Week 1: Intro to GANs

Generative Models:

* What are generative models?
* Types of generative models

GANs included!

Generative Models vs. Discriminative Models

Discriminative Models

Used for classification in ML. They learn how to distinguish between classes such as dogs and cats. Often called classifiers. Takes a set of features X, such as having a wet nose, purrs.

Features → Class

$X \rightarrow Y$

They try to model the probability,

$$P(Y \mid X)$$

Class Y, given a set of features.

Generative Models

Tries to learn how to make a realistic representation of some class. They take some random input represented by the noise here, ε

can take a random number 3, -5, 2.6... the noise represents a random set of values going into the generative model. The generative models sometimes takes in a class Y such as a dog.

Generative Models:

It's goal is to generate a set of features x that look like a realistic dog.

Noise ε, Class y → Features x

Why we need this noise?

- Generative Models try the capture the probability distribution of x.
  The noise is to ensure that what's generated isn't actually the same dog each time.

$P(x \mid y)$ → Realistic and diverse representation.

features | dog

Types of Generative Models

★ Variational Autoencoders:

Works with two model "encoder" and "decoder". These are typically a neural network.

Encoders → learns by feeding realistic images into the encoder. Encoder's job is to find a good way of representing that image in latent space. And the suitable point in the latent space can be represented by this vector of numbers 6.2, −3, 21.

Decoder → We put this vector to decoder and is goal is to reconstruct the realistic image that the encoder saw before.

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

The part we covered is the autoencoder part. Variational part add 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 encode the image into 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 → produce realistic image

Generator (Decoder)

$\begin{bmatrix} 1.2 \\ 3 \\ -5 \end{bmatrix}$

Discriminator

↓

Real or fake?

Generative models try to mimic the distribution of the data you're using to train them on.

GANs

Generators

* Generates fake that look real

both real and fake images in a pile

Discriminators

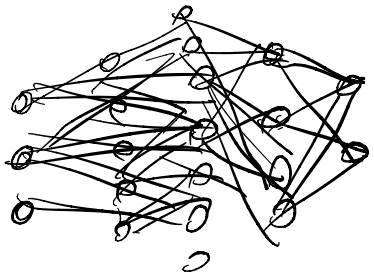
* learns to distinguish real from fake

After prediction tell it whether it was real or fake

each of these ↑ nodes

(Θ) Parameters

update parameters



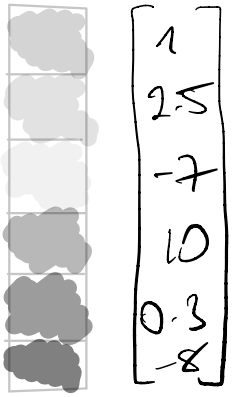
X → Discriminator → Output $\hat{Y}$ → Cost func.

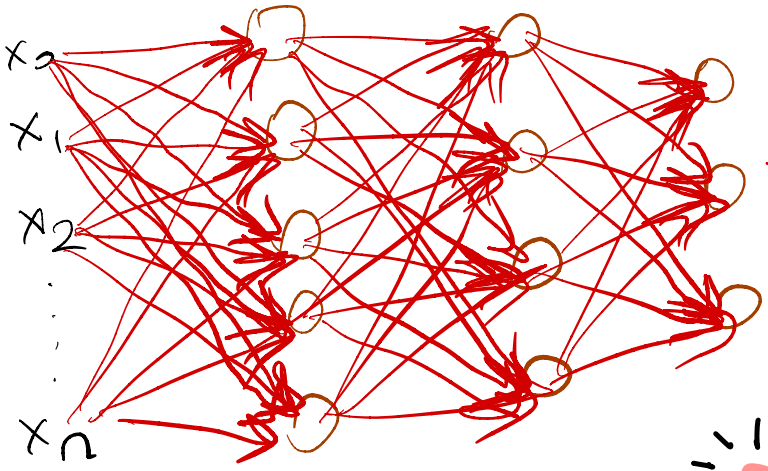
Y

* Discriminator is a classifier
* Learn the probability of class Y given features X (P(Y|X))
* The probabilities are the feedback of generator

Generator

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.





cat image

Noise vector (ε) + class y

↳ sometimes if we have more than one class

* It learns the 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.

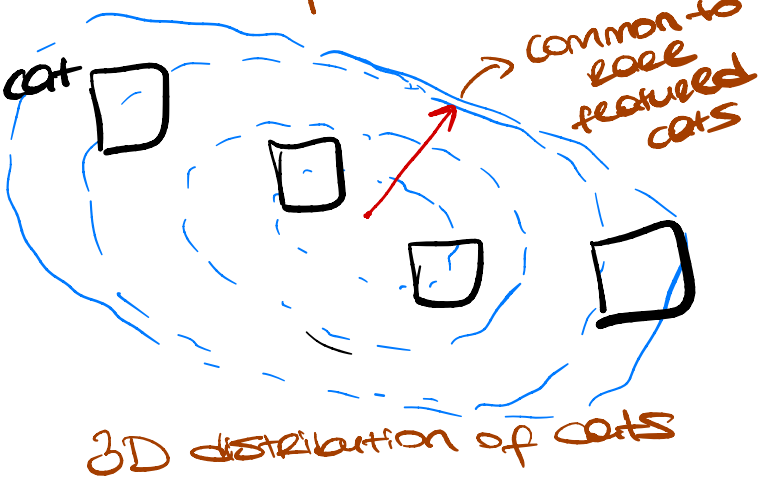
Probability distributions: $P(X|Y)$

feature ↖ ↗ class

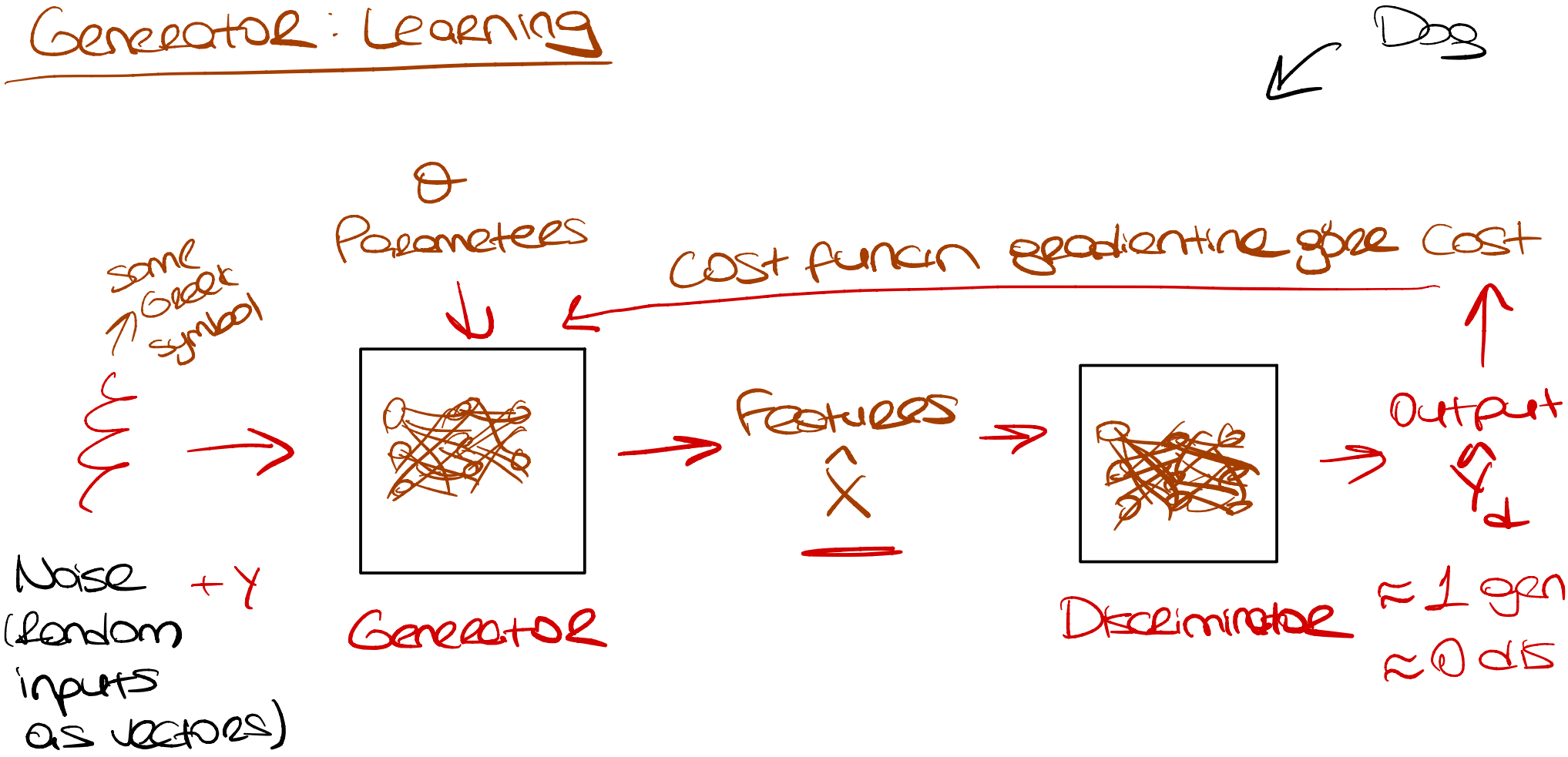
↳ conditional probability

$P(x)$ → we don't need label bc we now it's cat

All the features of a cat in the world



Generator: Learning



→ If it's good enough, you freeze the parameters of generator and you can generate images from new random noises.

BCE Cost Function (Binary Cross Entropy)

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

- ① the first term, $y^{(i)}\log h(x^{(i)},\theta)$; ② the second term, $(1-y^{(i)})\log(1-h(x^{(i)},\theta))$
- $-\frac{1}{m}\sum_{i=1}^{m}$ → Average loss of the whole batch
- $h(x^{(i)},\theta)$ → Prediction (0, 1)
- $y^{(i)}$ → true label (0, 1)
- $x^{(i)}$ → features
- θ → parameters

* $h(x^{(i)};\theta)$ ↳ parameterized by θ

① $y^{(i)}\log h(x^{(i)},\theta)$ → the product of true label y, times the log of the prediction.

	$y^{(i)}$	$h(x^{(i)},\theta)$	$y^{(i)}\log h(x^{(i)},\theta)$
false	0	any	0
true	1	0.99	~0 (close to 0)
	1	~0	$-\infty$

↓ Relevant when the label is 1.

↓ bc of the log

* It makes it 0 if your prediction is good. And it makes it negative infinite when it's bad.

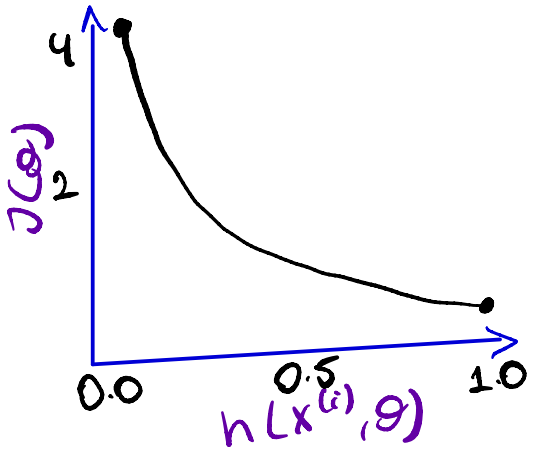
②

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

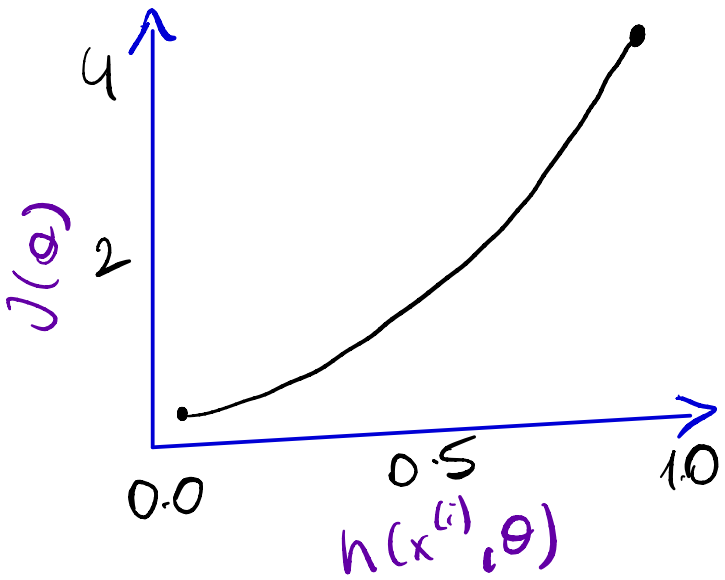
→ Relevant when the label is 0.

* Logarithm of a value between 1-0 is calculated, which returns that negative result. That's why we want that negative term.

$y^{(i)} \log h(x^{(i)}, \theta)$ → relevant when the label 1.



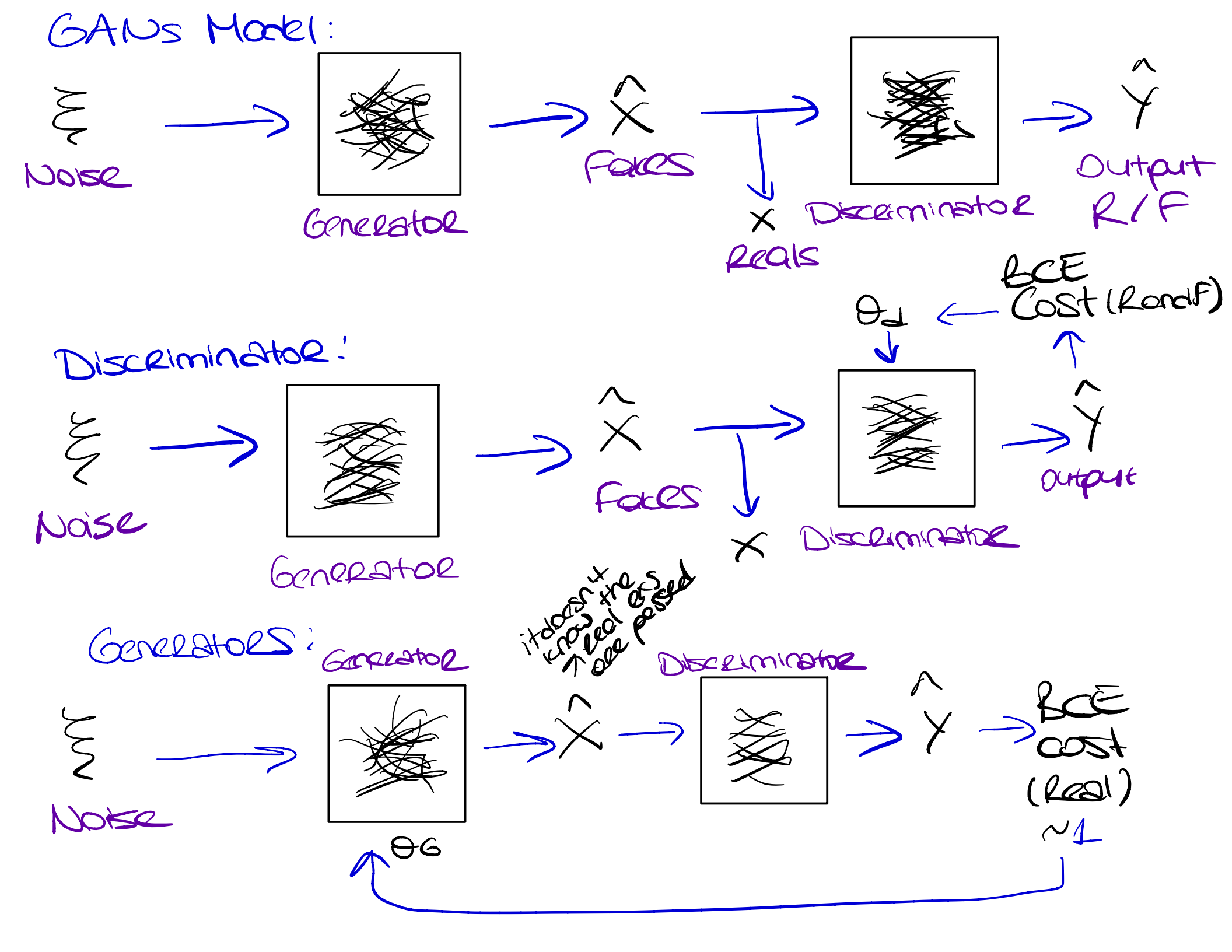
$(1-y^{(i)}) \log (1-h(x^{(i)}, \theta))$ → relevant label 0.



SUMMARY

* The BCE cost function has two parts that 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.

→ The BCE loss is performed across a mini batch of several examples and it takes the average of all those n examples.


GANs Model:
Noise
Generator
Fakes
Reals
Discriminator
Output
R / F
Discriminator:
Noise
Generator
Fakes
Discriminator
output
BCE
Cost (Rand F)
Generators:
Noise
Generator
θG
it doesn't know the real ones passed
Discriminator
BCE
cost
(Real)
~1

$W_k^{[j]}$ → column vectors

↳ neuron number

↳ when the input of a tanh is far from zero, it's gradient is very close to zero, because of the flat edges of tanh.)

- tanh always works better than sigmoid when used in hidden layers.
  - ↳ exception is for the output layer in classification.
- Logistic regression doesn't have a hidden layer. If you initialize weights to zero, the first example x fed into the LR will output zero but the derivatives of the logistic regression depends on the input x, which is not zero. So, at the second iteration, the weights' values follows x's distribution and are different from each other if x is not a constant vector.

$W^{[L]}$ → row is the number of neurons of k^{th}

→ column is the " " input of k^{th}.

- Output of the tanh is between -1 and 1. It thus centers the data which makes the learning simpler for the next layer.

hidden layer with tanh

weight initialization

- The use of random numbers helps to "break the symmetry" between all the neurons allowing them to compute different functions. When using small random numbers the values $z^{[L]}$ will be close to zero thus the activation values will have a large gradient speeding up the training process.

Week 2

Activations: are functions that take any real number as input also known as it's domain and outputs a number in a certain range. using a non-linear differentiable function

Why non-linear and differentiable?

1) Differentiable for backpropogation because you use backpropogation for training and updating its parameters, so it needs to be able to differentiate and provide gradient to previous layer.

2) It needs to be non-linear, the features you compute within the neural networks can be complex.

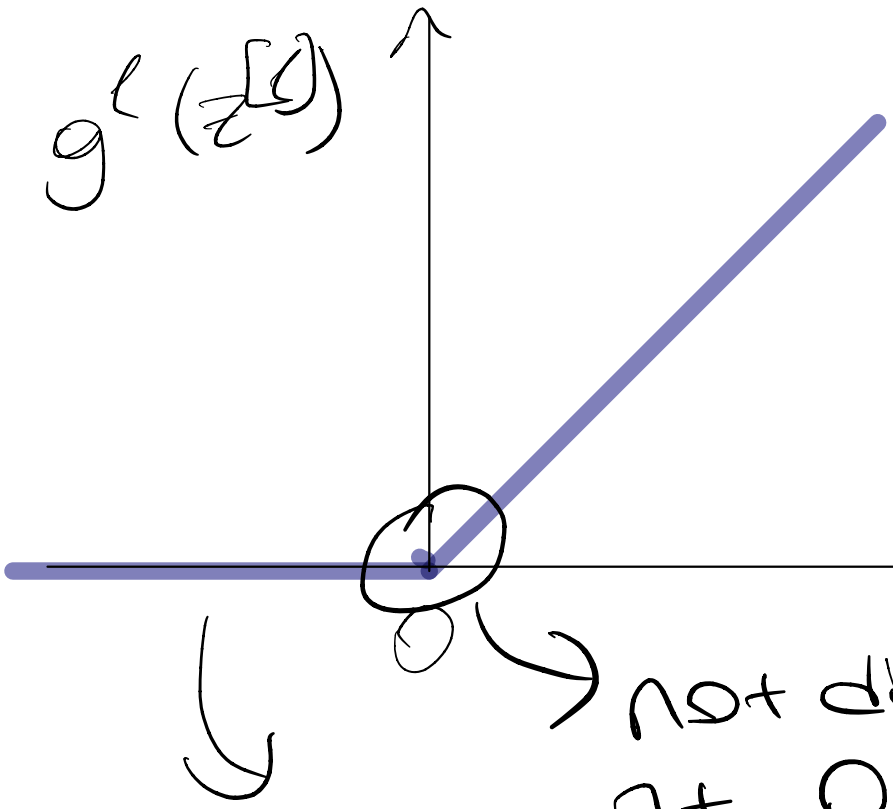
↳ So you need non-linear differentiable activation functions to take advantage of deep learning models and to have the layers stack onto each other to construct a complex neural network to model complex nonlinear functions.

* Non-linearity ensures that your linear layers dont reduce to a singe layer

Common Activation Functions

ReLU → Rectified Linear Unit

$= \max(0, z^{[l]})$



→ no-negative values are allowed.

→ not differentiable at 0

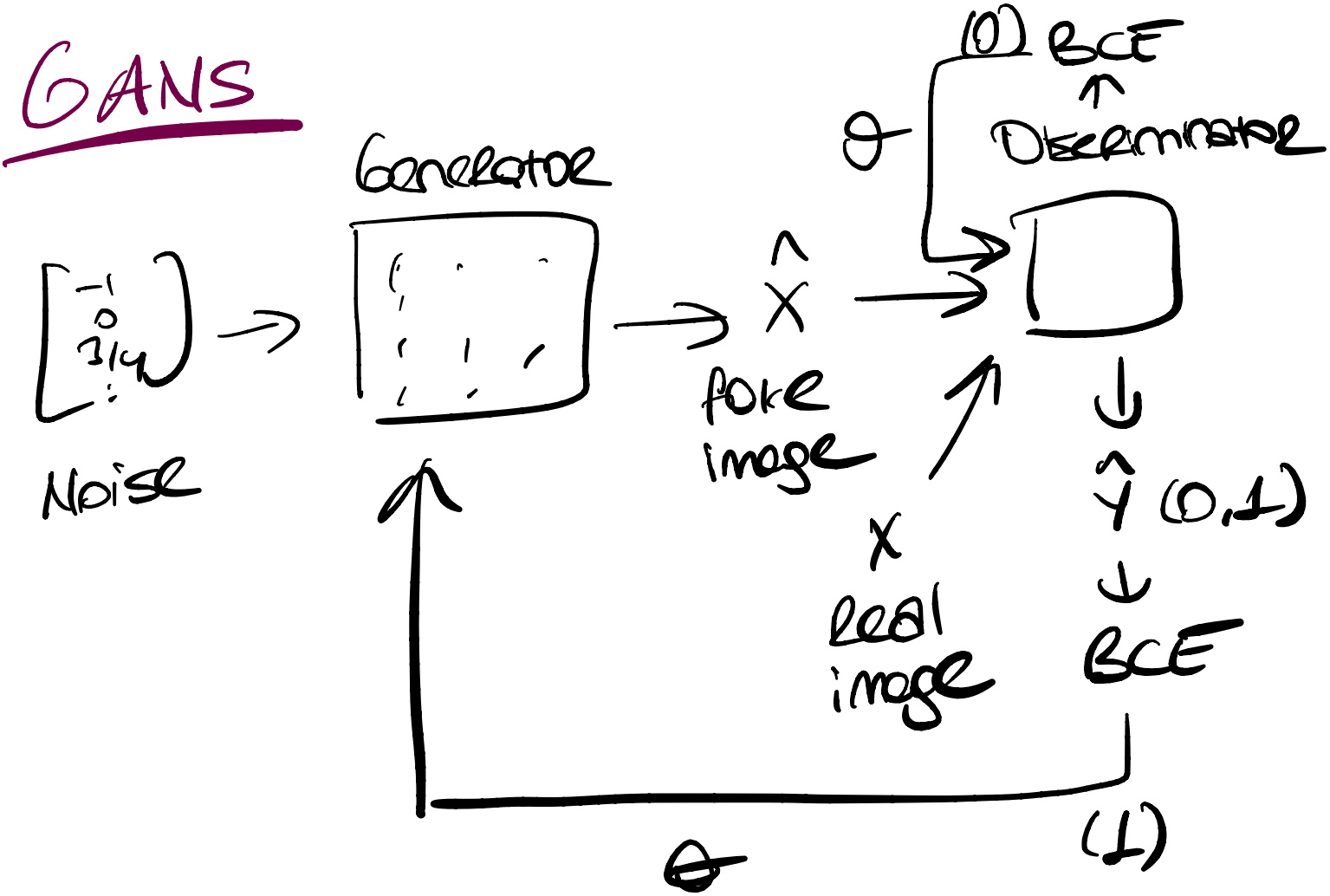
this part
z is negative
always has a derivative
equal to zero!

Since the learning process depends on the derivative to provide important information on how to update the weight with 0 some values and their weights

stop learning. All the components will be affected as well.

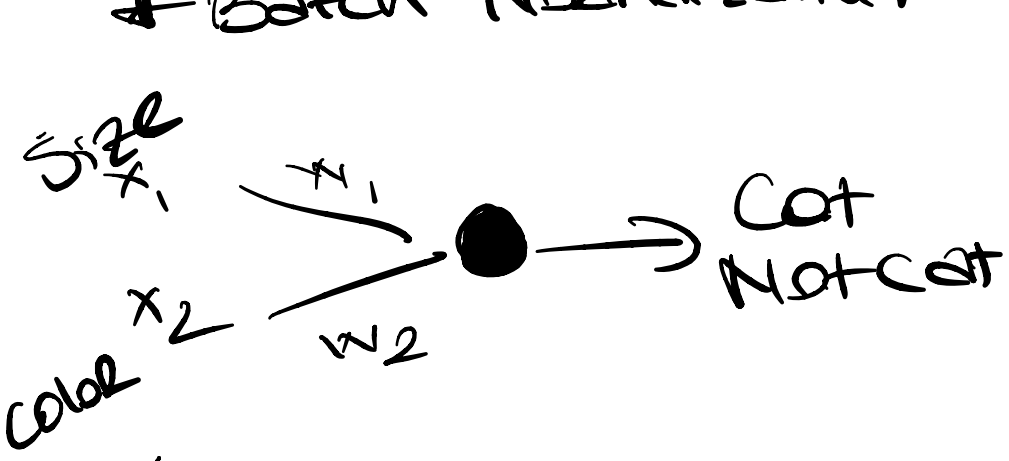
This is dying ReLu problem.

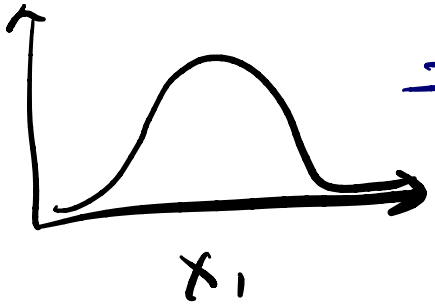
GANS



Batch Normalization

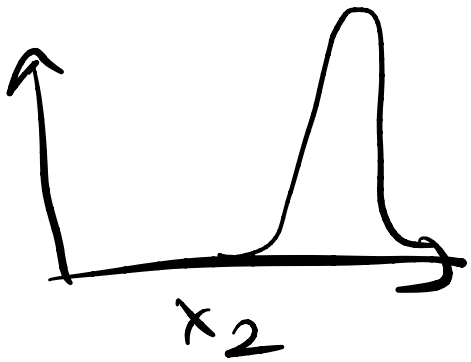
* Normalization
* Internal Covariate Shift
* Batch Normalization





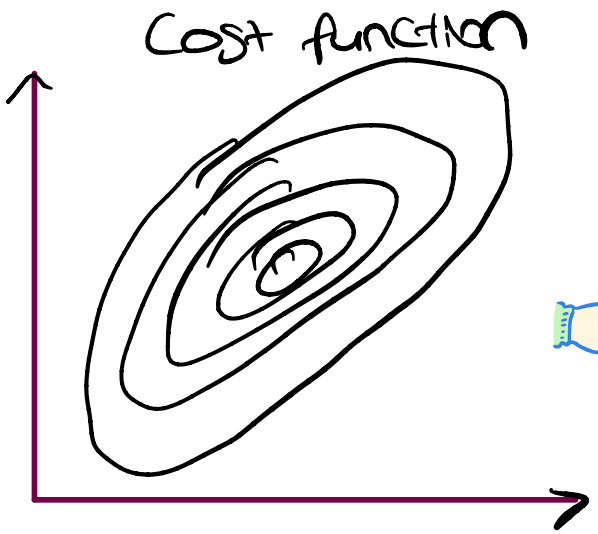
→ normally distributed

Skews a little bit towards higher values



→ higher mean lower std

Comparing the same value on two features is not possible

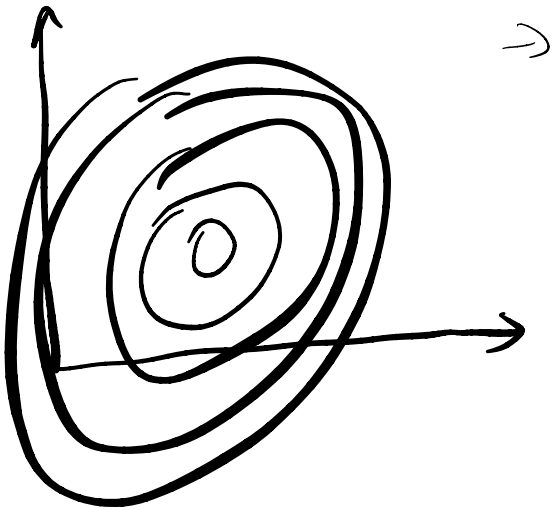


* Different distributions across inputs, cost function will be elongated.

So the changes to the weights relating to each of the inputs will have kind of a different effect of varying impact on this cost functions.

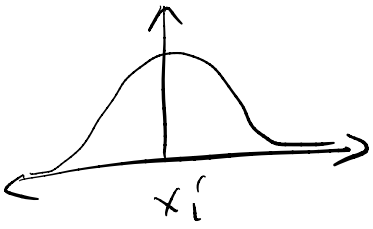
* This makes training difficult and slow. And highly dependant on how your weights are initialized.

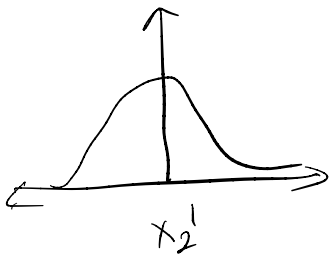
If train or test data has different distributions, kinds of shifts or changes in some way then the form of cost function could change too.



→ Little bit more round and location of the minimum could also move. This known as "covariance shift".

What if the input variables x_1 and x_2 are normalized?

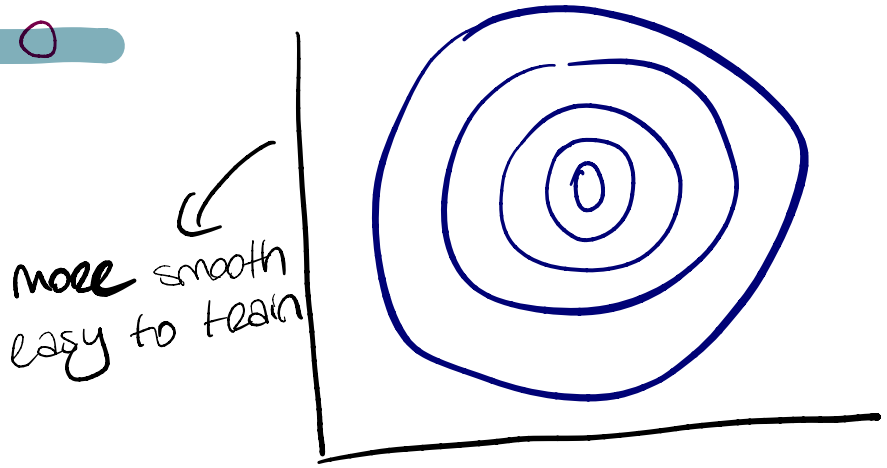




Cost function will also look more balanced and smoother

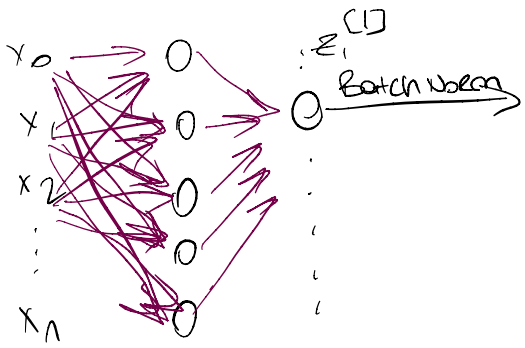
Normalized in same way, around of 0 and a standard deviation of 1.

- For training data, you train a batch. You take the mean and standard deviation and shift it to be around mean 0 and Sd of 1.
- For the test data, look at the statistics that we're gathered overtime as you went through the training set and use those to center the test data to be closer to the training data.



→ Smoothing the cost function and reducing the covariance shift.

* Covariant shift shouldn't be a problem if you just make sure that the distribution of your data set is similar to the task you're modelling.



All the weights that effect this activation value are updated. So all of the weights are updated. The distribution of values contained in that activation changes.

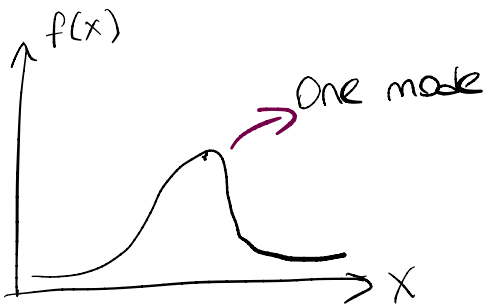
→ This makes training process difficult due to the shifts.

↓ B.N normalizes all these internal nodes based on statistics calculated for each input batch.

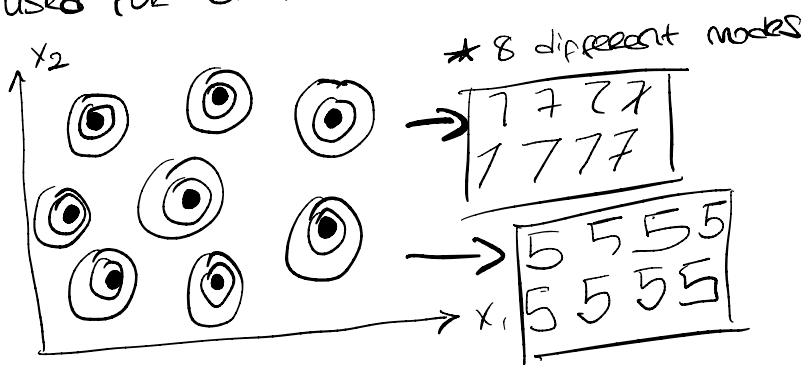
→ Internal Covariance Shift

Mode Collapse:

* Any peak on the density function is a mode.



Many real life distributions used for GANs are multimodal.

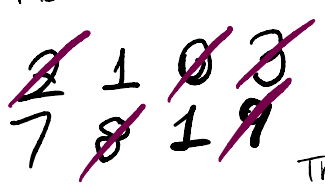


↳ Multimodal

* Sounds like things collapsing on one mode or fewer modes and the modes disappearing.

Discriminator

Good at identifying fake images but generated images that looks like 1 and 7.



Generated values

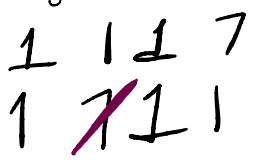
This means the the discriminator is at of local minima of cost function.

Then this information is passed on to the generator.

Generator

The generator sees this and looks at the feedback from the discriminator and gets a good idea of how to fool the discriminator in the next round.

It generates 1 and 7's.



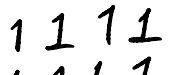
Passes to the discriminator

Misclassify every picture except for one maybe.

Passes to the generator

Sees that discriminator's weakness is with 1.

Produce images

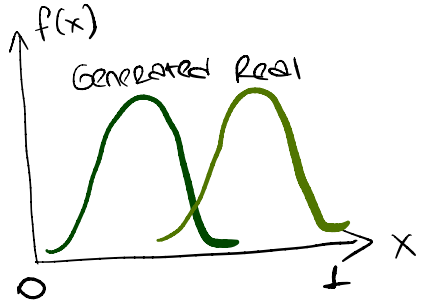


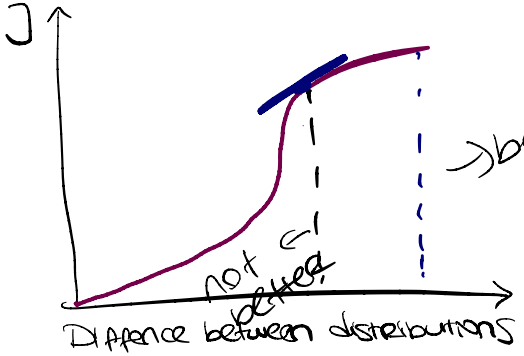
Collapsing to a single mode.

Problem with BCE Loss

* Discriminator is easier to train. → Often the discriminator gets better than the generator.
* Generator is difficult to train.
  ↳ Complex output

But beginning in training this isn't such a problem. Because discriminator is not that good.

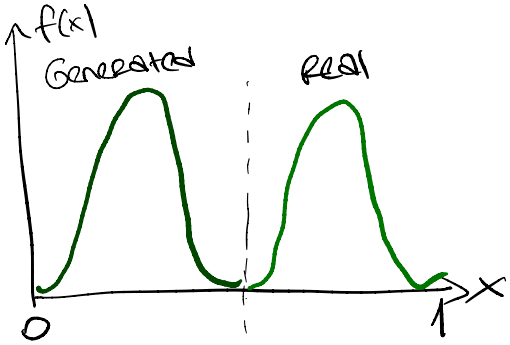




* It has problem distinguishing generated from real. As a result, it's able to give usefull feedback in the form of a non-zero gradient back to the generator.

→ As discriminator getting better it'll start giving less informative feedback. It might give gradients close to zero and becomes unhelpful to the generator. Because the generator doesn't know how to improve.

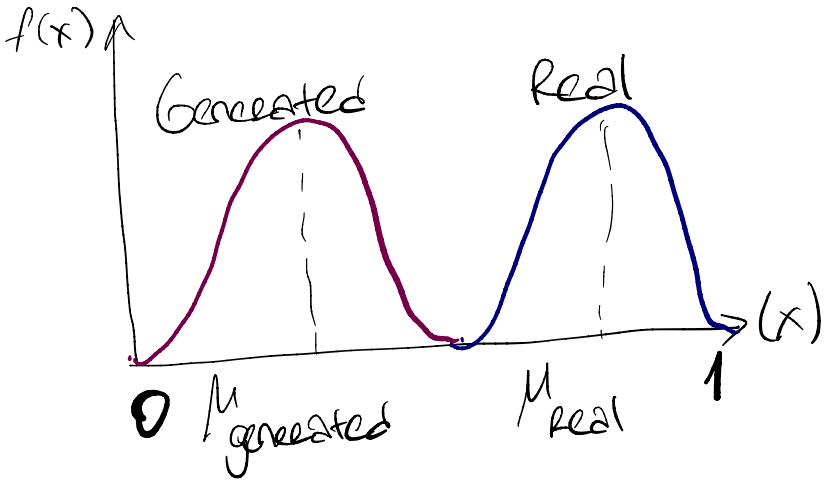
→ How the vanishing gradient problem will arise.



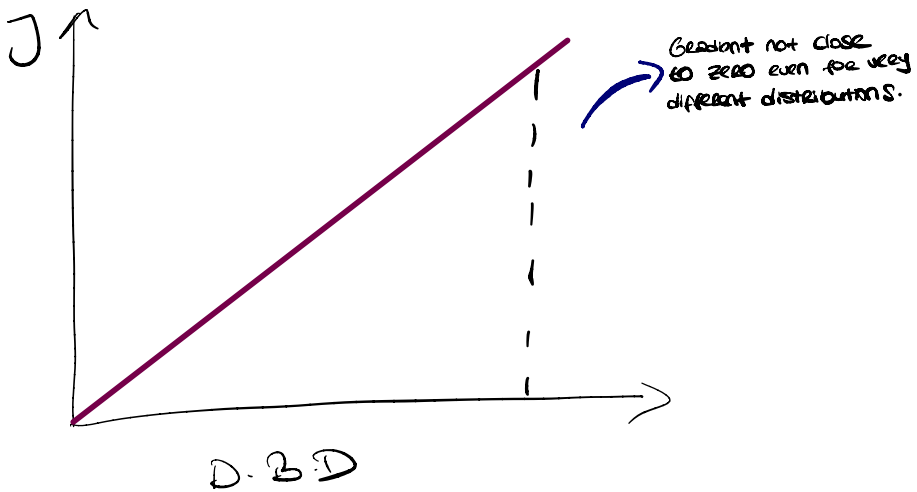
* However, as it's get better at training, it starts to delineate the generated and real distributions a little bit more.

* Discriminator improves during training and sometimes improves more easily than the generator, that underlying cost function will have those flat regions when the distributions are very different from one another

Earth Mover's Distance (EMD)



The function depends on the distance and amount moved.



* What EMD does is it measures how different these two distributions are, by estimating the amount of effort it takes to make the generated distributions equal to the real.

* With EMD there's no such a ceiling to the zero and one. So cost function continues to grow regardless how far apart these distributions are.

* Doesn't have flat regions when the distributions are very different.

* EMD solves the problem associated with BCE.

Wasserstein Loss

- W-Loss approximates the Earth Moon's distance.
- Generator's trying the maximazing the cost, discriminator's trying to minimize the cost.
  - This is often called minimax game!

$$\min_{d} \max_{g} -\left[\mathbb{E}(\log(d(x))) + \mathbb{E}(1-\log[d(g(z))])\right]$$

→ Simplified BCE loss

- $\log(d(x))$: How bad disc classifies real observations.
- $\log[d(g(z))]$: How bad disc classifies fake observations.

W-Loss: the function calculates the difference between the expected values of the predictions of the discriminator. Called "critic".

$$\min_{g} \max_{c} \mathbb{E}(c(x)) - \mathbb{E}(c(g(z)))$$

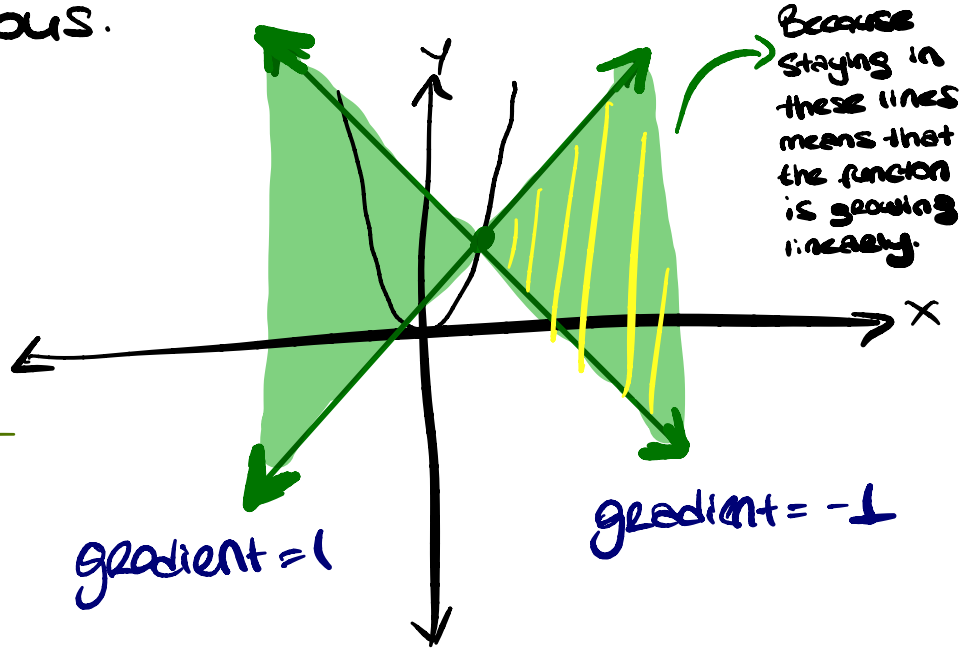
- $\mathbb{E}(c(x))$: Critics outputs are no longer bounded to be between 0 and 1.
- $\mathbb{E}(c(g(z)))$: Generator wants to minimize this bc it want the discriminator to think that its fake images are as close as possible to the reals.
- → Discriminator wants to maximize the distance between its thoughts on the real vs fake.

1-Lipschitz Continuous (1-L continuous)

Critic needs to be 1-L Continuous.

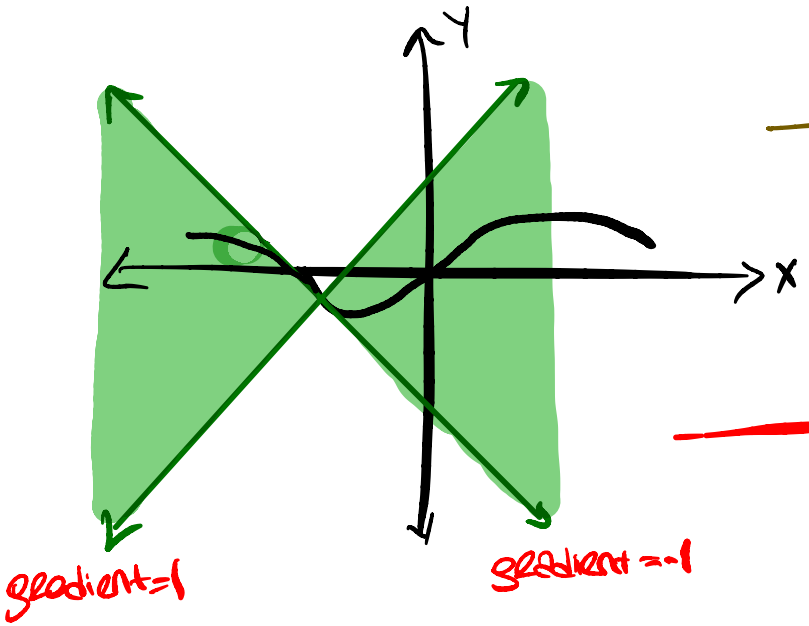
↓

The norm of it's gradient needs to be at most **one** for every point.



You want to make sure that growth of this function never goes out of bounds from these lines.

Not 1-L Continuous



→ You do it for every point and make sure that it grows linearly.

→ 1-L Continous

* This condition on the critics is important for W-Loss bc it assures that the W-loss is not only continuous and differentiable but also that it doesn't grow too much.

1-L Enforcement on Critic

1-L Continuous → $\|\nabla f(x)\|_2 \leq 1$ → Norm of gradient at most

($\|\cdot\|_2$: L2 norm; ∇: gradient, f: critic, x: image)

Weight Clipping

Forces the weights of the critic to a fixed interval. After your update the weights during gradient descent, you'll clip any weight outside of the desired interval. The weights over that interval either too high or too low, will be set to the maximum or the minimum amount allowed.

↓

It has downside!

Forcing the weights of the critic to a limited range of values could limit the critics ability to learn and ultimately for the gradient to perform because if the critic can't take on many different parameter values, it's weights can't take on many different values.

Gradient Penalty

$$\min_G \max_C \mathbb{E}(C(x)) - \mathbb{E}(C(G(z))) + \lambda \text{reg}$$

(reg: regularization)

Its penalizes the critic when it's gradient norm is higher than one.

Checking the gradients at every possible point of the feature space, not practicle.

Sample some points by interpolating between real and fake examples.

$(\|\nabla C(\hat{x})\|_2 - 1)^2$ → regularization term

∇C: gradient; $\hat{x}$: intermediate image

$\hat{x} = \epsilon x + (1-\epsilon) G(z)$ — Interpolation

x: Real; $G(z)$: generated

$$\min_G \max_C \mathbb{E}(C(x)) - \mathbb{E}(C(G(z))) + \lambda \mathbb{E}(\|\nabla C(\hat{x})\|_2 - 1)^2$$

Critic's gradient to be less than or equal to one.