Outline

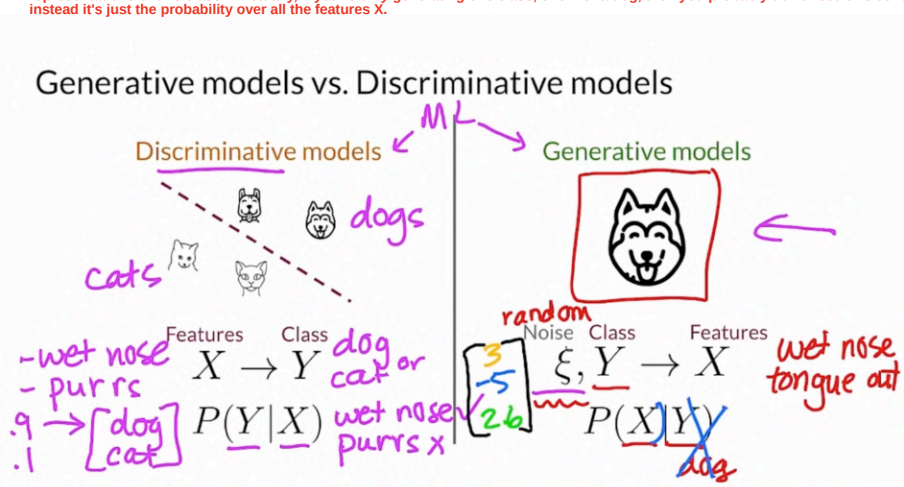
- What are generative models?
- Types of generative models
  including GANs!

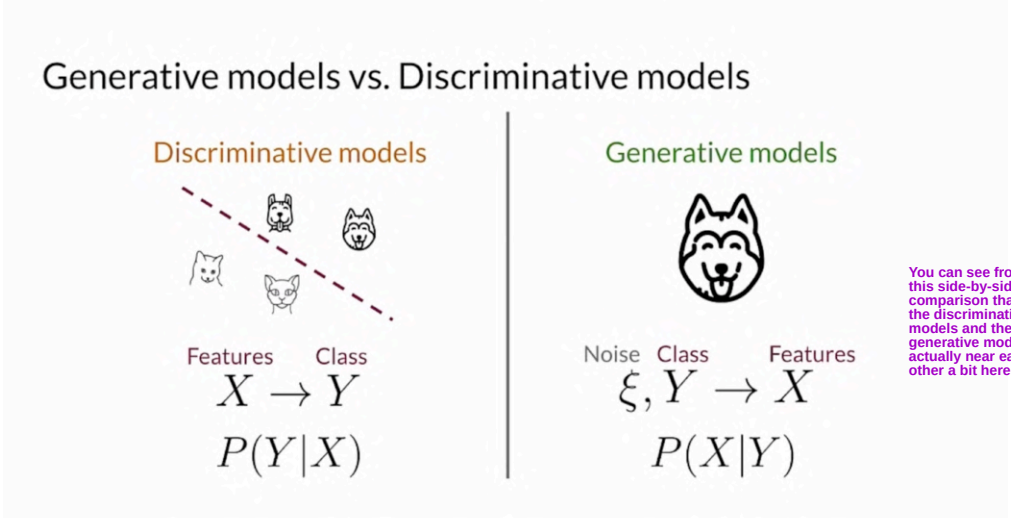
You may be familiar with discriminative models, but you might not have known how they fit into the larger context of machine learning or ML. A discriminative model is one typically used for classification in machine learning. They learn how to distinguish between classes such as dogs and cats, and are often called classifiers. Discriminative models take a set of features X, such as having a wet nose or whether it purrs and from these features determine a category why of whether the image is of a dog or a cat. In other words, they try to model the probability of class Y given a set of features X as having a wet nose, but it doesn't purr, so it's probably a dog.

On the other hand, generative models try to learn how to make a realistic representation of some class. For instance, a realistic picture of a dog you see here. They take some random input represented by the nose here, which could take on the value, let say three, a random number, or negative five, or 2.6 or actually just a vector of all of those values. The point is, the noise represents a random set of values going into the generative model. The generative model also sometimes takes in a class Y such as a dog.

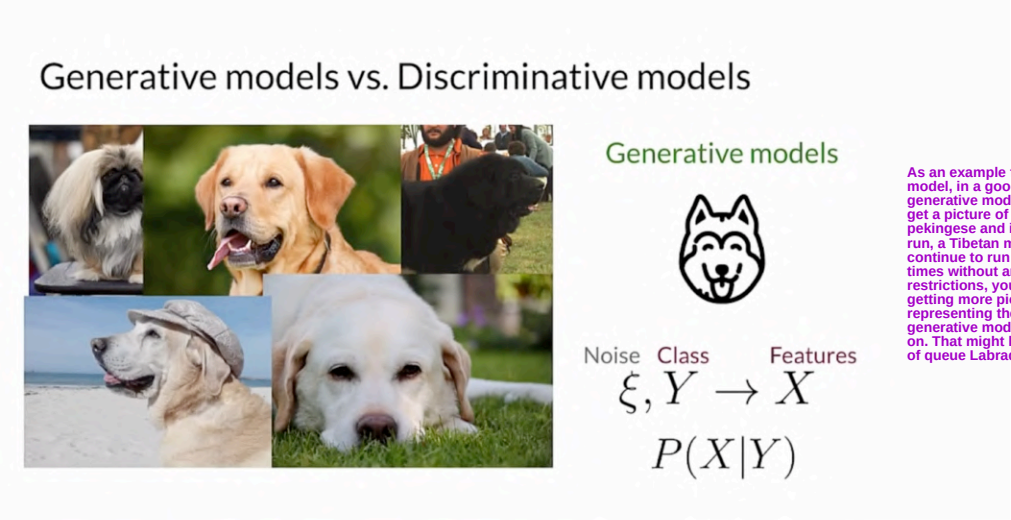
From these inputs, it's goal is to generate a set of features X that look like a realistic dog. So an image of a dog with features such as a wet nose or a tongue sticking out. You might wonder why we need this noise in the first place. Why can't we just tell it, "Hey, generate a dog for me," and then it'll generate a dog. The noise is larger to ensure that what's generated isn't actually the same dog each time and you'll see this theme play out shortly. This is because generating just one dog is no fun and also a little pointless. For now, think of this as some random noise that also goes in as an input.

More generally, generative models try to capture the probability distribution of X, the different features of having a wet nose, the tongue sticking out, maybe pointy ears sometimes but not all the time given that class Y of a dog. With the added noise, these models would generate realistic and diverse representations of this class Y. Actually, if you're only generating one class, one Y of a dog, then you probably don't need this conditioning on Y and instead it's just the probability over all the features X.

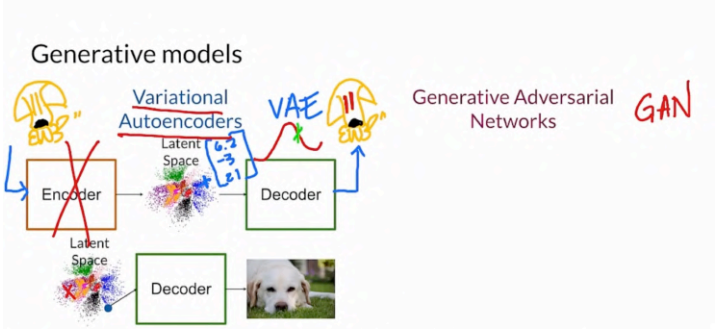




You can see from this side-by-side comparison that the discriminative models and the generative models actually near each other a bit here



As an example for generative model, in a good run of a generative model, you could get a picture of this pekingese and in another run, a Tibetan mastiff. If you continue to run it multiple times without any restrictions, you'll end up getting more pictures representing the dataset your generative model was trained on. That might look like a lot of queue Labrador retrievers.



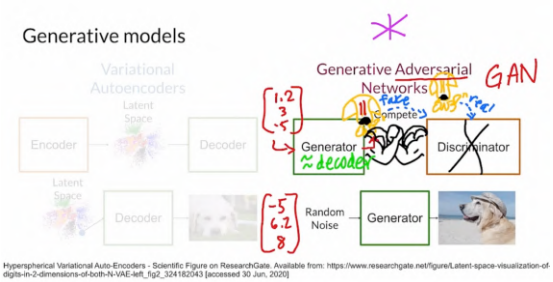
Hyperspherical Variational Auto-Encoders - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Latent-space-visualization-of-digits-in-2-dimensions-of-both-N-VAE-left_fig2_324182043 [accessed 30 Jun, 2020]

There are many types of generative models, and I'll briefly introduce you to the most popular ones. Variational autoencoders or VAE for short, and GANs.

VAEs work with two models, an encoder and a decoder and these are typically neural networks. They learn first by feeding in realistic images into the encoder, such as this really realistic image of a dog that I drew. Then the encoder's job is to find a good way of representing that image in this wanky latent space. Let's say it finds a place right here. Let's say this point in the latent space can be represented by this vector of numbers 6.2, negative three, 21. What the VAE does now is take this latent representation or a point close to it and put it through the decoder. The goal of the decoder is to reconstruct the realistic image that the encoder saw before. That's assuming the decoder has already been trained to be pretty good. But in the beginning, the decoder won't be able to reconstruct the image and maybe the dog produced will have evil eyes.

After training, we actually lop off the encoder and we can pick random points in the latent space, such as here, and the decoder will have learned to produce a realistic image of a dog.

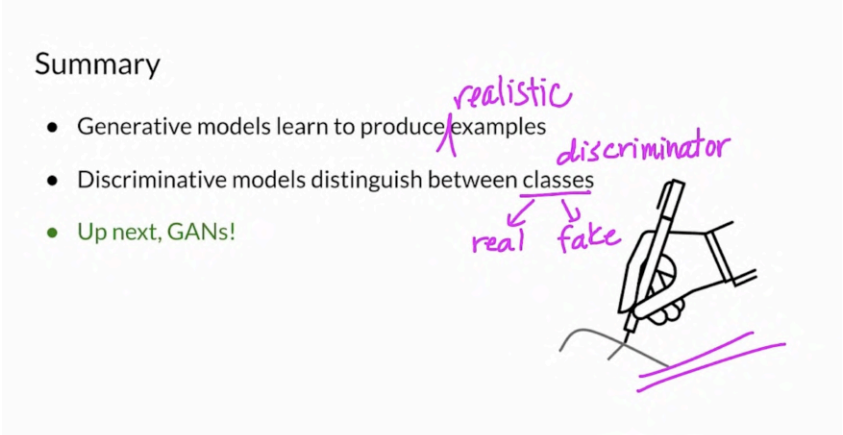
What I just described is largely the autoencoder part or variational autoencoder. The variational part actually inject some noise into this whole model and training process. Instead of having the encoder encode the image into a single point in that latent space, the encoder actually encodes the image onto a whole distribution and then samples a point on that distribution to feed into the decoder to then produce a realistic image. This adds a little bit of noise since different points can be sampled on this distribution.



GANs work in a rather different way. They're composed of two models again, but now there is a generator which generates images like the decoder and a discriminator that's actually a discriminative model hidden inside of this. It's a little bit of inception going on. The generator takes in some random noise input, as you saw before, for example, 1.2, three, negative five as a vector, and that is input into this generator. Of course an optional class of dog, but if we're just generating dog, we don't need to input that. As an output, it can generate that same dog over time, of course, and of course, in the beginning they can also be evil.

The generator's role in some sense it's very similar to the decoder in the VAE. What's different is that there's no guiding encoder this time that determines what noise vector should look like, that's input into the generator. Instead, there's a discriminator looking at fake and real images and simultaneously trying to figure out which ones are real and which ones are fake.

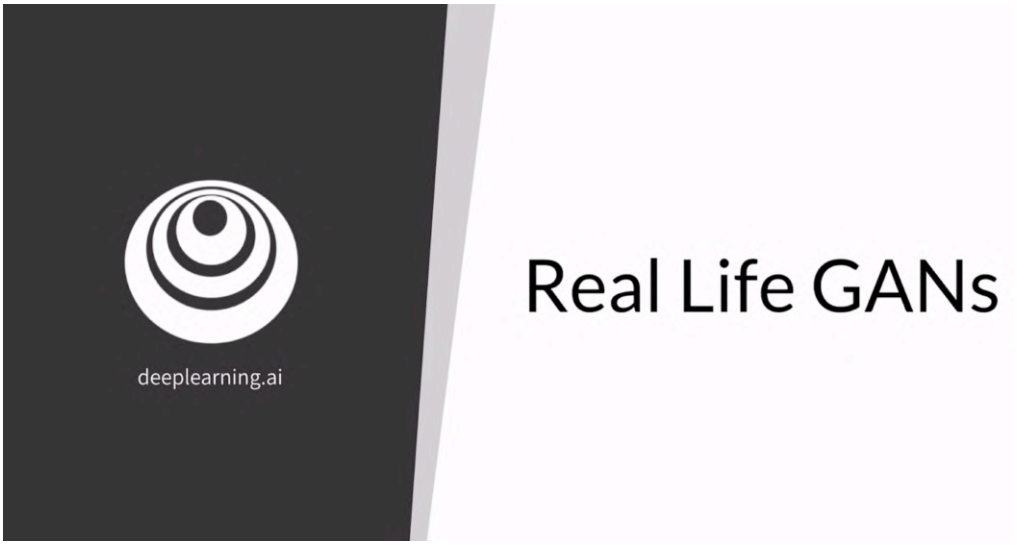
Over time, each model tries to one up each other. These models compete against each other which is why they're called adversarial, in the name generative adversarial networks. You can imagine those muscles growing over time as they compete against each other and learn from each other until they reach a point where we, again, don't need this second model anymore, the discriminator, and the generator can take in any random noise and produce a realistic image. For example, a vector of random numbers negative five, 6.2, and eight can generate this cute Labrador retriever.



In summary, generative models learn to produce realistic examples such as producing those pictures of cute dogs you just saw.

A generative model is an artist who's trying to learn how to create photo-realistic art. Meanwhile, discriminative models distinguish between different classes, such as a dog or a cat. But of course you also saw that a discriminative model can be a sub-component of a generative model, such as the discriminator whose classes are real and fake.

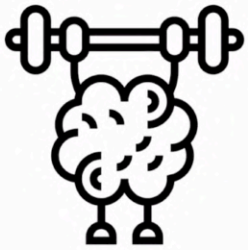
There are many kinds of generative models, but in this specialization, you're going to study GANs.



Outline

- Cool applications of GANs
- Major companies using them

Even though they've only been around since 2014, GANs have already achieved super impressive performance across a multitude of tasks. If you haven't seen some of their results, you're in for a treat.



GANs over time

2014 Black and white

2018 Colored photo-realistic

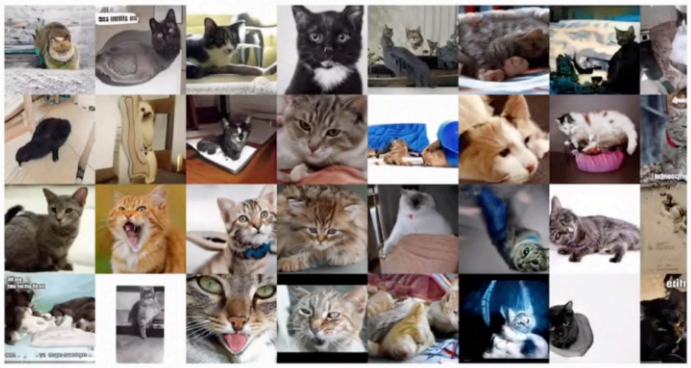
GANs over time

Face Generation
StyleGAN2

These people do not exist!

Karras, Tero, et al. "Analyzing and improving the image quality of stylegan." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020.

GANs over time



StyleGAN2

Mimics the distribution of the training data

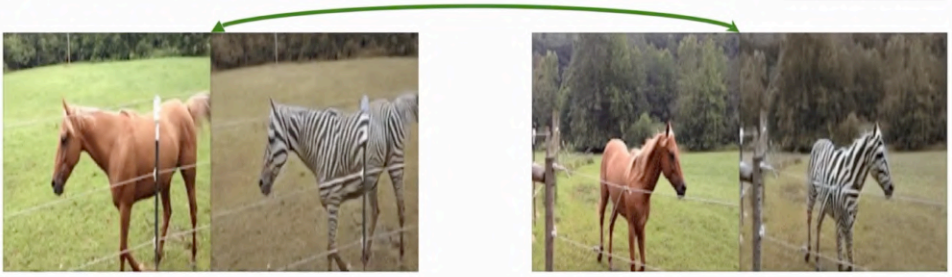
Karras, Tero, et al. "Analyzing and improving the image quality of stylegan." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020.

https://9gag.com/gag/aWYZKWx

GANs for Image translation

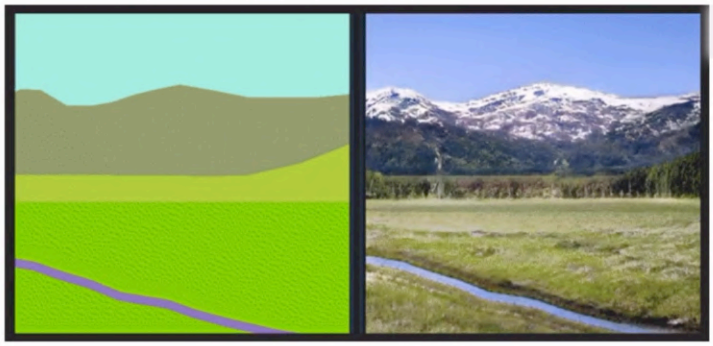
From one domain to another

CycleGAN



Park, Taesung, et al. "Semantic image synthesis with spatially-adaptive normalization." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019.

GANs for Image translation



GauGAN

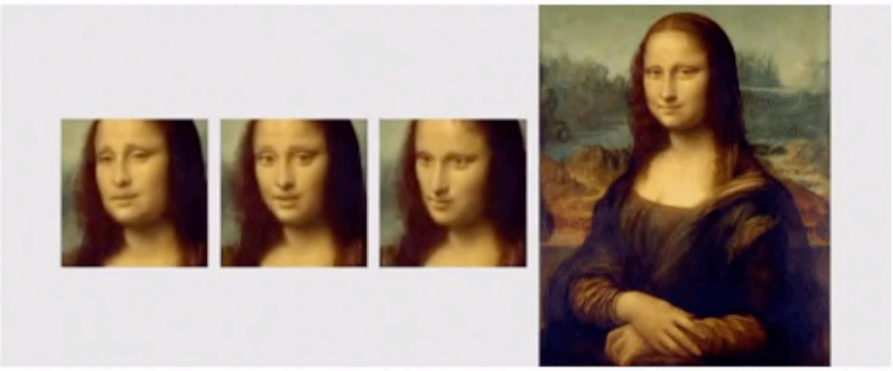
Doodles

↓

Pictures

Park, Taesung, et al. "Semantic image synthesis with spatially-adaptive normalization." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019.

GANs are Magic!



Zakharov, Egor, et al. "Few-shot adversarial learning of realistic neural talking head models." *Proceedings of the IEEE International Conference on Computer Vision*. 2019.

GANs for 3D objects

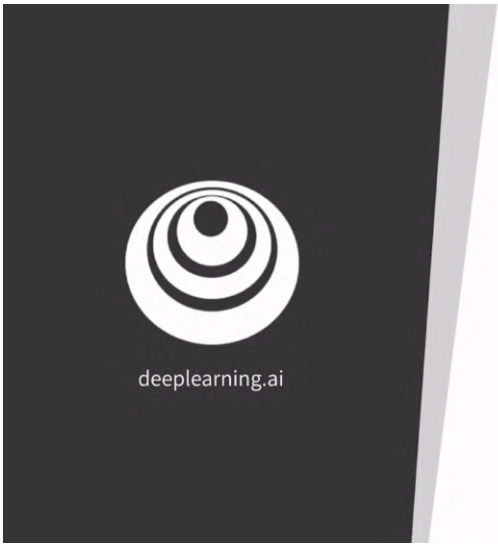
Wu, Jiajun, et al. "Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling." *Advances in neural information processing systems*. 2016.

Companies using GANs

Adobe	Next-gen Photoshop
Google	Text Generation
IBM	Data Augmentation
Snapchat, TikTok	Image Filters
Disney	Superresolution

Summary

- GANs' performance is rapidly improving
- Huge opportunity to work in this space!
- Major companies are using them



Intuition Behind GANs

Outline

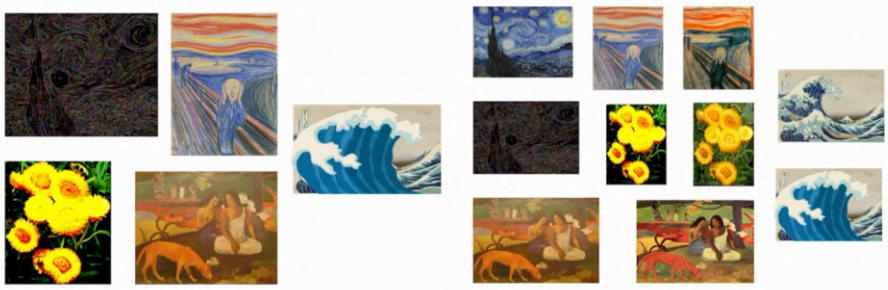
Recall that GANs have two components, one's called the generator and the other is called the discriminator. And these are typically two different neural networks.

- The goal of the generator and the discriminator
- The competition between them

Generative Adversarial Network

Generator learns to make *fakes* that look **real**

Discriminator learns to distinguish **real** from *fake*



The generator learns to generate fakes that look real, to fool the discriminator. And the discriminator learns to distinguish between what's real and what's fake.

So you can think of the generator as a painting forger and the discriminator as an art inspector.

So the generator forges fake images to try to look as realistic as possible, and it does this in the hopes of fooling the discriminator. So you can see here Starry Night and Scream, they look pretty good.

Meanwhile the discriminator here looks for a pile of both real famous paintings and the fake ones created by the generator. And tries to tell which ones are real and which ones are fake.

Generative Adversarial Network

Discriminator learns to distinguish **real** from *fake*

Fake

Real

So here are the ones bordered with that pink purplish color are fake and the ones that are bordered with green are real.

Generative Adversarial Network

Generator learns to make *fakes* that look **real**

Discriminator learns to distinguish **real** from *fake*

And as you can see a bit from this set up, in order to fool the discriminator, the generator will try to forge paintings that look more like the real ones. And in order to catch the generator, the discriminator will try to learn how to not get fooled, even by the closest replica. So to start this game, all you need is a collection of these real images, like some famous paintings. If you want the generator to paint famous paintings.

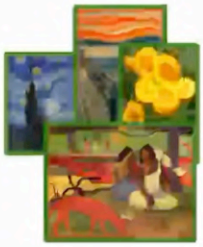
And so at the beginning of this game, the generator actually isn't very sophisticated. It doesn't know how to produce real looking artwork, so I'm going to represent the generator as this meme of a dog trying to paint.

Additionally, the generator isn't allowed to see the real images. It doesn't know how this painting should even look. So this is really, really tough for the generator, especially in the beginning. So at the very beginning, the elementary generator initially just paint a masterpiece of scribbles And don't judge her, nobody ever told her what to do and what to generate here.

And so your other beginning component is actually an elementary discriminator that doesn't know for sure what's real and what's fake.

Generative Adversarial Network

Discriminator learns to distinguish **real** from *fake*



And represented here as a dog with a beret on, trying to be an art critic here. But in this case, he's allowed to look at the real artwork. It's just jumbled up with the fake ones as well, and he doesn't know which one is which. That's for him to figure out and learn how to decide.

The Game is On!

5% Real

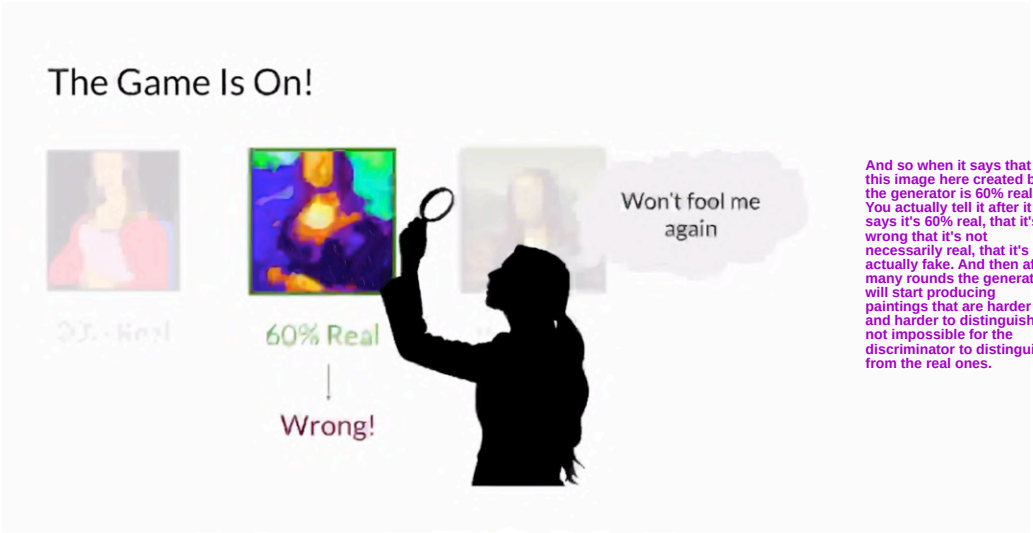
40% Real

So to start the competition, you train the discriminator using the real artwork so it's able to know which images are actually real. So after it decides hm, maybe this looks real, you actually tell it yes, that's real or no, that's fake. This way you can get a discriminator that's able to differentiate a poorly drawn image like this, from the ones that are slightly better and eventually also the real ones.

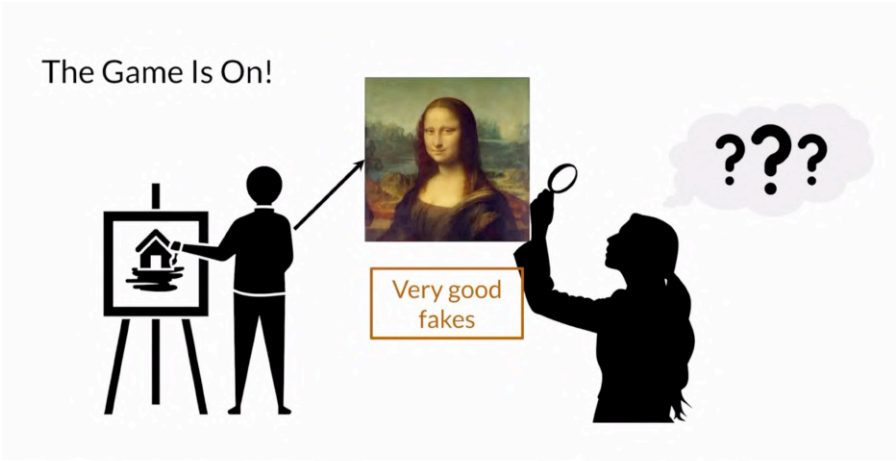
Of course, the discriminator gets these all jumbled up, so it doesn't know up front which ones are real, which ones are fake. But you do tell it as it learns which ones are real and which ones are fake, whether it's right or wrong in determining those two classes. And when the generator produces a batch of paintings, the generator will know in what direction to go on and improve, by looking at the scores assigned to her work by the discriminator.

So this one looks a little bit more real, so maybe the generator here will start painting a little bit more realistically. Towards the face of maybe the Mona Lisa, not quite there yet, but almost.

And the discriminator also improves over time because it receives more and more realistic images at each round from the generator. And it always remember, receives both real and fake images, all jumbled up in a pile. But essentially it tries to develop a keener and keener eye as these images get better.

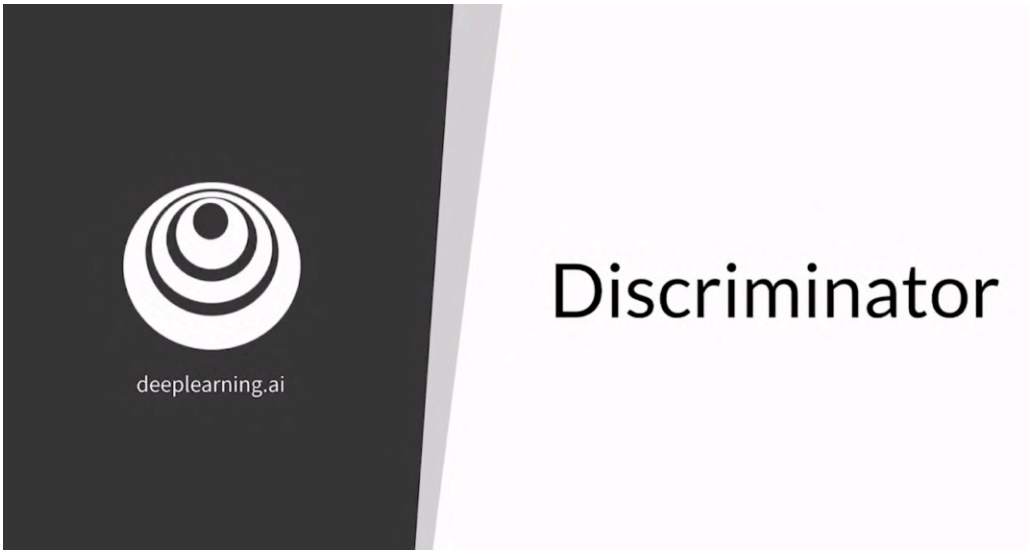


And so when it says that this image here created by the generator is 60% real. You actually tell it after it says it's 60% real, that it's wrong that it's not necessarily real, that it's actually fake. And then after many rounds the generator, will start producing paintings that are harder and harder to distinguish. If not impossible for the discriminator to distinguish from the real ones.



And then at this point, when you, the person who wants a good generator to generate awesome fake images. When you are happy with the result of this generator, the game will end.

In summary from this lesson, it's important to take away that the goal of the generator is to produce fakes that look real to the discriminator. While the discriminator's goal is to tell the generator's fakes apart from real examples that you give it. And so both models learn from their competition with each other, until the examples produced by the generator are good enough to fool the discriminator.



Outline

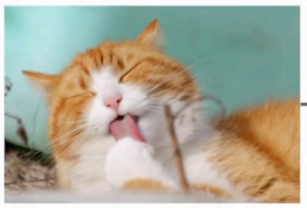
- Review of classifiers
- The role of classifiers in terms of probability
- Discriminator

So GANs are composed of two models, the discriminator and the generator. In this video, I'll show you first how the discriminator works.

The discriminator is a type of classifier, so I'll start with a refresher on what that is. Then you'll see in probabilistic terms what classifiers learn to model, and at the end of this video, I'll show you how it all translates to the GANs discriminator.

Classifiers

Distinguish between different classes



Classifier

Turtle

Bird

Cat

Dog

Fish

So as a quick recap, the goal of the classifier is to distinguish between different classes. So given this image of a cat, the classifier should be able to tell that it's a cat and a dog, for example. In fact, it can learn to differentiate cats from multiple different classes and this depends on which classes you wanted to differentiate. Probably the simplest case is probably just cat and not cat.

Classifiers

Distinguish between different classes

"It meows, and plays with yarn"

Classifier

Turtle

Bird

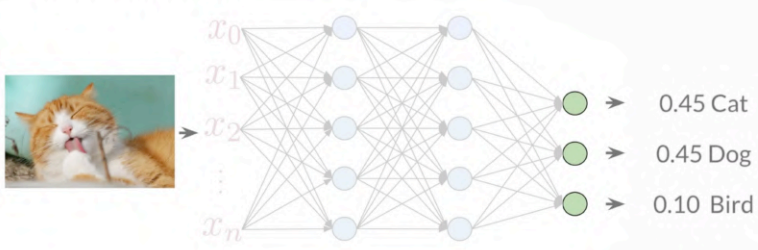
Cat

Dog

Fish

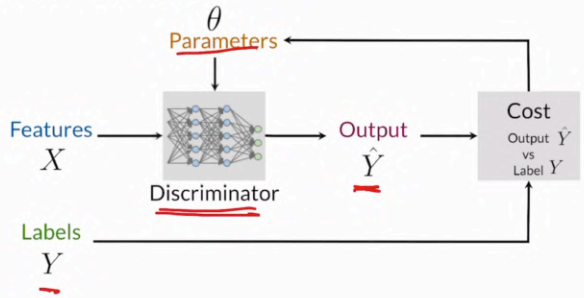
Classifiers aren't limited to determining image classes, so you could have this piece of text right here, it meows and plays with yarn to classify into a cat. You could have a video of a cat purring, all sorts of different things here.

Neural Networks



So one type of model for a classifier is using a neural network and this neural network can taken some features X. For example here there's X0, X1, X2, all the way up to Xn, so n different features. It computes a series of nonlinearities, which I'll get into later and outputs the probabilities for a set of categories. And, it thinks that it's 45% likelihood of this image being a cat, 45% likelihood of this image being a dog, and 10% likelihood of this image being a bird. And in the beginning, the model probably won't know how to classify correctly, so that's why cat and dog are probably on par with each other here. And it learns overtime trying to improve its predictions according to the true labels from the data. So you'll tell it at the end, no. This is 100% cat, 0% dog, 0% bird.

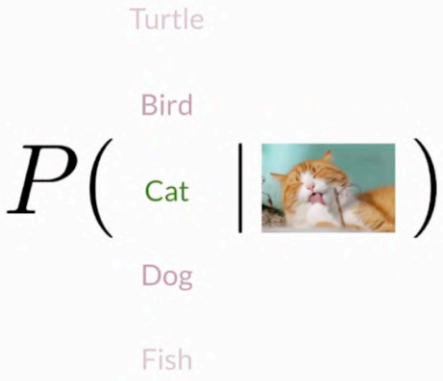
Classifiers (training)



So this learning process can be summarized as follows. You have some input features X, such as that it purrs, and it likes to play with yarn, and a set of labels wise associated with each of your classes. For example, a cat, a dog, and a bird. And you use your neural network, which takes in those features and learns these set of parameters, which I'll call theta. And these are each of those nodes you see in that neural network. And these weights change overtime as it learns what a cat looks like and what a dog looks like and what a bird looks like. And these parameters data are trying to map these features X to those labels Y. And those predictions you'll call Y hat because they're not exactly the exact Y labels. They're trying to be the Y labels. And so the goal is to reach a point where the difference between the true values Y in the predictions Y hat is minimized.

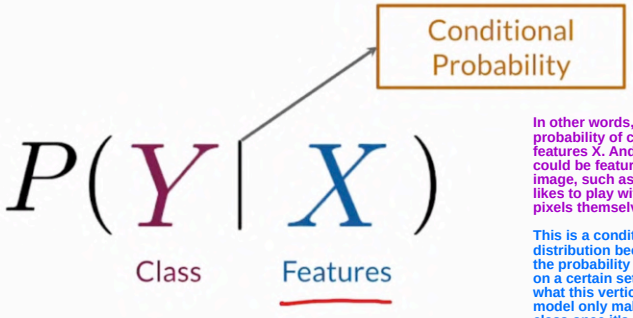
This is where a cost function comes in, and it's computed by comparing how closely Y hat is to Y. And that's the goal of this cost function, which tells this discriminative model. This neural network, how close it is to classifying cat correctly and dog correctly and bird correctly. So from this cost function, you can update those parameters. The nodes in that neural network according to the gradient of this cost function. And that just means generally which direction those parameters should go to try to get to the right answer, to try to get to a Y hat, that's as close as possible to Y. And then you repeat this process until your classifier is in good shape.

Classifiers



So let's take a little ride into math land where the goal of the discriminator is to model the probability of each class. For example, not just cat here, but turtle, bird, dog, and fish as well. Given a set of input features, for example, here is an image of a cat. So given this image of a cat, which class is this?

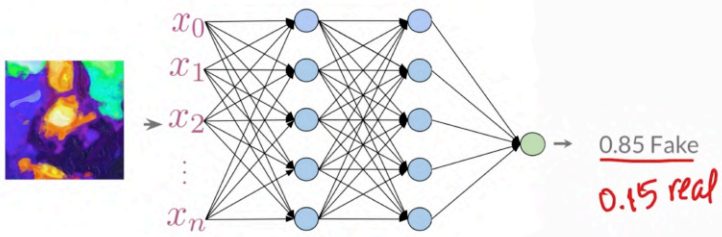
Classifiers



In other words, this is modeling the probability of class Y given input features X. And again, these features could be features extracted from this image, such as that it purrs and that it likes to play with yarn, but also the pixels themselves.

This is a conditional probability distribution because it's predicting the probability of class Y conditioned on a certain set of features, which is what this vertical bar is for. So the model only makes a prediction of a class once it's seen the input features. And in this case, it's in image.

Discriminator

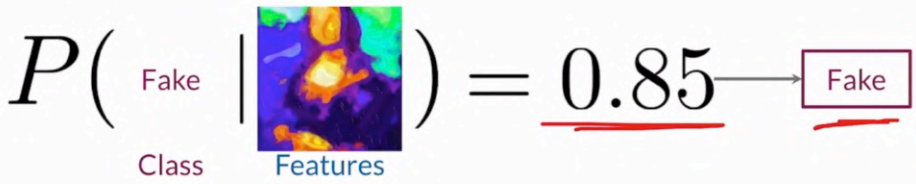


And now, bring it back to the GAN context. The discriminator is a classifier that inspects the examples. They're fake examples, the real examples, and determines whether they belong to the real or fake class. So taking in this fake Mona Lisa here, and instead of determining whether there is a cat, dog, or bird in this image, determining how fake this image is, and here it thinks it's 85% fake.

In probabilistic terms, the discriminator models the probability of an example being fake given a set of inputs X. For instance, it will look at this picture of a fake Mona Lisa and determined that with 85% probability it isn't the real one. And as a simplification, it will be classified as fake. And you can also think of this as 0.15 real and rewrite this as a probability over real given that image.

Discriminator

The discriminator models the probability of an example being fake given that set of inputs X. For instance, in a look at a picture of a Mona Lisa or a fake Mona Lisa and determine that with 85% probability this isn't the real one, 0.85 fake. So in this case, it will be classified as fake and that information. And not just this fake information, but this 0.85 will be given to the generator to improve its efforts.



Summary

- The discriminator is a classifier
- It learns the probability of class Y (**real** or *fake*) given features X
- The probabilities are the feedback for the generator

In summary, the discriminator is a type of classifier that learns to model the probability of an example being real or fake given that set of input features, like RGB pixel values for images. The output probabilities from the discriminator are the ones that help the generator learn to produce better looking examples overtime

deeplearning.ai

Generator

Outline

- What the generator does
- How it improves its performance
- Generator in terms of probability

The generator and again, is like it's heart. It's a model that's used to generate examples and the one that you should be invested in and helping achieve a really high performance at the end of the training process.

First, revisit the role of the generator and you'll see how it's able to improve its performance. Then see what it models in terms of probability.

Generator

Turtle

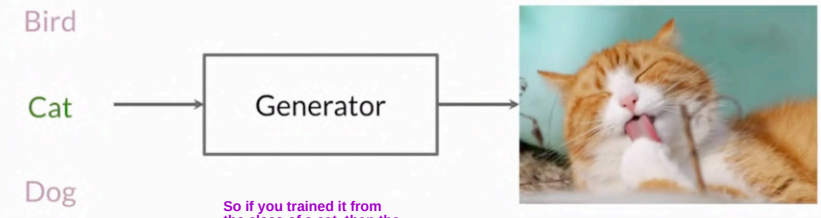
Bird

Cat

Dog

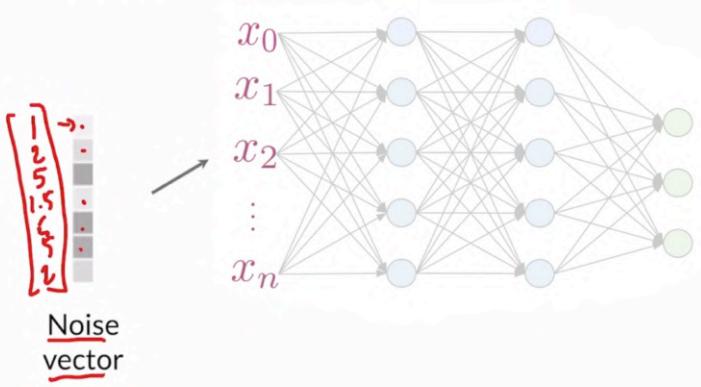
Fish

Generates examples of the class



So if you trained it from the class of a cat, then the generator will do some computations and output a representation of a cat that looks real.

Neural Networks

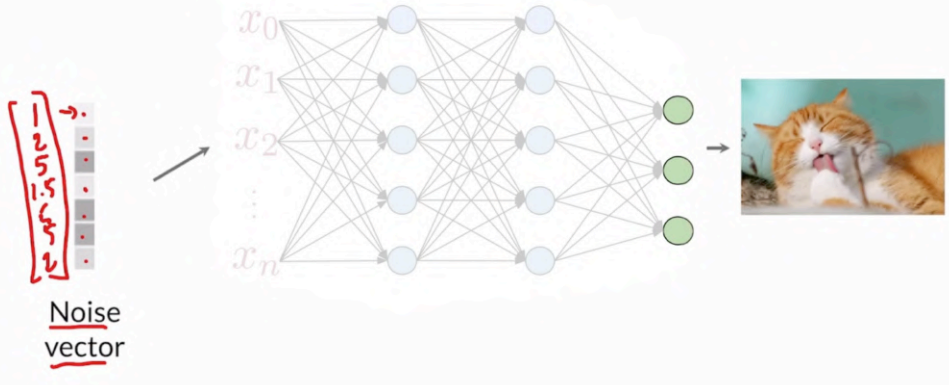


Noise vector

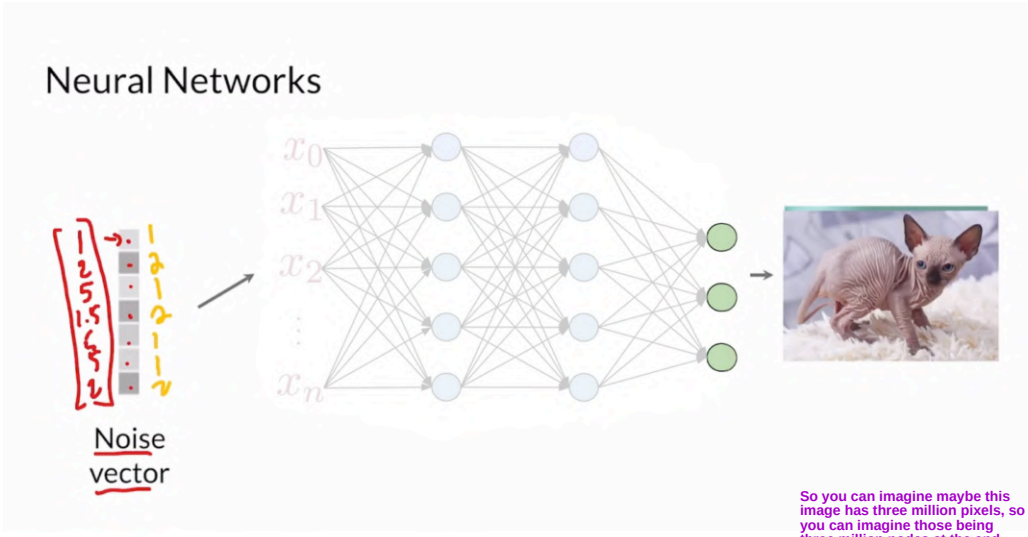
So ideally, the generator won't output the same cat at every run, and so to ensure it's able to produce different examples every single time, you actually will input different sets of random values, also known as a noise vector. So here this noise vector is actually just a set of values where these differently shaded cells are just different values. So you can think of this as 1, 2, 5, 1.5, 5, 5, 2. Then this noise vector is fed in as input, sometimes with our class y for cat into the generators neural network. This means that these features, x_0, x_1, x_2, all the way up to x_n, include the class, as well as, the numbers in this noise vector.

So then the generator in this neural network will compute a series of nonlinearities from those inputs and return some variables that look like a q, brown and white cat and run. So here instead of different classes, it's output will actually be an image.

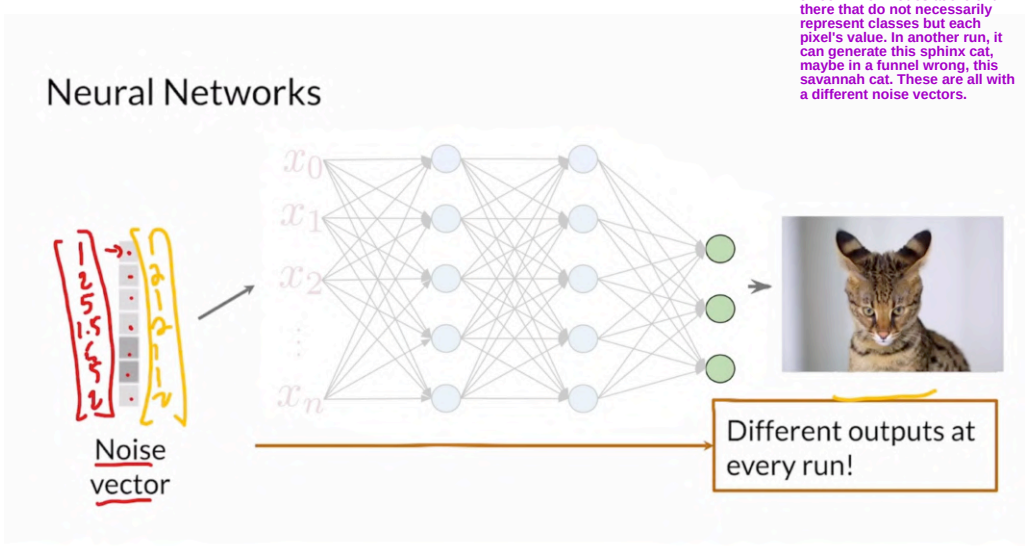
Neural Networks

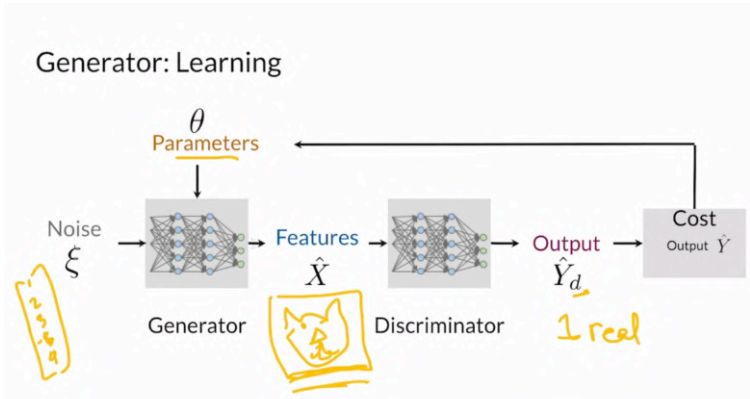


Noise vector



So you can imagine maybe this image has three million pixels, so you can imagine those being three million nodes at the end there that do not necessarily represent classes but each pixel's value. In another run, it can generate this sphinx cat, maybe in a funnel wrong, this savannah cat. These are all with a different noise vectors.



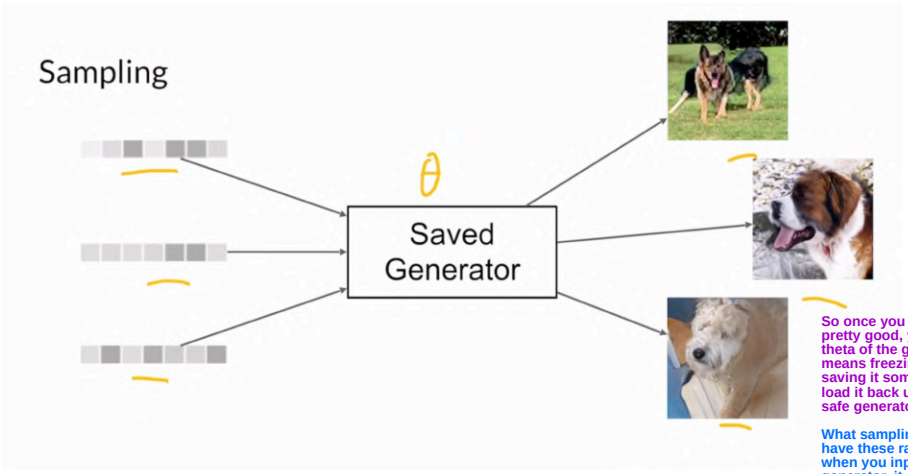


So let's first consider conceptually how the generator improves over time. First, you have a noise vector or those random input values you saw, which I'm going to represent with this Greek symbol here of phi. You pass this into a generator represented by a neural network to produce a set of features that can pose an image of a cat or an attempt at a cat. For example, your generator might generate this.

In this image, x-hat is fed into the discriminator, which determines how real and how fake it thinks it is based on its inspection of it. After that, from what the discriminator thinks of it, which is this y-hat with a D here representing that it's the discriminators predictions, you can compute a cost function that basically looks at how far the examples produced by the generator are being considered real by the discriminator because the generator wants this to seem as real as possible.

So basically, the generator wants y-hat to be as close to one, meaning real as possible. Whereas, the discriminator is trying to get this to be zero, fake.

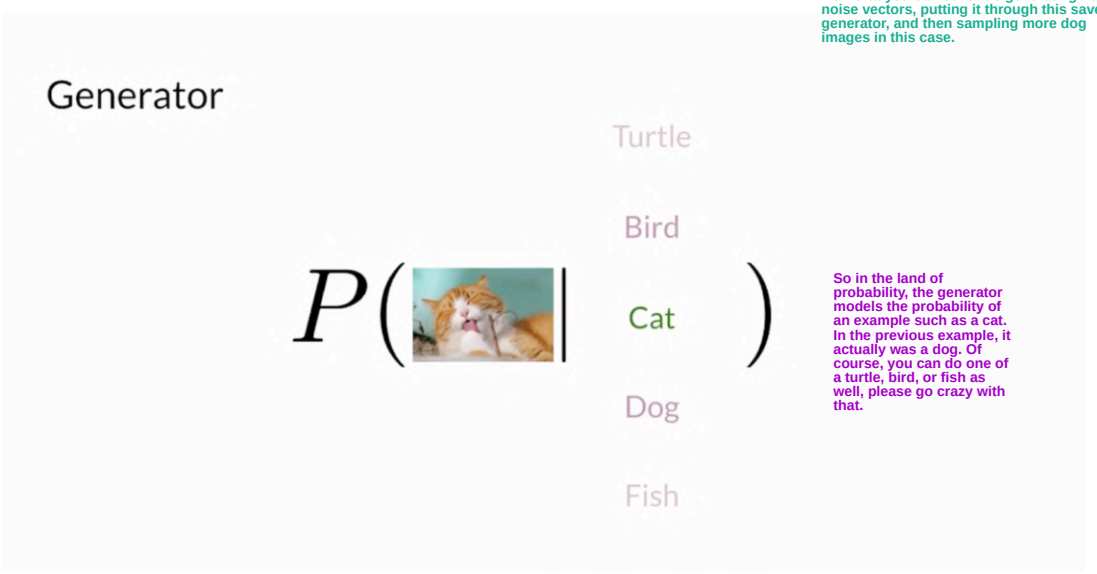
It uses the difference between these two to then update the parameters of the generator, and that gets it to improve over time and know which direction to move it's parameters to generate something that looks more real and will fool the discriminator.



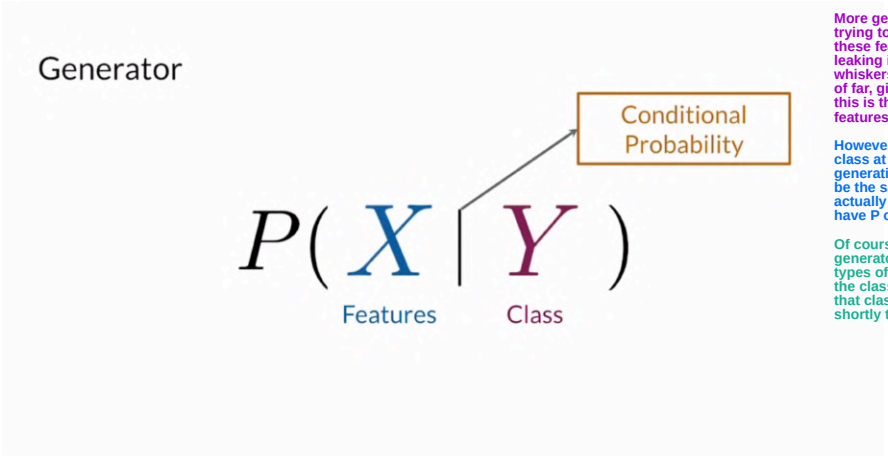
So once you get a generator that looks pretty good, you can save the parameters theta of the generator. That typically means freezing those theta values and saving it somewhere and then you can load it back up and then sample from this safe generator.

What sampling basically means is you have these random noise vectors, and when you input that into the saved generator, it can generate all sorts of different examples. So this saved generator was not trained on cats, I hope, because it's generating dogs.

Now that you can continue generating new noise vectors, putting it through this saved generator, and then sampling more dog images in this case.



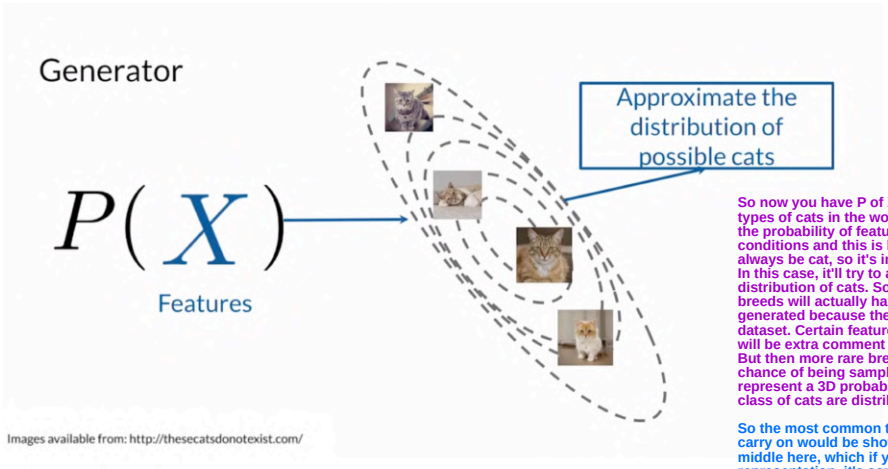
So in the land of probability, the generator models the probability of an example such as a cat. In the previous example, it actually was a dog. Of course, you can do one of a turtle, bird, or fish as well, please go crazy with that.



More generally, the generator is trying to model the probability of these features of a cat, such as leaking it's paw or having cute whiskers are having different types of far, given this class y of a cat. So this is the conditional probability of features X given the class Y.

However, since we only have one class at the moment for let's say just generating cats, then Y will always be the same and so you don't actually need this here so that you have P of X that you're modeling.

Of course, if you want your generator to learn across all different types of classes and you care about the class, then you do need to put that class in. You'll see more on this shortly through our Course 1.



So now you have P of X across all the different types of cats in the world. The generator will model the probability of features X without any additional conditions and this is because the class Y will always be cat, so it's implicit for all probabilities X. In this case, it'll try to approximate the real distribution of cats. So the most common cat breeds will actually have more chances of being generated because they're more common in the dataset. Certain features such as having pointy ears will be extra comment because most cats have that. But then more rare breeds will have a less likely chance of being sampled. so these lines here just represent a 3D probability distribution of how the class of cats are distributed.

So the most common types of features that cats carry on would be shown and sampled in this middle here, which if you think about this as a 3D representation, it's coming out at you. Then a rare breeds or rare looking hats will be at the edges. So this means that the most common cap rates will have more chances to be generated, while the less common ones like the sphinx will be produced much more rarely. You'll see in future sections how to control the sampling process and get what you would like, but right now, the generators drop is just to model how cats are in the natural world.

Summary

- The generator produces fake data
- It learns the probability of features X
- The generator takes as input noise (random features)

So to wrap up, the generator produces fake data that tries to look real. It learns to mimic that distribution of features X from the class of your data. In order to produce different outputs each time it takes random features as input. In this week's assignment, you'll build a gun that generates images of numerical digits. It has the same setup, you just give it random noise and it can produce all these different digits that are handwritten like this five and this eight.

This is cool because handwriting doesn't look perfect, it doesn't look the same each time and it would be able to model and generate the range of different 5s and 8s and all sorts of different digits from this data set of handwritten digits

Outline

- Binary Cross Entropy (BCE) Loss equation by parts
- How it looks graphically

Binary Cross Entropy function, or BCE for short, is used for training Gantt. It's useful for these models, because it's especially designed for classification tasks, where there are two categories like, real and fake. In this video, I'll show you the equation used to get the BCE loss, and what every part of the equation means. At the end, I'll wrap up this lesson by showing you how the BCE cost function looks for different labels.

BCE Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$

BCE Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$

Prediction

Label

Features

Parameters

Average loss of the whole batch

First at the beginning here you see a summation sign from 1 to m, as well as dividing by that m. This basically just means summing over the variable m, which is actually the number of examples in the entire batch, and taking the average of those examples. Taking the average cost across this mini-batch. I'll go over what this negative sign means later on.

Then h denotes the predictions made by the model. You see that here, y is the labels for the different examples. These are the true labels of whether something is real or fake. For example, if real could be a label of 1, and fake or be a label of 0. X are the features that are passed in through the prediction, so this could be an image, and theta are the parameters of whatever is computing that production. In this case, it's probably going to be the discriminator. As the parameters of the discriminator, looking at those features, and that's doing this, h of x, theta. Often you'll see this also written as h of x;theta, and that just means parameterized by theta. That's slightly more accurate, but you don't have to worry about that now.

BCE Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))\right]$$

In these brackets, you can break these down into two different terms. Let's look at each of them.

BCE Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))\right]$$

The term on the left, is the product of the true label y times the log of the prediction, which is h of x, the features parameterized by theta for the model. To understand what this value is trying to get at, let's look at some examples.

The case where the label 0, so let's say this means it's fake, and you have any prediction here, then this value actually results in 0. If you have a prediction of 1, let's say that's real, and you have a really high prediction that is close to 1, of 0.99, then you also get a value that's close to 0.

In the case where it actually is real, but your prediction is terrible, and it's 0, so far from 1, you think it's fake, but it's actually real, then this value is extremely large. This is largely caused by the log there.

What this is trying to say, is that in the case when the true prediction is 0, this term doesn't matter, it just goes to 0. This term is mainly for when the prediction is actually just 1, and it makes it 0 if your prediction is good, and it makes it negative infinity if your prediction is bad

$y^{(i)}$	$h(x^{(i)},\theta)$	$y^{(i)}\log h(x^{(i)},\theta)$
0	any	0
1	0.99	~0
1	~0	-inf

Relevant when the label is 1

Fake

real

Now looking at the second term, it looks very similar, save some of these minus signs in here. In this case, if your label is 1, then 1 minus y is equal to 0. Actually if your prediction is anything, this will evaluate to 0, and if your prediction is saying, Hey, that's pretty fake, gets close to 0, then this value is close to 0.

However, if it's fake, but your prediction is really far off, and thinks it's real, then this term evaluates to negative infinity. Basically, each of these terms evaluates to negative infinity if for their relevant label, the prediction is really bad.

That brings us to this negative sign a little bit. If either of these values evaluates to something really big in the negative direction, this negative sign is crucial to making sure that is a positive number and positive infinity. Because for our cost function, what you typically want is a high-value being bad, and your neural network is trying to reduce this value as much as possible. Getting predictions that are closer, evaluating to 0 makes sense here, because you want to minimize your cost function as you learn.

In summary, one term in the cost function is relevant when the label 0, the other one is relevant when it's 1, and in either case, the logarithm of a value between 1-0 was calculated, which returns that negative result.

BCE Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))\right]$$

$y^{(i)}$	$h(x^{(i)},\theta)$	$(1-y^{(i)})\log(1-h(x^{(i)},\theta))$
1	any	0
0	0.01	~0
0	~1	-inf

Relevant when the label is 0

Fake

BCE Cost Function

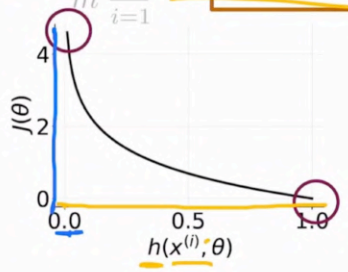
$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$

Ensures that the cost is always greater or equal to 0

That's why you want this negative term at the beginning, to make sure that this is high, or greater than, or equal to 0.

BCE Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$



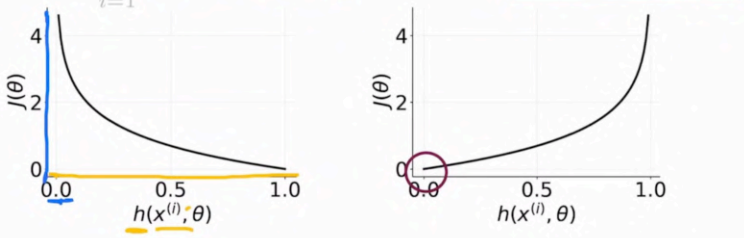
Now I'll show you what the loss function looks like for each of the labels over all possible predictions. In this plot, you can have your prediction value on the x-axis, where h is your model, and gives a prediction based on x parameterized by theta. The loss associated with that training example is on the y-axis.

In this case, the loss simplifies to the negative log of the prediction. When the prediction is close to 1, here at the tail, the loss is close to 0 because your prediction is close to the label. Good job here, this is good. When the prediction is close to 0 out here, unfortunately your loss approaches infinity, so a really high value because the prediction and the label are very different.

The opposite is true when the label is 0, and the loss function reduces to the negative log of 1 minus that prediction.

BCE Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$



When the prediction is close to 0, the loss is also close to 0. That means you're doing great. But when your prediction is closer to 1, but the ground truth is 0, it will approach infinity again.

Summary

- The BCE cost function has two parts (one relevant for each class)
- Close to zero when the label and the prediction are similar
- Approaches infinity when the label and the prediction are different

In summary, the BCE cost function has two main terms that are relevant for each of the classes. Whether prediction and the label are similar, the BCE loss is close to 0. When they're very different, that BCE loss approaches infinity.

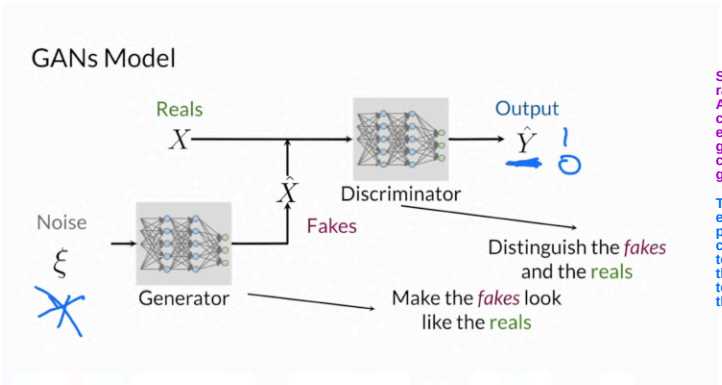
The BCE loss is performed across a mini-batch of several examples, let say n examples, maybe five examples where n equals 5. It takes the average of all those five examples. Each of those examples can be different. One of them can be 1, the other four could be 0, for their different classes.

Putting It All Together

Outline

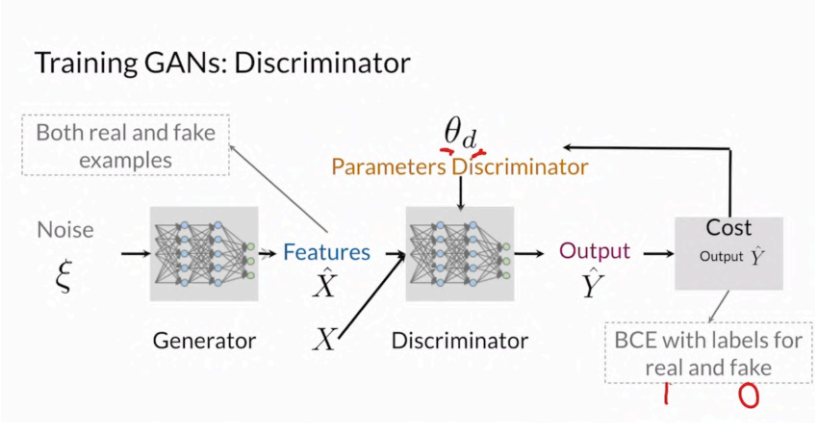
- How the whole architecture looks
- How to train GANs

So first I'll share a representation of what a GAN architecture might look like. And then I'll show you how to train a GAN by alternating the training of the discriminator with the training of the generator.



So recall that in a basic GAN, the generator takes a random noise as input, with this side Greek letter. And this generator that produces fake examples called X hat. So and these are fake images, for example. And I don't need to pass a class of generator, but I can if I'm generating lots of different classes. For now, I'm not going to pass it class of the generator, I'll show you how this works later.

Then the generated examples, along with some real examples are passed to the discriminator. In this probability, this output from the discriminator is called Y hat. And so the goal of the discriminator is to distinguish between the generated examples and the real examples. While the goal of the generator is to fool the discriminator by producing fake examples that look as real as possible.

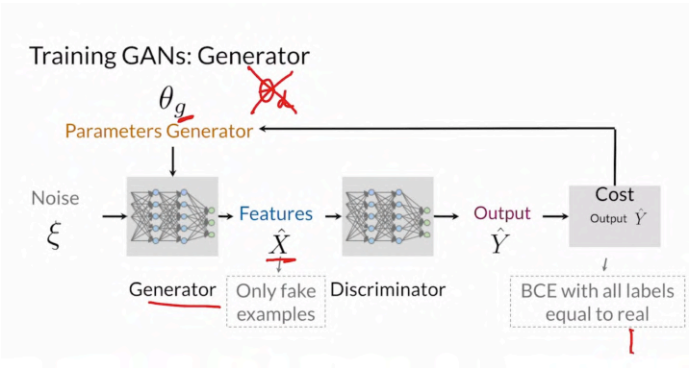


So to train a basic GAN, you alternate the training on the generator and the discriminator. So let's first start with how the discriminator works.

So first of course you get some fake examples X hat produced by the generator from that input noise. And then those examples, the fake ones, X hat and real ones X are both passed into the discriminator without telling the discriminator just yet which ones are real and which ones are fake. And then the discriminator makes predictions Y hat of which ones are real and which ones are fake. Or more specifically a probability of score of how fake and how real each of these images are.

After that, the predictions are compared using that BCE loss with the desired labels for fake and real. And that helps update its parameters or theta d, where d represents parameters for the discriminator.

And so this only updates the parameters of the discriminator, only this one neural network, and not the generator.



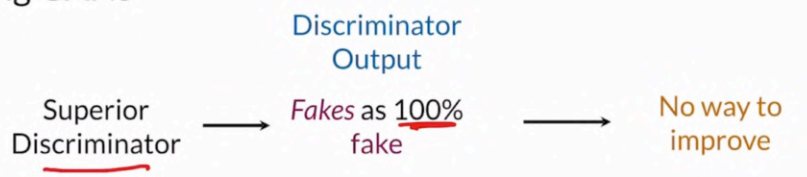
All right, so for the generator it first generates a few fake examples, again X hat, and this is from the input noise. And then these are again passed into the discriminator. But in this case the generator only sees its own fake examples. So it doesn't see the real examples at all. So it only knows that these are being passed to some discriminator.

Then the discriminator makes predictions. Y hat of how real or fake these are. And after that the predictions are compared using the BCE loss with all the labels equal to real. Because the generator is trying to get these fake images to be equal to real or label of 1 as closely as possible.

So this is where it's a little bit tricky and a little bit different between the generator and discriminator. Discriminator wants the fake examples to seem as fake as possible, but the generator wants fake examples to seem as real as possible. That is, it wants to fool the discriminator. And so, after computing the cost, the gradient is then propagated backwards and the parameters of the generator or theta sub g are updated.

Again now, it's only the generator, this one neural network that is getting updated in this process, not the discriminator.

Training GANs



So as you alternate their training, only one model is trained at a time, while the other one is held constant. So in training GANs in this alternating fashion, it's important to keep in mind that both models should improve together and should be kept at similar skill levels from the beginning of training. And so the reasoning behind this is if you had a discriminator that is superior than the generator, like super, super good, you'll get predictions from it telling you that all the fake examples are 100% fake. Well, that's not useful for the generator, the generator doesn't know how to improve. Everything just looks super fake, there isn't anything telling it to know which direction to go in. Maybe to add something a little bit more realistic and how to learn over time.

Meanwhile, if you had a superior generator that completely outskills the discriminator, you'll get predictions telling you that all the generated images are 100% real.

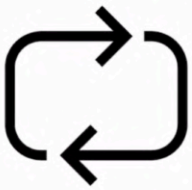
So when training GANs in this alternating fashion, it's important to keep in mind that both models should improve together. And should be kept at similar skill levels from the beginning of training. And the reasoning behind this is largely because of the discriminator. The discriminator has a much easier task, it's just trying to figure out which ones are real, which ones are fake, as opposed to model the entire space of what a class could look like. What all cats could look like. And so the discriminator's job is much easier than the generator's.

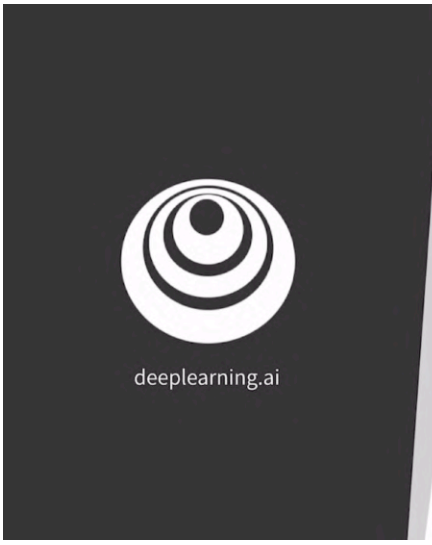
One common issue is having a superior discriminator, having this discriminator learn too quickly. And when it learns too quickly and it suddenly looks at a fake image and says, this is 100% fake, I know this is fake. But this 100% is not useful for the generator at all because it doesn't know which way to grow and learn. And so having output from the discriminator be much more informative, like 0.87 fake or 0.2 fake as opposed to just 100% fake. One, probability one fake, is much more informative to the generator in terms of updating its weights and having it learn to generate realistic images over time.

Summary

- GANs train in an alternating fashion
- The two models should always be at a similar “skill” level

Thee most important takeaway here for you is that GAN training works in this alternating fashion typically. And in order to improve that training over time, you have to keep the scale of both the generator and discriminator close to each other to maintain a good training process. And of course, if your generator is already really good thing, you're done.





Intro to PyTorch (Optional)

Outline

- Comparison with TensorFlow
- Defining Models
- Training

Defining Models in PyTorch

```
import torch
from torch import nn
```

Custom layers for DL

```
class LogisticRegression(nn.Module):
    def __init__(self, in):
            super().__init__()
        self.log_reg = nn.Sequential(
          nn.Linear(in,  1),
          nn.Sigmoid()
          )
    def forward(self, x):
        return self.log_reg(x)
```

Define the model as a class

Initialization method with parameters

Definition of the architecture

Forward computation of the model with inputs x

Training Models In PyTorch

```
model = LogisticRegression(16)
```

Initialization of the model

```
criterion = nn.BCELoss()
```

Cost function

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

Optimizer

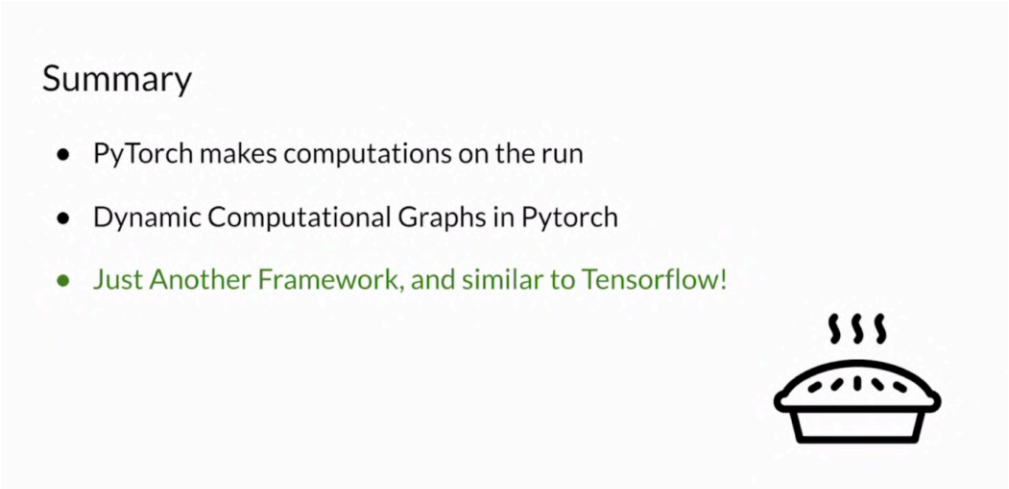
```
for t in range(n_epochs):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Training loop for number of epochs

Forward propagation

Optimization step



Works Cited

All of the resources cited in Course 1 Week 1, in one place. You are encouraged to explore these papers/sites if they interest you! They are listed in the order they appear in the lessons.

From the videos:

- Hyperspherical Variational Auto-Encoders (Davidson, Falorsi, De Cao, Kipf, and Tomczak, 2018): https://www.researchgate.net/figure/Latent-space-visualization-of-the-10-MNIST-digits-in-2-dimensions-of-both-N-VAE-left_fig2_324182043
- Analyzing and Improving the Image Quality of StyleGAN (Karras et al., 2020): https://arxiv.org/abs/1912.04958
- Semantic Image Synthesis with Spatially-Adaptive Normalization (Park, Liu, Wang, and Zhu, 2019): https://arxiv.org/abs/1903.07291
- Few-shot Adversarial Learning of Realistic Neural Talking Head Models (Zakharov, Shysheya, Burkov, and Lempitsky, 2019): https://arxiv.org/abs/1905.08233
- Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling (Wu, Zhang, Xue, Freeman, and Tenenbaum, 2017): https://arxiv.org/abs/1610.07584
- These Cats Do Not Exist (Glover and Mott, 2019): http://thesecatsdonotexist.com/

From the notebooks:

- Large Scale GAN Training for High Fidelity Natural Image Synthesis (Brock, Donahue, and Simonyan, 2019): https://arxiv.org/abs/1809.11096
- PyTorch Documentation: https://pytorch.org/docs/stable/index.html#pytorch-documentation
- MNIST Database: http://yann.lecun.com/exdb/mnist/