace Hub as well. So let me just show you very quickly, the same kind of thing actually works here.

All right. So let's say that you want to classify something. You just import the pipeline as before. And once you import it, you can just literally give it the standard task that you care about, like image classification. And then, you can start using it right from that point on. OK. All right. OK. So now, I'm going to just get this image. So it's a very famous image. And we're going to ask it to classify this image. So we just literally run it through the pipeline.

And it says the most likely label is 94% probability. It's an Egyptian cat. Seems reasonable. I mean, it's a tough picture because there are lots of things going on in that picture. It's not like, one image, one object. So you don't have to use the default model. You can actually give it your own model that you want. So for example, you can go, you can go to the Hub, HuggingFace Hub, and you can go in there and say, all right I want image classification.

These are all the models, 10,487 models. They're sorted by, I don't know, most downloads or maybe most likes. And you have all these models. You can pick any one of them. So for example, let's say you want to pick Microsoft ResNet as your one. That's what I tried here. So I have Microsoft ResNet. You just do model equals that, run it, and it takes care of all the tokenization, this, that and whatnot. It's really very handy. And then, you run it through the pipeline again, and it says tiger cat. 94% probability, according to ResNet.

So yeah, that's how you do it. Now, let's actually try a more interesting example where you want to detect all the objects in the picture, which we didn't talk about in class, object detection. So just create an object detection pipeline. Same thing as before. When you actually run this command, an astonishing some amount of complicated stuff is going on under the hood. And we are all the beneficiaries of that, so thank you.

So yeah. So we have this here, and then we run it through the pipeline. It's looking at all the possible things that might be sitting in the pipeline. The results are hard to read, so let's actually visualize them.

And I got some nice code from this site, for how to visualize them. Let's just reuse it. So yeah. So if you plot the results, look at that. So it has picked up the cat, 100% probability, I guess. The remote, the couch, the other remote, and then the cat. Pretty good, right? Off the shelf, ready to go. No heavy lifting required.

Now, in this case, we are actually putting these boxes, called bounding boxes, around each picture. But what if you actually don't want a bounding box? What if you want to actually find the exact contour of that cat or the remote? No problem. We do something called image segmentation. So let's do an image segmentation pipeline. And run it through. It takes some time. All right. All right. Let's visualize it.

So each object it finds, it gives you a mask. It basically tells you for each object what object it is, and then which pixels are on for that object and off for everything else. It's a mask. It tells you where it stands. And you can see here, it has the first object that's found, is this thing here. And it's perfectly delineated. It's pretty amazing. So we can overlay this on the original image and see that it's found that.

And let's look at other objects. Oh, it is found the remote. That's the second object. And the third remote. And the fourth. You think any other objects are remaining?

STUDENT: The couch.

PROFESSOR: Couch, good. All right, let's find the couch. And look, the couch is pretty good, except that the middle part has gotten confused. All right. But it's still pretty good, right? So, yeah. So Hugging Face is all these things, and so you should definitely check it out if you're not already very familiar with it. So we have one minute left. Any questions?

No questions? OK. All right, folks. See you on Wednesday. Thanks.