CS 182/282A Deep Neural Networks **Fall 2022**

Lecture 25: 11/17/22

Lecturer: Anant Sahai *Scribe: Liam Tan*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

25.1 Generative Tasks/Approaches

We will consider generation tasks, which come from the idea that if we can truly understand the underlying regularities of some data, the model should be able to generate a new unseen example of the data.

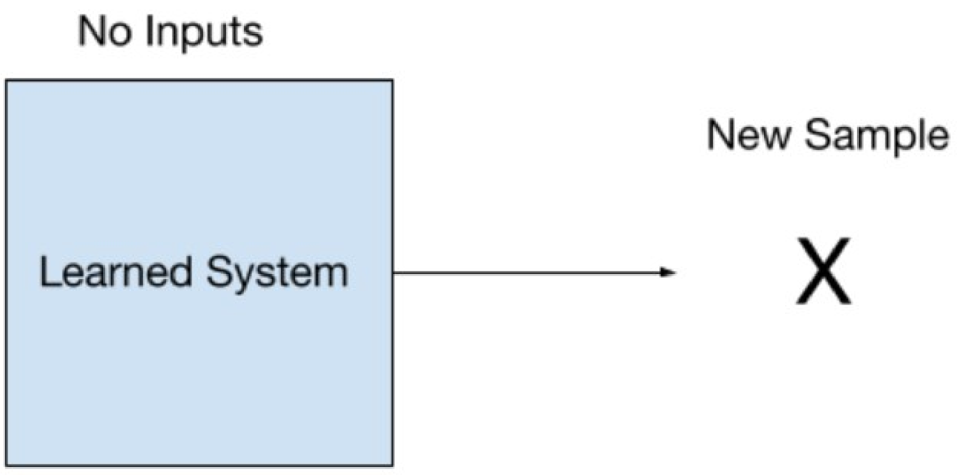


Figure 25.1: Basic Generation Model Framework

Take the system above. We have a learned generation system that takes in no inputs. Say it has learned to generate cats. When the model is called upon, we want it to generate a new cat. How can we ensure that a fresh image is generated each time, rather than a memorized image? We would like to have this system generate a different example each time. We do not want this to act as a deterministic circuit, one that cannot generate different things.

Thus, we will feed some sort of randomness to our system in order to ensure an identity map is not learned.

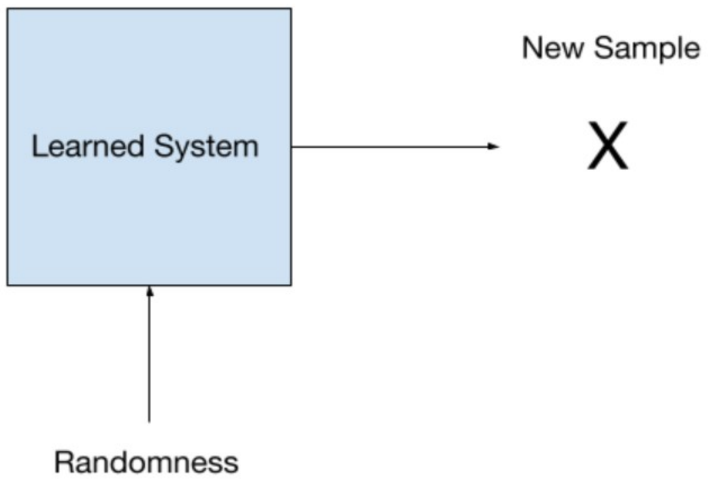


Figure 25.2: Generation Model Framework with Randomness

Given some randomness, we can generate new examples each time. Assume our learned system can generate fresh cat images each time it is run. What if we want to generate images of dogs, or images of planes?

Simple generation is not as useful as being able to control that generation. We turn to conditional generation.

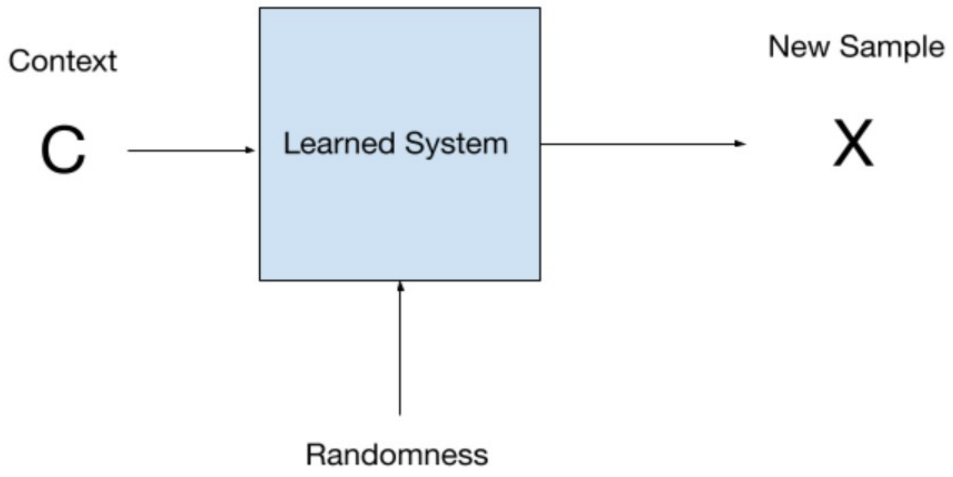


Figure 25.3: Conditional Generation Framework

Each time we run our generational model, we provide it some context (e.g tell the model to generate cats). Out comes a random sample that is correct in context. Say we tell it to generate an image of ice cream. It will create a new image of ice cream.

Examples:

We give it an image of an object. The generational model outputs images of randomly sampled angles of that object.

Context can be some text. For example, we can feed it a label of an image. If we tell it to generate "cats playing poker" it will do so.

Questions:

- Why do we care about this? Isn't this just some novelty or a party trick?
  → There is a grand tradition of engineering and that is putting people out of work when machines can perform the task at hand
  → People are hired to construct 3D models for movies and video games and this is quite a laborious task. If generational models can do this, these designers are no longer needed.

  Generational models can aid in tasks where there is large amounts of human labor needed to create something.
  → Stock image generation. Photographers spend countless hours taking photos of simple objects for stock photos to be used in advertisements and such.

  With the rise of advanced generational models, photographers no longer need to spend that time taking and editing photos for that purpose.

- How can this be used for machine learning itself?
  We can use GANs for data augmentation.

  → We can use generational models to "create" more training data. These generated examples are viewed as a kind of data augmentation.

  They have learned a lot of regularity that you can then use to feed into your model.
  → Suppose we have very sensitive data that we do not want the model to memorize. We can use generational models for this.

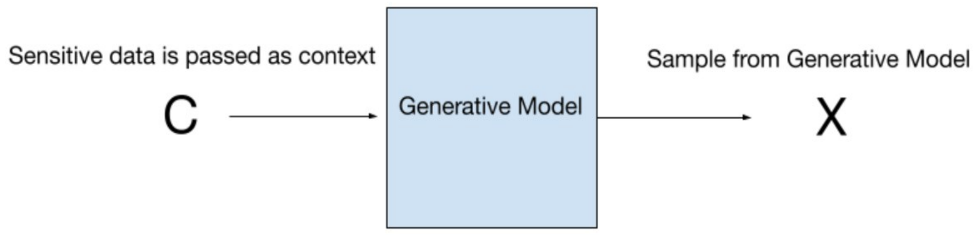

Sensitive data is passed as context
C
Generative Model
Sample from Generative Model
X

Figure 25.4: Data Sanitization/Privacy with Generational Models

We have a machine learning problem that requires us to learn from sensitive data. How can we make sure that our model does not memorize that data while still capturing the important underlying patterns within it? We can pass in our sensitive data into a generative model to create new data that can be passed into our machine learning model. If it is a good generative model and did not simply memorize things, we hope it does not output too much sensitive information. Thus we never have to give the sensitive data to someone that we may not trust.

25.2 Exploring and Designing Generational Models

25.2.1 Notes on Generation

Generation is a task unlike any previous task we've seen. It is very different than classification or regression. Currently, we've seen autocomplete for text generation and it can take a look at the previous word and try to guess the next one. More advanced autocompletes can take a look at all your words so far and try to predict the next words. Very advanced autcompletes can look at all the words so far and try to predict the next sentence. This is even being explored to see how these models can generate code. They take a look at what you are typing and suggest changes based on what it thinks you are trying to do. Given this rise in ability, researchers really want to understand generation tasks.

Generational Models is a phrase often used to capture the models that do the generation tasks. However, we must make the distinction between "model" and "task". Tasks refer to the different objective a system would like to achieve. This refers to regression, classification, and generation. There are a large scope of architectures that can do each task, so therefore when we refer to generation, we discuss the task, not the specific architecture that can do that task. For instance, transformers are useful in classification, as well as text generation(GPT). Generation is a task. The specific architectures that do these tasks are their own concepts.

25.2.2 Example with GPT

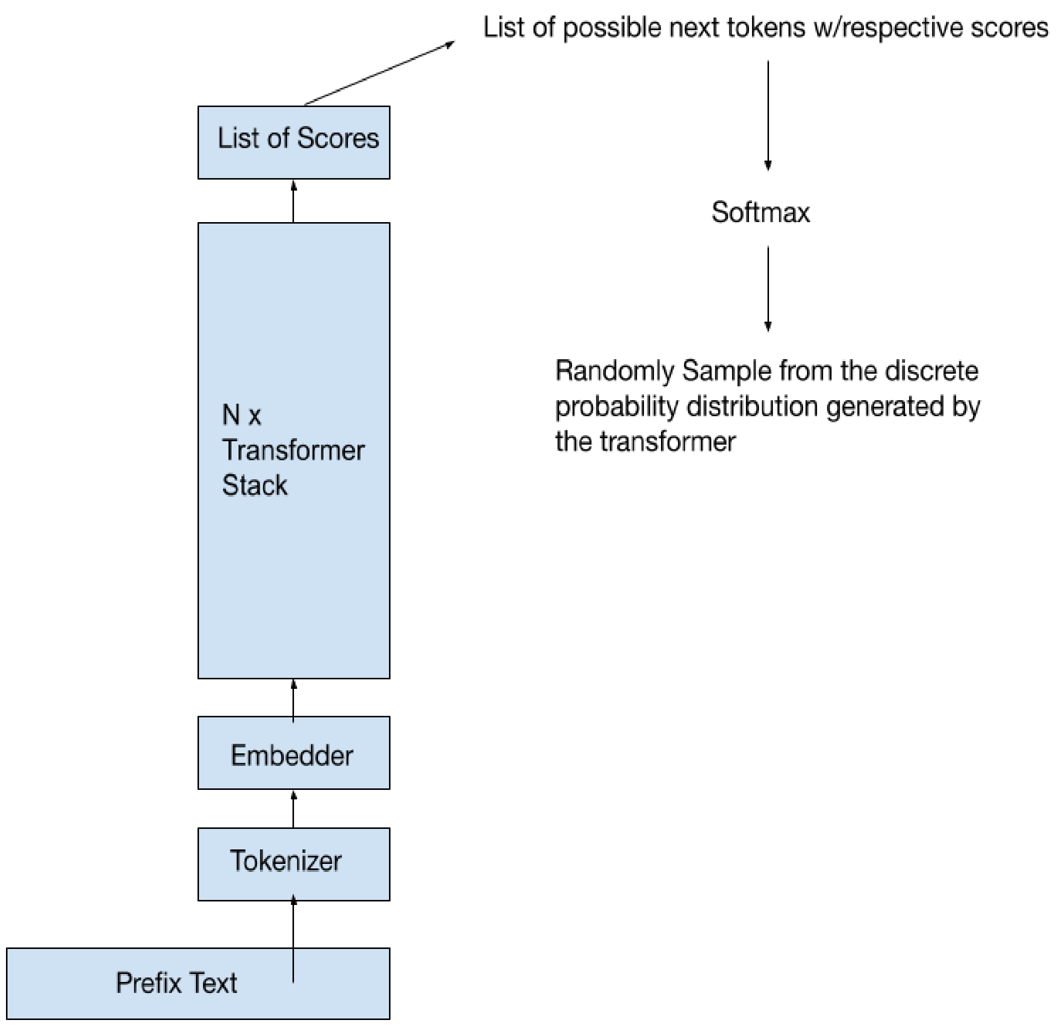

List of possible next tokens w/respective scores
List of Scores
Softmax
Randomly Sample from the discrete probability distribution generated by the transformer
N x Transformer Stack
Embedder
Tokenizer
Prefix Text

Figure 25.5: GPT for next token prediction

We use the GPT model for sentence generation. We provide a prefix text, which is the context in this scenario.

The prefix text will be accessed through the attention tables in each layer. As we pass input through the transformer stack, we observe a list of scores as output.

How can we use these scores to generate the next output?

We can take a softmax of all scores and according to this probability distribution, generate a sample for the next input. This sampling is the randomness in generation that allows us to create fresh examples each time.

Aside

- How do we actually sample from a discrete distribution?
  $\rightarrow$ A computer's random number generator will allow us to sample from $Unif[0,1]$, for instance. How can we use this to sample from a given discrete distribution?

Item	Probability	CDF
1	p_1	0
2	p_2	p_1
3	p_3	$p_1 + p_2$
4	p_4	$p_1 + p_2 + p_3$
...	...	...

Figure 25.6: Sampling from a discrete distribution

  $\rightarrow$ Let U be a random variable with distribution $Unif[0,1]$. Sample from U and look at what interval the realized value is in. Pick that item.

- How do we sample from a continuous distribution?

  $\rightarrow$ The main part we can take away from the discrete example is the CDF. Let's devise a strategy that utilizes the CDF.

  Let $F_X(x)$ be the CDF. $F_X(x) = P(X \leq x)$
  $P(X \leq x) = \int_{-\infty}^{x} p_X(x)dx$
  Sample U from $Unif[0,1]$. Solve $P(X \leq x) = u$ for x. $F(x) = u \implies x = F^{-1}(u)$
  Note that the CDF is monotonically increasing.

Aside Cont.

- How do we sample from a multidimensional $X \in R^d$ with density $f_X(x)$?
  $\rightarrow$ When we can do something for one dimension and want to try it for multiple dimensions, the first thing we must ask ourselves is "How can I turn this problem into a one dimensional problem?

  I know how to solve that, so if I can turn this into a one dimensional problem, I can solve this too". We can turn this into a repetition of one dimensional problems. We think, "maybe I can somehow factor the pdf into d different things".

  We can factor the pdf into a product of conditional probabilities. $f_X(\vec{x}) = f_{X_1}(\vec{x}[1]) * f_{X_2|X_1}(\vec{x}[2]|\vec{x}[1]) * f_{X_3|X_1,X_2}(\vec{x}[3]|\vec{x}[2], \vec{x}[1]) * ...$

  Start with $\vec{u}$ from i.i.d $Unif[0,1]$ d times. Use $\vec{u}[1]$ to get $\vec{x}[1]$ leveraging the marginal CDF $F_{X_1}(\vec{x}[1])$. Use $\vec{u}[2]$ and $\vec{x}[1]$ to find $\vec{x}[2]$, leveraging the conditional CDF $F_{X_2|X_1}(\vec{x}[2]|\vec{x}[1])$. Repeat to generate all d terms.

  The conditional generation, where we use the previous term to generate the next term is the Autoregressive generation used in GPT.

- How do we sample from any distribution when we are only provided a standard normal distribution?

  $\Phi(x)$ is the CDF of the Gaussian distribution.

  $\rightarrow$ Sample n from the standard normal $N \sim Normal(0,1)$. $\Phi(n) = u$. $\Phi(N) \sim Unif(0,1)$. This shows that if we have any continuous distribution, we can sample from any other continuous distribution. We take any arbitrary distribution and by applying the CDF, we can get a sample from a uniform distribution from 0 to 1. Now that we have a uniform distribution, we can sample any distribution we want with the methods from above. Any continuous function can simulate any discrete or mixed distribution.

  In general, X is a random variable with some arbitrary continuous distribution.

  Sample from X to realize the value x. Apply the CDF to x($F_X(x) = u$), and we now get a value u from 0 to 1. I claim that u is sampled uniformly from 0 to 1. Try to think about why this is true. Suppose I want to sample from a distribution with pdf $f_Y(y)$. Take $P(Y \leq y) = u$. Thus $F_Y(y) = u \implies y = F_Y^{-1}(u)$. y is the sample from our desired distribution.

- How can we use a discrete distribution to sample from a continuous distribution in spirit? We can also use more than just the discrete distribution.
  $\rightarrow$ We sample from the discrete distribution to decide which interval we are in. Then we use a uniform distribution to decide where in the interval we are.

- What are the interesting properties of the normal distribution?
  $\rightarrow$ If we add up many distributions, in limit the distribution becomes normal. This is the central limit theorem. In addition, adding up two independent gaussian distributions results in another gaussian distribution.

Say we want to sample images the same way GPT does. We want to use an autoregressive approach to mask out all future parts of the image and generate the image that way. Like GPT, we must do this in a one-by-one approach. This can either be one pixel at a time or one patch at a time.

The two main things we need in an autoregressive generation is randomness and an ordering of conditioning. The randomness comes from sampling our discrete distributions and the number of random variables is the dimensionality of samples to generate. The ordering is used to compute the marginal distributions/CDF.

What is the main problem? With GPT next word prediction, we have some sort of ordering in which the words were passed into the transformer model, but with images, it is unclear as to how order the pixels/patches.

In comes raster scan ordering.

1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	...	...	...	...	...	...	...	X

Figure 25.7: Raster Scan Ordering for Pixel Prediction

The pixels are ordered left to right top to bottom in raster scan ordering. Now we have an ordering and can train like GPT with a transformer model. This tends to work ok and is not bad. There are ways to train this with transformer models and even convnets. We can also do a similar approach patch by patch.

This raster scan approach is quite unintuitive because it does not capture the true topology of an image. Images are not exactly sequence data. This approach is not an intuitive method that comes from an idea of how images are structured. However, we use this because it tends to work somewhat.

Convnets and in turn weight sharing can also be utilized here but we must be careful about how to define the topology. What is near and what is far? In a convnet, the pixel right under the current pixel is classified as near. In raster scan, pixel 1 and 10 in figure 25.7 are 9 pixels away, which is quite far. Thus we need to include some sort of 2 dimensional positional encoding to encapsulate the properties of images.

With Convnets, we must think about how to bake in the inductive bias in the structure of the convolution. We have gradients and we need to mask out the gradients coming from the future data.

This method of masking and generating is not terrible, and there is some nice conditioning with the generation. The transformer allows context text with cross attention.

Images are large, and can often be millions of pixels. With the quadratic time complexity of attention, this is a problem. This style of generation is completely sequential and cannot be parallelized. Even with patching this method is considered to be quite slow.

25.2.3 Two Naive Approaches

25.2.3.1 Naive Approach #1

Due to the transformer approach being quite compute heavy, we look towards solutions that are faster.

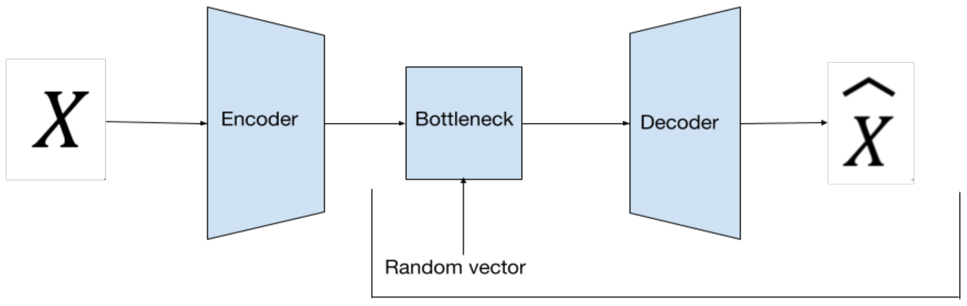


Figure 25.8: Utilizing an Autoencoder for Image Generation

We attempt to only use an autoencoder for image generation. We first train this autoencoder as a standard one. If it is a good autencoder, X and $\widehat{X}$ should be pretty similar.

We then take only the decoder side and feed in a random vector, hoping that it will generate a fresh image. This approach has a very fast runtime. However, the decoder outputs garbage. It turns out that there is still some structure in the bottleneck and that not all vectors in the bottleneck correspond to a real image when passed through the decoder.

Thus, we need to try to learn this structure, and therefore pass in gradients that guide us to learn that structure.

25.2.3.2 Naive Approach #2

Let's try using a classifier to help generate an image.

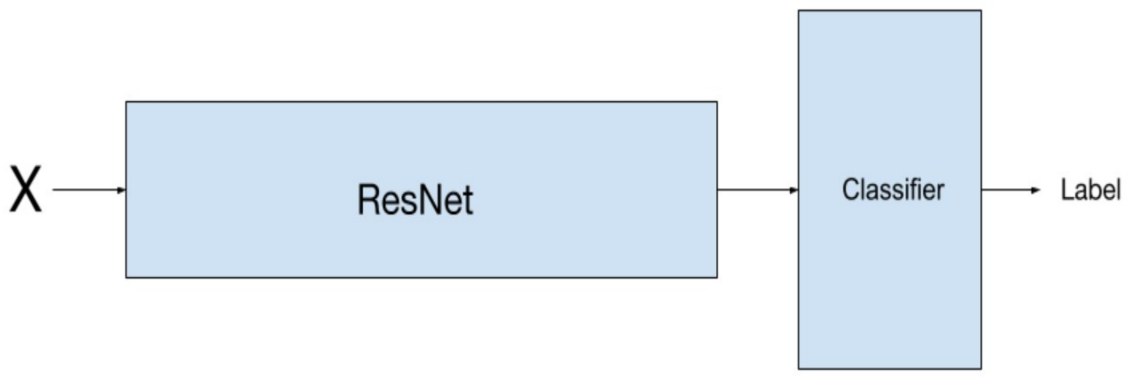


Figure 25.9: Utilizing Classifier for Image Generation

We start with random noise and pass that in. In each iteration, we want to make our image more catlike. Thus, we will do gradient ascent to make our score for the classification of a cat as large as possible

$X_t = X_{t-1} + \eta(\Delta Score_{cat}(X_{t-1}))$.

Thus we try to maximize the cat score and therefore we try to make our image look more like a cat each iteration. Every iteration we change our input image to make it have a higher cat score.

Suprisingly, our image output at the end of training looks like noise. We think, "Hrm, maybe we've ended up at a local minima and got unlucky". However, when we take a look at the classification scores/softmax scores, we see that the image is confidently classified as a cat! What happened here? "Hrm, perhaps it is because we started with random noise, which is so far from a real image that we could never possibly start with that image and get a real life image". Thus, we decide to pass in real images, say a picture of a table. We think, "maybe if we start with a real image like a table, the classifier can sort of work with the structure and go from there. We expect some modifications to the image and maybe a cat will appear on the table".

What was even more surprising was that after training this, the resultant X_{opt} looked like X and the scores said that this was a cat now! The image was somehow able to fool the classifier. This is called an adversarial example.

25.2.3.3 Generative Adversarial Networks

What if we combined these two approaches to train a classifier and a generator together during training?

Our generator would generate images, and our classifier would detect if the image was generated or if it was real. The classifier is now called the discriminator, because its job is to discriminate between real and fake images. The generator's job is to try to fool the discriminator into believing that the generated image is real. The loss will backpropagate and help train both parts of the model at the same time.

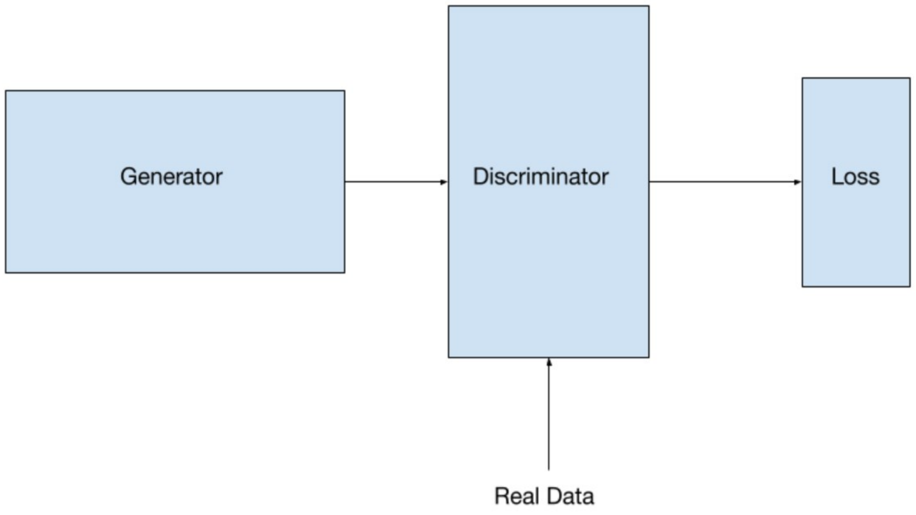
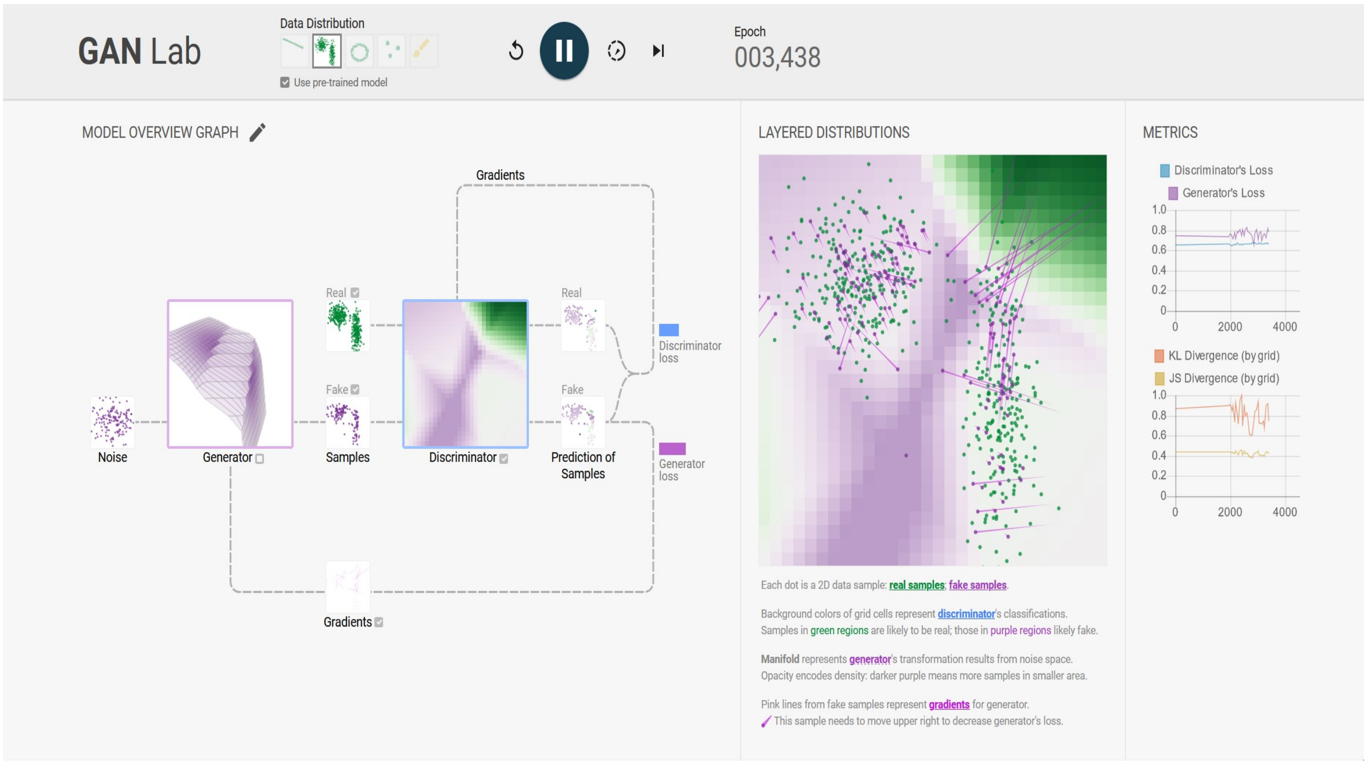


Figure 25.10: Basic GAN model

GANs are notoriously hard to train because they are so finicky and delicate. There has to be just the right alignment and balance for them to train well.

A common issue is called mode collapse. This is when a generator finds a decent output and continues to just produce that output. The discriminator soon learns to always reject that output, even if it could be real. A cat and mouse scenario happens and the generator is limited to producing a small subset of possible outputs. The next lecture goes into more detail about this.

See here for a visualization of GAN training: GAN Simulation



Each dot is a 2D data sample: real samples; fake samples.

Background colors of grid cells represent discriminator's classifications.
Samples in green regions are likely to be real; those in purple regions likely fake.

Manifold represents generator's transformation results from noise space.
Opacity encodes density: darker purple means more samples in smaller area.

Pink lines from fake samples represent gradients for generator.
This sample needs to move upper right to decrease generator's loss.

Figure 25.11: GAN Simulation

Live GAN training and even mode collapse can be observed.

25.3 References

References

[1] Minsuk Kahng, Nikhil Thorat, Polo Chau, Fernanda Viégas, Martin Wattenber. GAN Lab, “Play with Generative Adversarial Networks (GANs) in your browser!” In: (2019) DOI https://poloclub.github.io/ganlab/