# Generative & Adversarial Neural Networks

Robert Currie

# Artificial Neural Networks so far (a Recap)

- A **DNN** is a network of Neurons which are densely interconnected.
- We have used a **DNN** to build a generic classifier and train it on data.
- We have used a **CNN** to build a better classifier which can extract correlations within each datapoint.
- We have used **DNN** and **CNN** to build **A**uto-**E**ncoders to smooth input data and/or perform anomaly detection.

# AE, VAE & Generation

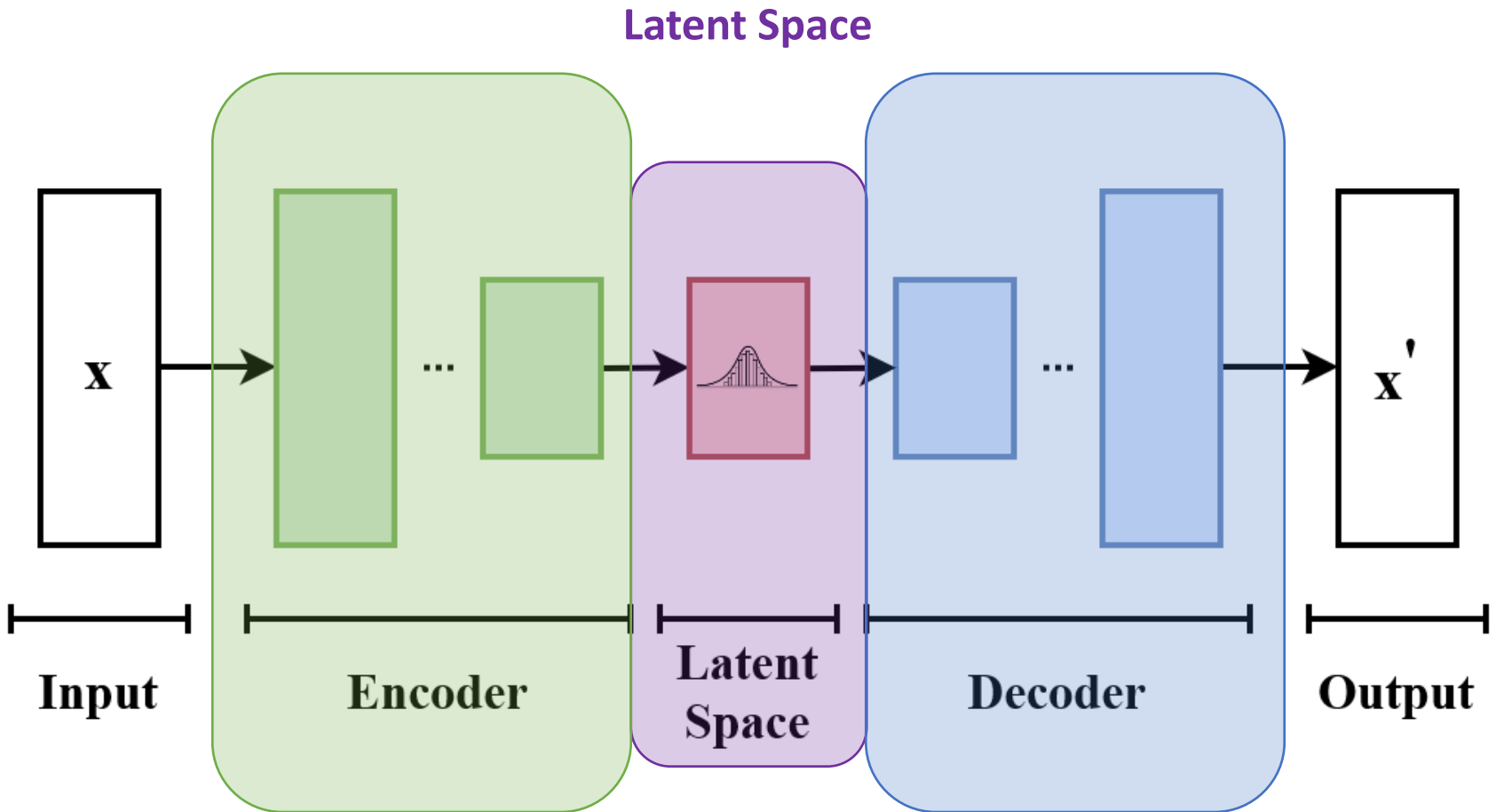
- **CNN**s are a good way of handling image and dense data
  Train/Generate large amount of data with less parameters than **DNN**

- **A**uto**E**ncoder (**AE**) models are good at learning features and generating unique output which appear '*the same*' as the input.

- **AE** can be used for:
  - **Image de-noising**
    (generating idealised input based on training data)
  - **Image segmentation**
    (generating image-masks/object-masks for object identification)
  - **Image classification**
    (identifying features in images and data)
  - **Image generation**
    (generating of 'new', 'unique' data based on the training set)

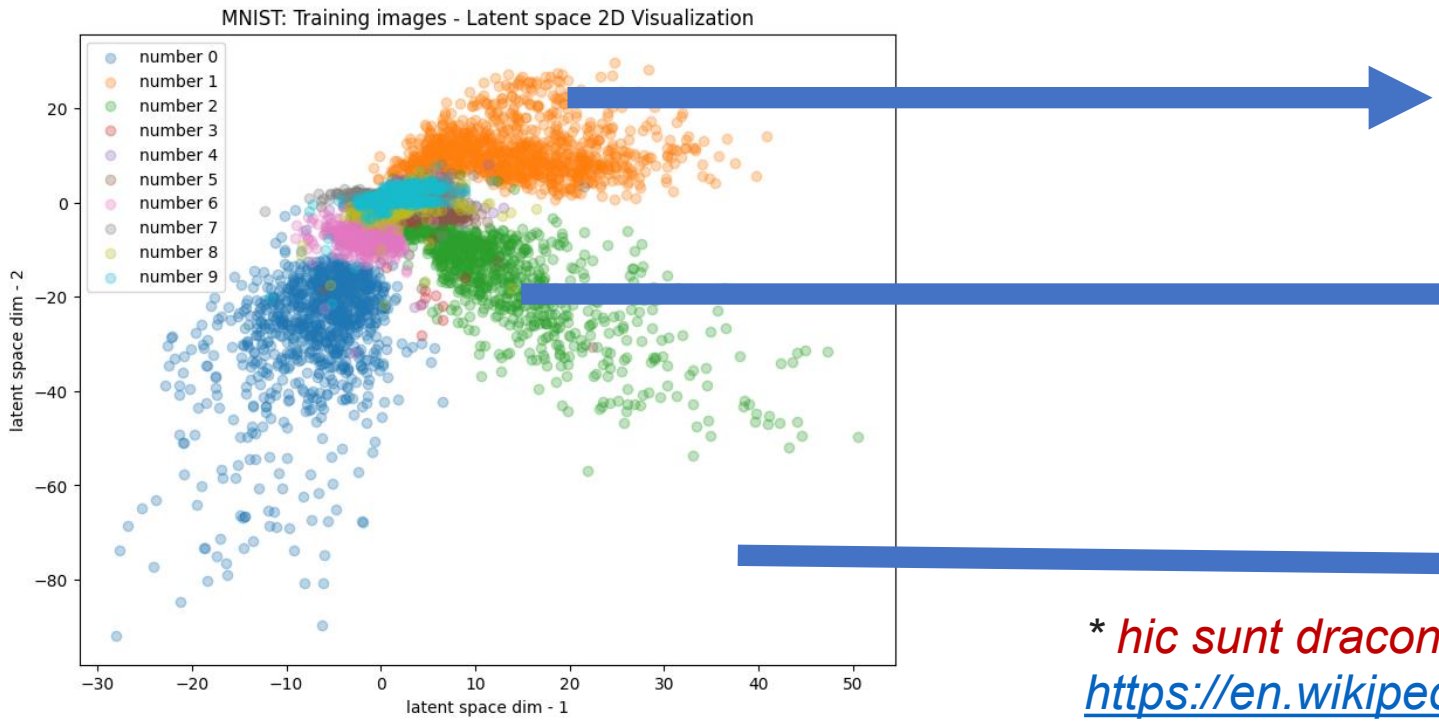
# Recap – AE, VAE & Generation

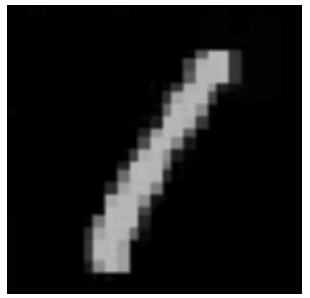
- **A**uto-**E**ncoder model at a high level:



# Using AE as Generative AI models

- AutoEncoders models already have a **generator**(**decoder**) component
- Problem with a pure **AE** is:
  How to use the trained model to generate output?



?

* *hic sunt dracones(!)*
https://en.wikipedia.org/wiki/Here_be_dragons

# VAE uses, training and further comments

- **VAE** are trained by compressing information through a latent-space to generate pseudo-data with less noise.

- The ideal latent-space distribution is model dependent, but the first attempt is often to assume a normally distributed probability-space will work.

  It is, *in principle*, possible to do better, but at the cost of complexity and computational demand.

- Knowing that **VAE** **<u>are not concerned with faithfully reproducing input data</u>** with 100% fidelity, we can modify the loss function such that:

$$Loss = Loss_{\boldsymbol{BCE}}(\boldsymbol{logits}) + \boldsymbol{C} \times \ Loss_{\boldsymbol{KL}}(\boldsymbol{\mu}, \boldsymbol{\sigma})$$

  Where $\boldsymbol{C}$ is an additional hyper-parameter determining how strong the sampling of the latent-space impacts the training.

*NB: Sometimes referred to as* **β***VAE with* **β** *a free hyper-parameter in the loss-function*

# Generative AI models AE & VAE

- **AE** are **simpler to train** and learn more of the smaller features in a dataset
- **VAE** are **easier to use to generate** and can produce smoother outputs

- Both are useful for solving certain problems such as de-noising, producing pseudo-data to fill in gaps or generating simulated data.

  Both are **very well suited** to the solving the problem of **anomaly detection**.

- Neither are well suited to solving the problems:

  - How to generate a real-life example of a certain input class?
  - How to generate a pseudo-datapoint from a random input?
  - How can I be sure that the simulated data is indistinguishable from real data?

# Training AE & VAE models

- You can train/optimize your model for any given task.
- E.g.:

* There is no universal requirement that the input data into the model be used to constrain the output from the model.

  - You can train a model purely to remove input noise from data like a VAE.
  - You can train your model based on images to generate text.
  - You can train your text model to generate a new image.
  - You can train your model on blue images to generate red ones.
  - You can train an image model to generate text based on images.

Original Image

Image Mask

# Generative Adversarial Neural networks

- **GAN** models are a type of ML model which are better suited to solving 2 of the shortcomings with **AE**/**VAE**:

 - Generate better simulated data (***precision***)
 (**AE** results are often low-precision, **VAE** not "*realistic*")

- Able to better generate "unseen" entries; indistinguishable from real data (***interpolation***)

- They also tend to:

 - Generate simulated data with finer details
 - Have more stable image generation over wider range of possible values

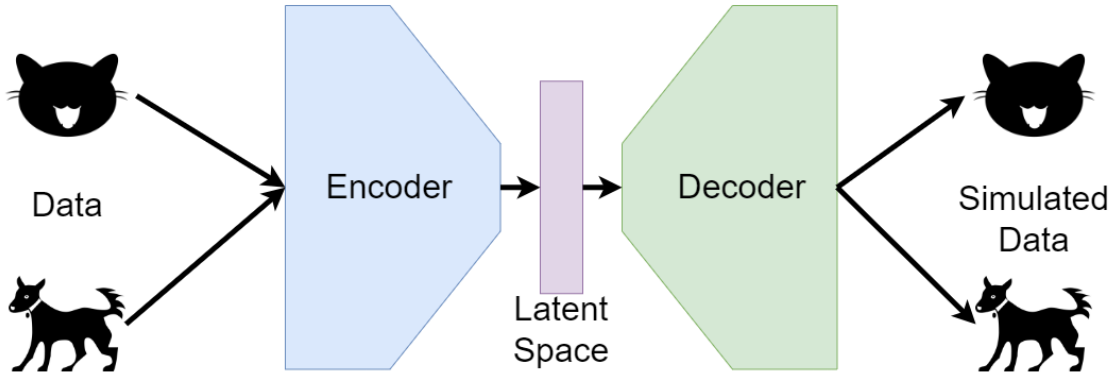
# AE vs GAN design

- Both (V)AE and GAN are built using similar (if not identical) components.
- For a (V)AE the whole network is dedicated to learning the structure of the input data and extracting the relationship between the inputs.
- For a GAN;

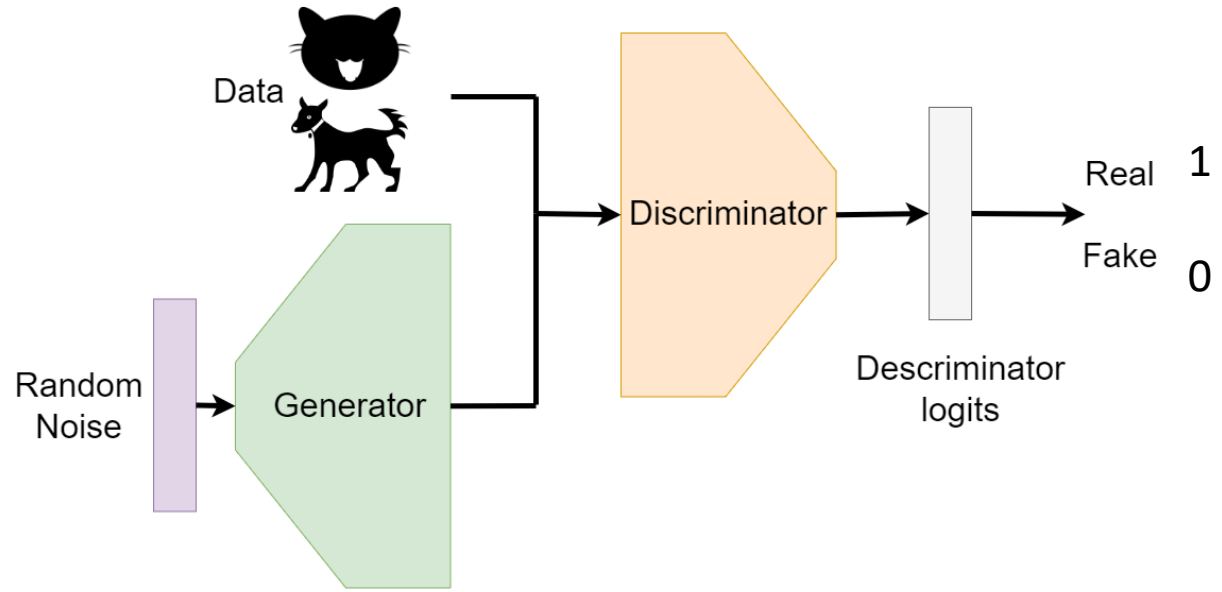
  The *Generator* focusses on building *new images* using a random input.

  The Discriminator is focussed on determining whether the input to the model is *real* or *fake*.

AutoEncoder:




GAN:



# Comparing GAN to VAE

- The *encoder* in a **VAE** can be built with the same graph(model) as a *discriminator* a **GAN**.
- The *decoder* in a **VAE** can be built with the same graph(model) as an *encoder* in a **GAN**.

- In a **VAE** these graphs are connected through the latent-space.

  In a **GAN** these graphs are only semi-connected through loss functions.

- The biggest difference between how the individual graphs are used is how the full model is constructed and trained.

# GAN loss function

- Ignoring the Generator, a Discriminator is trained using the **BinaryCrossEntropy** Loss function.

$$L(y, \hat{y}) = -[y.\log(\hat{y}) + (1 - y)\log(1 - \hat{y})]$$

1-bit Classifier:
***True/False***

- For real images: $L(1, \hat{y}) = -\log\big(D(x)\big)$

- For fake images: $L(0, \hat{y}) = -\log\Big(1 - D\big(G(z)\big)\Big)$

# GAN loss function

Combining the loss functions for possible **GAN** outputs gives the well cited and is slightly famous “min-max” function:

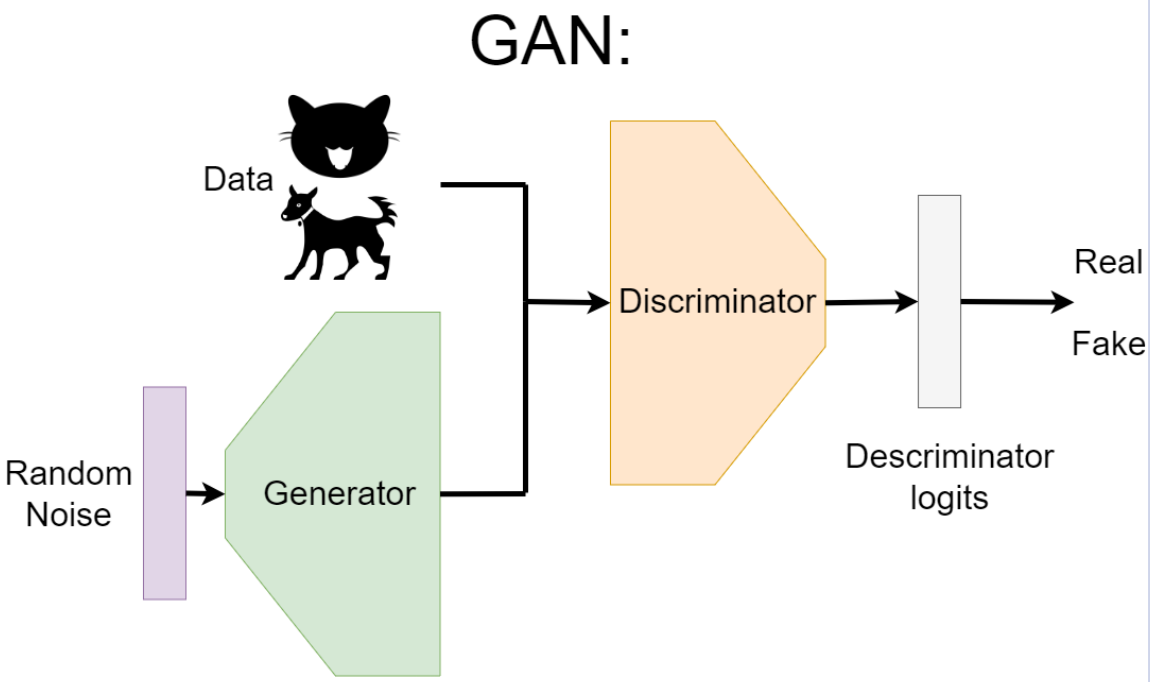
$$\min_{G}\max_{D}\ [\sum_{x}[\log(D(x))] + \sum_{z}[\log(1 - D(G(z))]\ ]$$

*Real Data*

**Generation Seed**

We’ll discuss implementing this later and in the workshop.

# GAN model




| Pros | Cons |
| --- | --- |
| • Powerful as **anomaly detection** models<br>• Generated data has **higher precision**<br>• Well suited to **data augmentation** tasks<br>• Supports complex conditioning in the form of **cGAN** | • **Model collapse** can lead to generated data failing to fully match whole phase-space<br>• **Training instability** in GAN can cause the models to rapidly diverge or mis-converge<br>• Potentially **computationally** more **expensive** than other comparably complex models<br>• Discriminator hitting 50% during training can cause the training to collapse |

# GAN model

- (**V**)**AE** can generate input based on a *Latent-Space* coordinate
  Different coordinate ➔ different generated output
  Sampling can be difficult

- **GAN** can generate input based on ‘any’ suitably random input.
  => Sampling can be much easier

- **GAN** generator can be thought of as ‘de-noising’ and upscaling input random-noise into pseudo-data

# Conditional-GAN (cGAN)

So far, we've discussed training and using models for generating based on a random input, but all these models have had 1 problem.

> **How do we control the category of the generation?**

For both **GAN** and **VAE** we can be 'clever' with our decisions and select a random point in the latent-space to generate a class that we're interested in.

This is clumsy, prone to errors, and constantly changes with training/seed/…

***> There is a better way(!)***

# Conditional-GAN (cGAN)

It is possible to build a **Conditional VAE** or a **Conditional GAN** model.

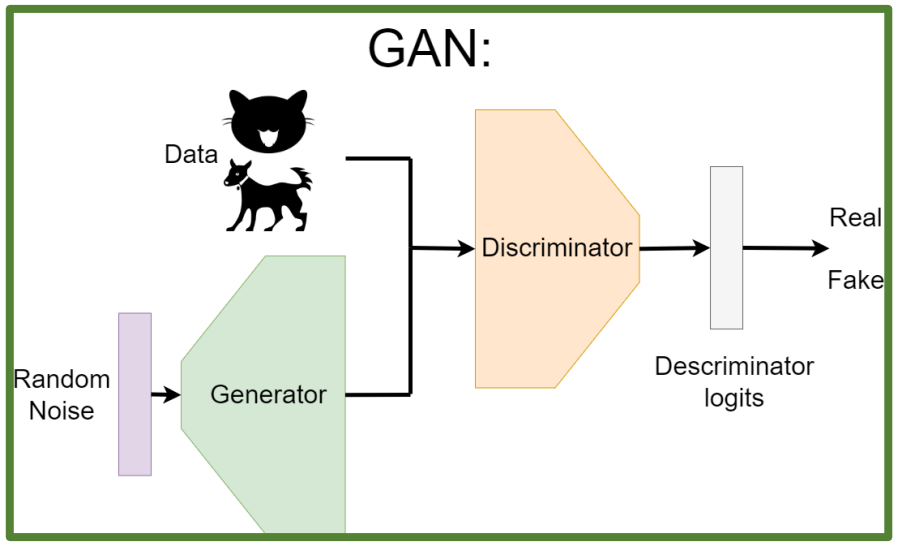
These models incorporate *tokens*, or *labels* into the model during training.

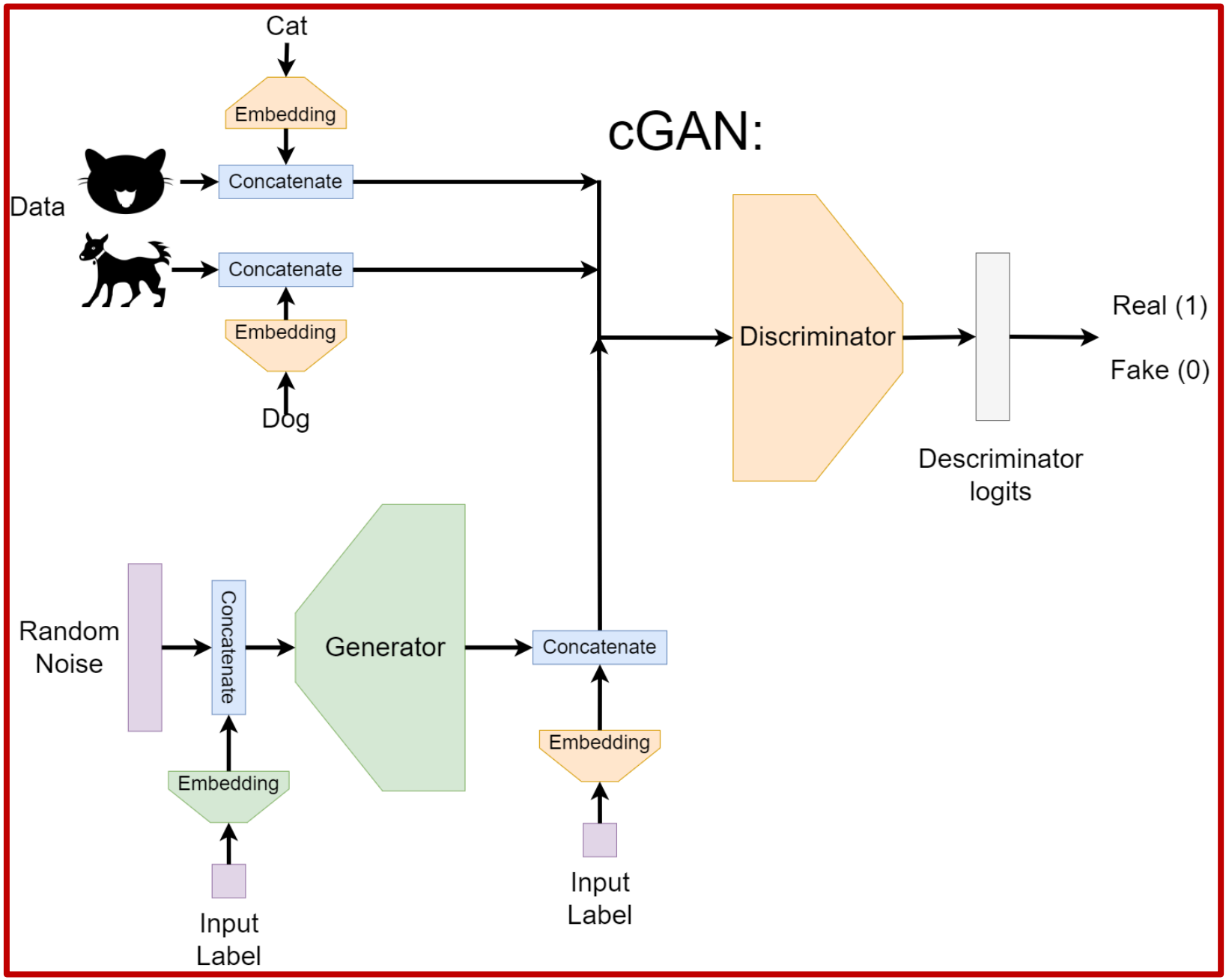
This allows the network to train itself such that we can request a certain type of input based on a certain combination of tokens.

One of the most common tools for this is referred to as an Embedding.

Tomorrow we will be training a **cGAN** from scratch to generate images of numbers on request.

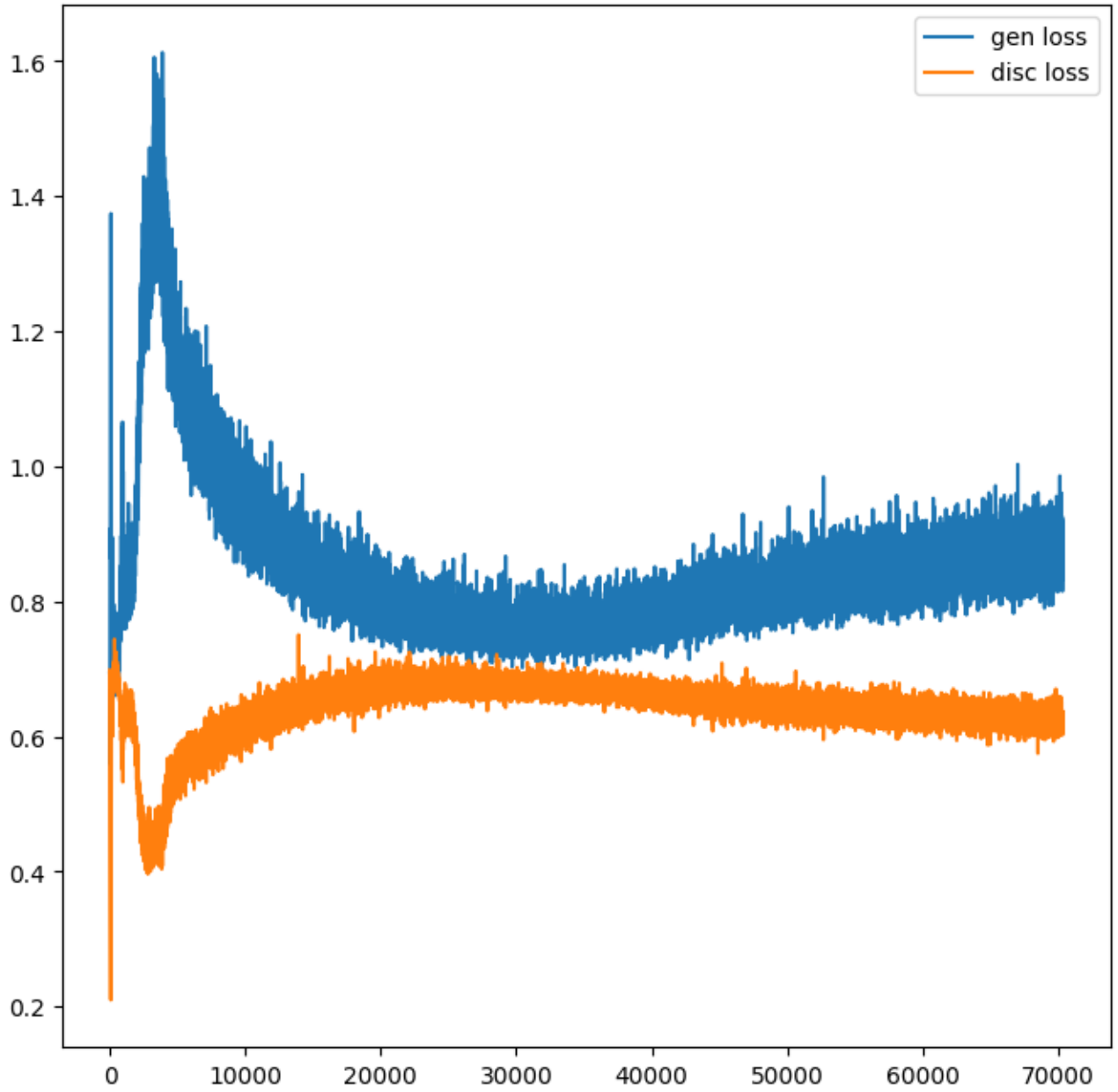
# GAN vs cGAN


GAN:
Data
Random Noise
Generator
Discriminator
Descriminator logits
Real
Fake


cGAN:
Cat
Embedding
Data
Concatenate
Concatenate
Embedding
Dog
Discriminator
Descriminator logits
Real (1)
Fake (0)
Random Noise
Concatenate
Generator
Concatenate
Embedding
Embedding
Input Label
Input Label

# cGAN training

- Training a **cGAN** or **GAN** model is pitting 2 different models against each other
- As the generator gets better the discriminator struggles more, and vice-versa
- If left to train for long enough the goal is to reach a point where both models reach an equilibrium.
- This can be ***very*** expensive(!).



# *Training a recap*

1. Model is constructed and data is batched up for processing.

2. Data is fed through the network to calculate the loss for each batch.
3. Move backwards through the network (graph) to calculate the gradient wrt the data in each batch. ***(Back-Propagation)***
4. Optimizer applies a Parameter Update based on gradient from training batch.
5. Repeat steps 2-4 until you converge or run out of money/time/data.

6. Use model to make a prediction & determine how accurate and efficient the model is.

# cGAN training

- Building a **cGAN** in code is a bit more involved than building a **uNET** or a **VAE**.

- The entire graph is composed of 2 effectively independent models linked via their loss function:

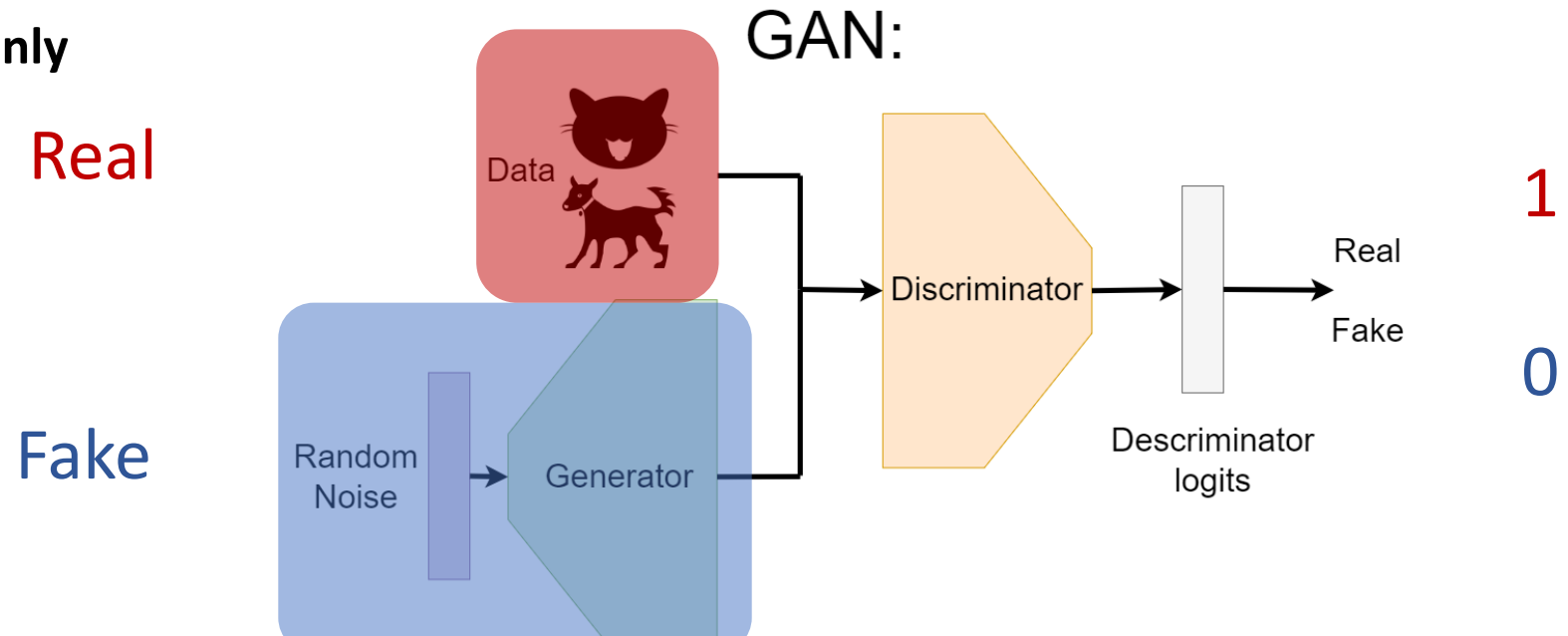
  **Generator** & **Discriminator**

- Training the whole model requires training in parallel **2 full training-graphs** with **2 different losses**.

# cGAN training – Discriminator

- Training a **Discriminator** **1-step** requires training it on both *real* and *fake* data and calculating the loss.

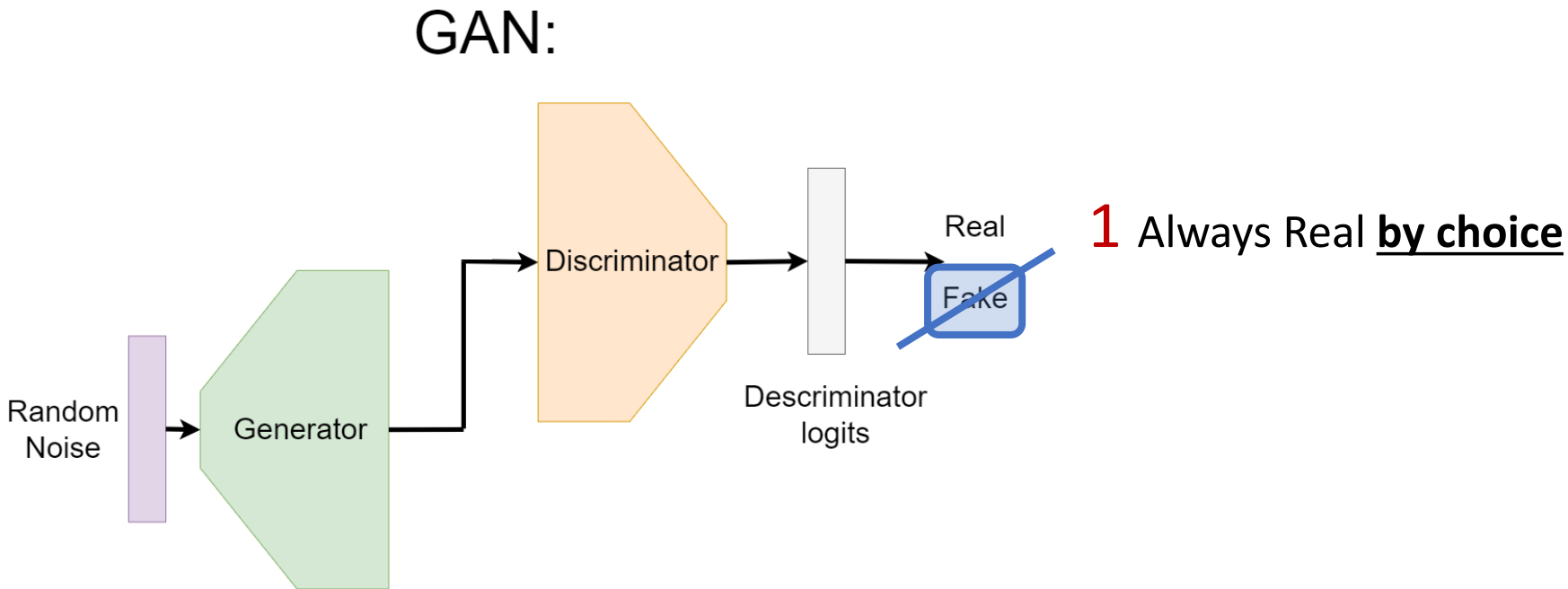
1. (Optional) Fix/Freeze the Generator at its trained state
2. Generate some new fake data
3. Train the Discriminator to tell real (training) data from fake
4. Update the Discriminator **only**



# cGAN training – Generator

- Training the **Generator** **1-step** requires training the generator to '*trick*' the *Discriminator*.

1. Generate new fake data (*whatever this fake data looks like*)
2. Evaluate the full model from noise to "decision"
3. Update the Generator **only**



# GAN vs cGAN

- **cGAN** have the advantages of **GAN** but can take even longer to train
- **cGAN** are better than **GAN** in the regard that the output classification is known
- Could train a **cGAN** on more complex classifiers than numbers (tokens).

  Complex tokens with complex models can be used to generate high fidelity images, or other outputs…
- Problem we've not discussed:

  > **How can we determine "*how realistic*" the output from a cGAN generator is?**

# cGAN model training

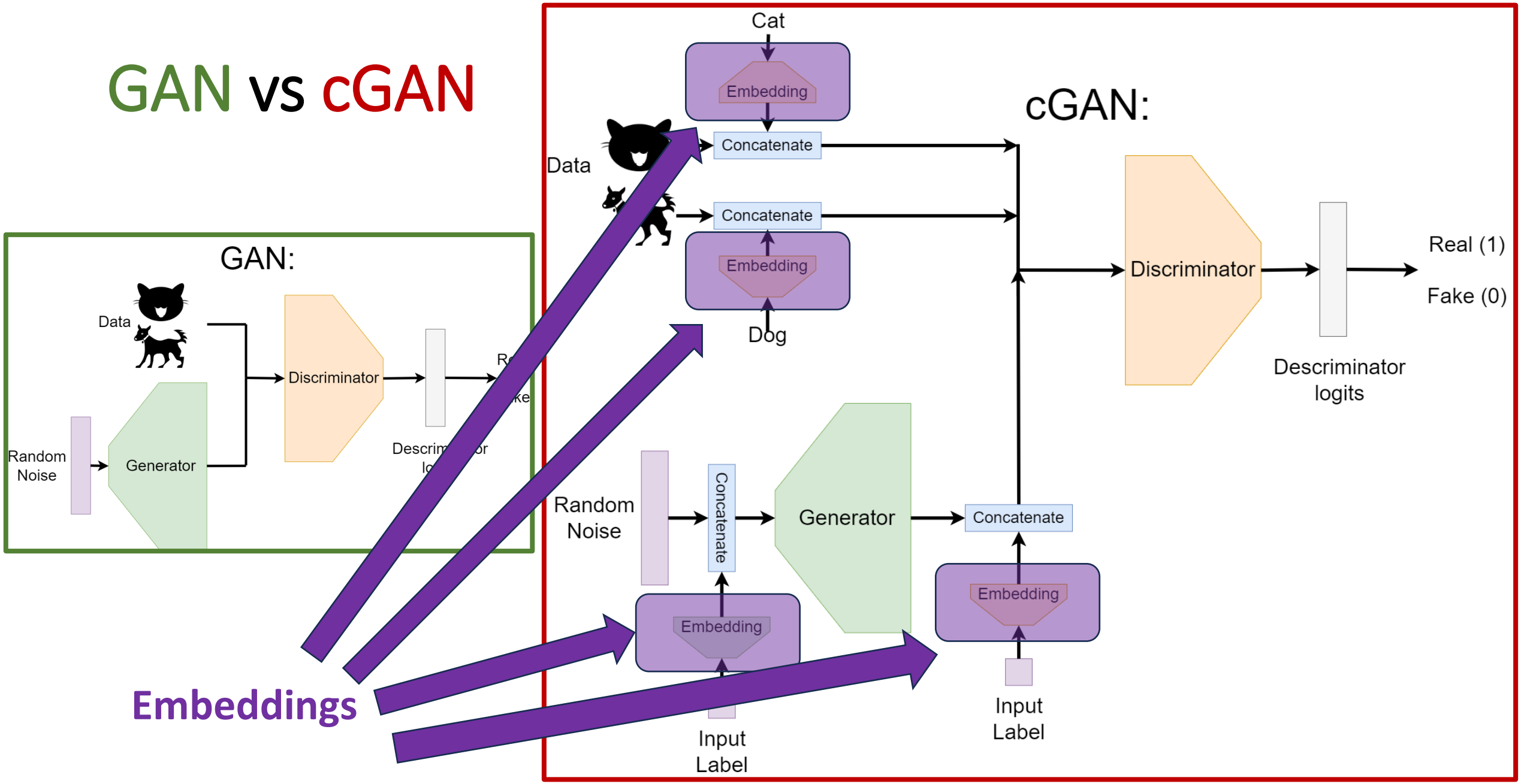
- The **Generator**(Decoder) component needs to know about what category/label it's trying to generate a new image of.

- The **Discriminator**(Encoder) component is trying to satisfy the condition, "is this a real image".

- The question becomes how close are the requirements:

  "is this a real image?"
  vs
  "is this a real image of category X?"

# GAN vs cGAN


GAN:
Data
Random Noise
Generator
Discriminator
cGAN:
Cat
Embedding
Concatenate
Data
Concatenate
Embedding
Dog
Discriminator
Real (1)
Fake (0)
Descriminator logits
Random Noise
Concatenate
Generator
Concatenate
Embedding
Embedding
Input Label
Input Label
Embeddings

# Embeddings

Embeddings in Machine Learning models can effectively take whatever form you find most convenient. However, some of the most used are:

- **Tabular** Encodings
- **Vector** Encodings
- **Positional** Encodings

Added alongside the data

- **RoPE** (Rotary Positional Encodings)

**NB:**

At this point people often talk about data being composed of token.

A single token is a single datapoint within a stream of data.

In a picture this can be thought of as a single pixel arranged within an image.
In textual data words (or fragments) in a sentence.

Multiple streams form one batch.

# Embeddings – Tabular

- Tabular embeddings are among the simplest to implement and the simplest to conceptually understand.

- Effectively we give every unique value a random location in n-dimensional space.

  Then, during model training, we allow back-propagation to move these locations.

- After training with many, many classes you would expect to see any concepts which have been “extracted” by the model represented by clustering in this embedding space.

- Labels embedded this way tend to be discrete.

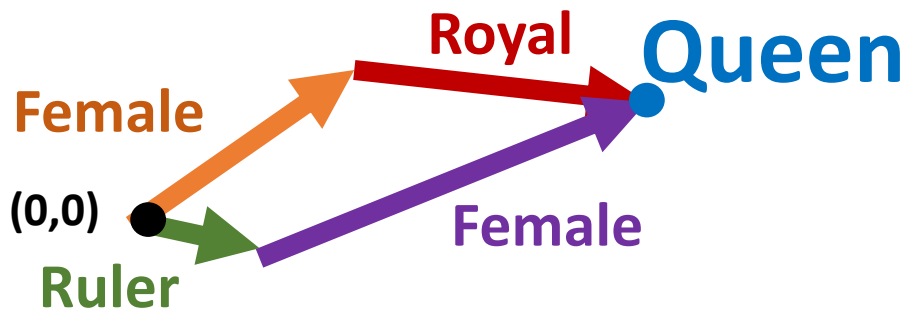
- Very *similar to*, but **distinctly different from**, model Latent Spaces.

# Embeddings – Vector

- Vector embeddings take the concept of absolute embeddings to the next level.

- It's expected that labels embedded this way are continuous or have internal relationships/semantics.

- The simplest examples of this come from language where we can get to words via multiple routes.




- The model in some way encourages the embeddings to embed into a relational space.

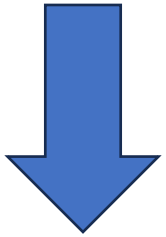
# Embeddings – (*Fixed*) Positional Encoding

- Positional encodings are based on the principle of preserving the ***relative*** position of a feature **within the stream** of data.

- The simplest way of encoding this is to use long-range sinusoidal representation of the position of a token in an input stream.

- Sinusoidal functions have limited output numerical range and are approximately linear over short distances for long streams.

a b c d e f g h I j k l m n o p q r s t u v w x y z



***sin(i/26)***

*e.g. encoding letter position in the alphabet*

(a,0.6e-3) (b,1e-3) (c,2e-3) (d,3e-3) … (z,1.7e-2)

# Embeddings – RoPE

- Rotary Positional Embeddings are different from *Fixed* Positional Embeddings.
- Positional embeddings act on the output of projections from tokens. RoPE acts on projected outputs from the input dataset.

Input data ***x*** is ***projected*** (via fully connected mlp) to different layer outputs ***q*** and ***k***

$$\boldsymbol{q} = W_Q \boldsymbol{x_1} = |\boldsymbol{q}|(cos\theta_q, sin\theta_q) \;, \qquad \boldsymbol{Q} = \boldsymbol{R}(\theta_1)\boldsymbol{q}$$
$$\boldsymbol{k} = W_K \boldsymbol{x_2} = |\boldsymbol{k}|(cos\theta_k, sin\theta_k) \;, \qquad \boldsymbol{K} = \boldsymbol{R}(\theta_2)\boldsymbol{k}$$

$$\boldsymbol{Q}^T \boldsymbol{K} = |\boldsymbol{q}||\boldsymbol{k}| \cos\left(\theta_k - \theta_q + \Delta\theta\right)$$

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{cos}\theta & -\boldsymbol{sin}\theta \\ \boldsymbol{sin}\theta & \boldsymbol{cos}\theta \end{bmatrix}$$

# Embeddings – RoPE

- **RoPE** then splits these outputs into pairs and uses rotary transformations to apply angles from pairs of projected outputs. (***individual angles related to global position***)

- Any difference between the angles from 2 pairs is now related to the **relative position** of tokens in the data. (***differences related to relative position***)

- **RoPE** allows for a model to *automatically extract relative relationships* during learning. It also allows the model to potentially extrapolate to new sequences beyond the training window.

# GAN training

Training a GAN model is in general more difficult than training simpler models.

There are 2 statistical minimizations happening in parallel, both competing to optimize for different goals.

In principle this is happening over our loss-function space that is a function of both a function of the input data and the model.

# Additional Important Concepts in ML training

- Initialization
- Data Augmentation
- Normalization; Batch, Layer, …
- Adaptive Learning Rate & LR Decay
- Gradient Clipping
- Dropout/Noise
- Label Smoothing
- WarmUp in training
- Snapshotting

# ML training – Initialization

- As covered previously models should be initialized with a good random values.
- Remember trained models will often have most parameters close to 0

- The best test of if a model is behaving randomly is to compare the output from the initialized model with what is expected from a random black-box.

- Low-values uniform random initialization doesn't guarantee that the model is well behaved but increases the chances that it is close to a well-behaved region of "***loss-space***"

# ML training – Data Augmentation

- We've already covered that data augmentation allows us to supplement a limited dataset to improve ML training

- This strategy has limitations, including shuffling data between:

  Storage<->CPU<->RAM<->CPU<->Accelerator
  vs, Storage<->Accelerator

- However, this does push models to learn more generic idealised solutions to problems.

# ML training – Normalization

- Inputs may run over all orders of magnitude; normalization helps ML training cope with this in an automated way
- The **2** main areas where this impacts training are: model design & parameters and gradients in optimization
- When designing a model, it's common to introduce normalization layers. This normalizes the input from a previous layer for output.
- When considering the gradients during optimization the gradients of all parameters are normalized to reduce the impact of a few gradients on the whole parameter space.

# ML training – Adaptive LR

- Large steps in "***loss-space***" during training can lead to unstable results
- Changing the LR during training means training remains stable as it approaches a minima
- Adam/AdamW and other optimizers have built-in adaptive LR algorithms with various tuning parameters
- You can also explicitly request the optimizer use a given strategy such as increasing the relative step size once the training becomes more predictable

# ML training – Gradient Clipping

- When gradients become too large it can drag a model into an *unstable* area of "***loss-space***"
- Gradient Clipping helps keep the model in well defined regions

- Calling this clipping is a misnomer

- The simplest approach does just clip parameters to have maximum absolute values
- The most used approach just uses "clipping" to trigger a normalization of gradient values

# ML training – Dropout / Noise

- Using dropout or noise layers in model design is an attempt to force a model to improve its internal representation of what the ideal solution

- Dropout randomly zeros nodes in a model and noise adds random offsets into a model. This varies from batch to batch during training
  The effect is that the trained solution becomes more invariant to noise and/or offsets

- Unfortunately, this is roughly equivalent to data augmentation in that it increases the amount of training required for model to converge.
- Sometimes this can slow training by a significant %-age, but sometimes it can double the computational training requirements or more…

# ML training – Label Smoothing

- This is usually a relatively easy approach to take in improving model stability
- For labelled data where you can't trust the labels 100% you instead report a distribution of likely values as the truth to the model
- This helps improving training stability by communicating to the model how the labels aren't certain
- However, it can slow training as small gradients flow back from incorrect labels and upset the model stability

# ML training – WarmUp

- WarmUp in ML training is a period right at the beginning of training
- During this phase the behaviour of the model is less predictable, and the loss function is much more random
- The concept of warm-up is quite simple, for the first n steps during training reduce the LR to a small value to reduce the impact of large fluctuations during the start of training
- After n steps the training resumes as normal

# ML training – Snapshotting

- The idea of snapshotting is to capture the state of a model during training and commit it to storage
- This allows training to recover and to be modified (*nudged*) should the training become unstable due to unexpected behaviour
- When training large models this can take a significant fraction of the training time (20%+ in some cases)
- Saving, pausing and loading a model to/from disk requires preserving the current state of the model being trained as well as the full internal state of the optimizer(!)

# GAN training – WGAN

- **Wasserstein GAN** was introduced in 2017 by researchers at facebook. https://arxiv.org/abs/1701.07875

- Some key differences to a standard **GAN** model:

  – Replace the Discriminator with a Critic *(main difference is raw logits output)*

  – Change loss function to have a gradient penalty measuring the difference between data real and fake data populations

  – The optimization is acting to optimize slightly simpler loss functions of ***max[E(f(real)) – E(f(fake))]*** (values not bound by a probability space)

# GAN training – WGAN

- The simplest equivalent to the **WGAN** is to introduce a gradient clipping into the training.

- We won't be covering **WGAN** in this course due to the additional complexity in training implementation vs **GAN** or **cGAN**

- Effectively:

**(c)GAN**: How well can we classify real vs fake inputs?
**WGAN**: How much higher can I score real vs fake inputs?

# Diffusion models

Simple premise;

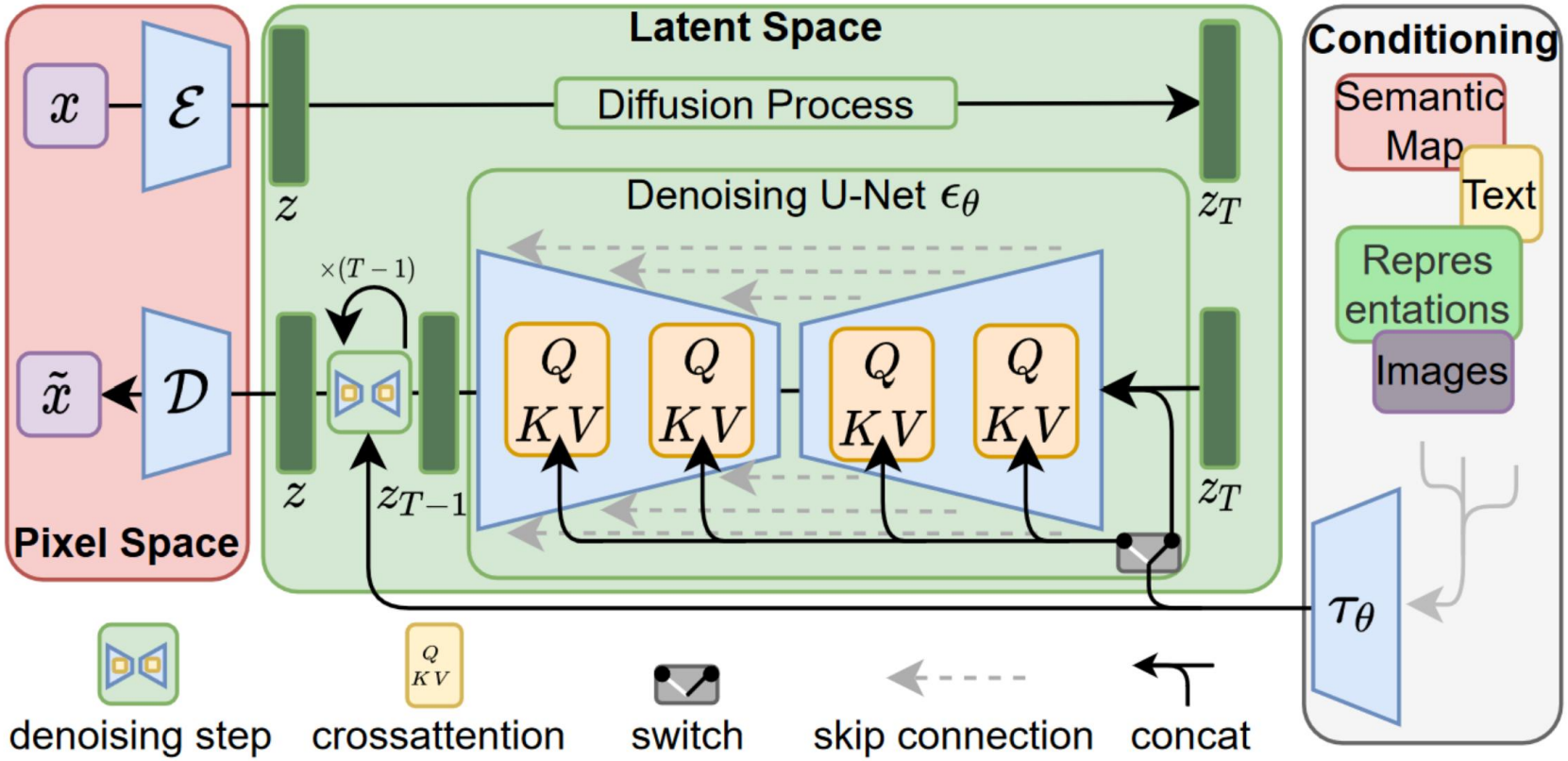
*First publication: https://arxiv.org/pdf/1503.03585*

1. We can train a **VAE** like model to “de-noise” a noisy image.
2. What if we take this model and “de-noise” from ***pure noise*** repeatedly?

*> This model is part-inspired by diffusion modelling in non-equilibrium statistical physics.*

Start from an image(signal) and add noise to diffuse the information moving forward.

Whilst in back-propagation train the model to undo the diffusive process and to create signal out of the noise.




https://arxiv.org/pdf/2112.10752

# Diffusion models

- Diffusion models are stepped in “time” based roughly on real world diffusion that we see in Physics.

- It’s based on **2** separate processes:

  **Forward:**

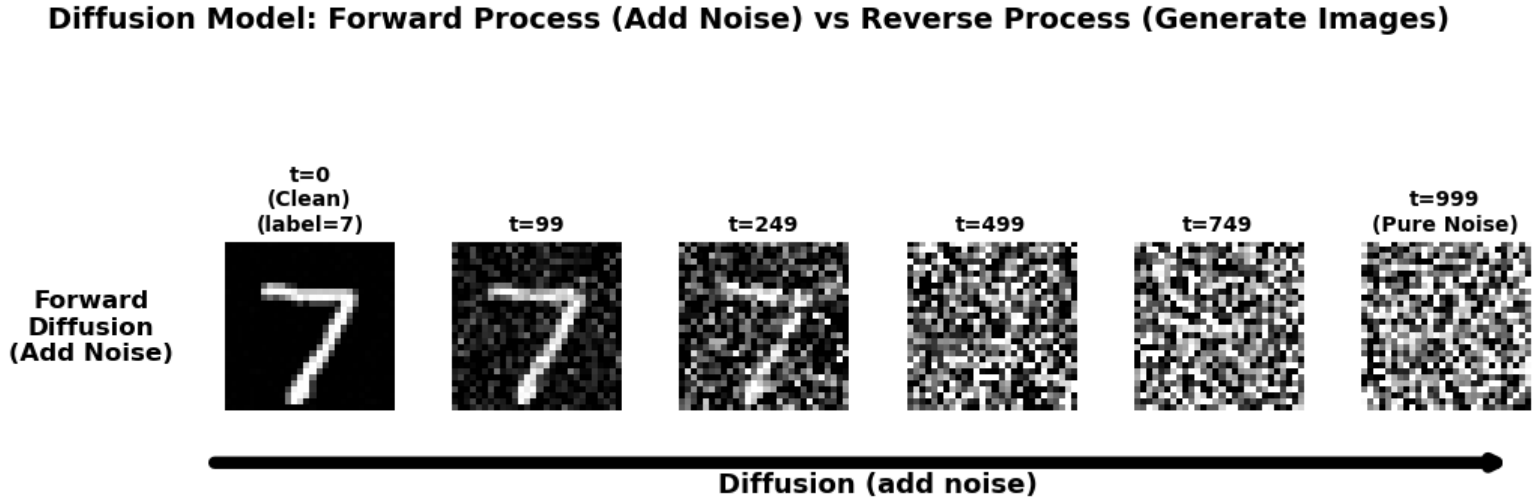
  For “t” time steps; starting from our starting image add more and more noise to the image, until the result is a pure noise image.

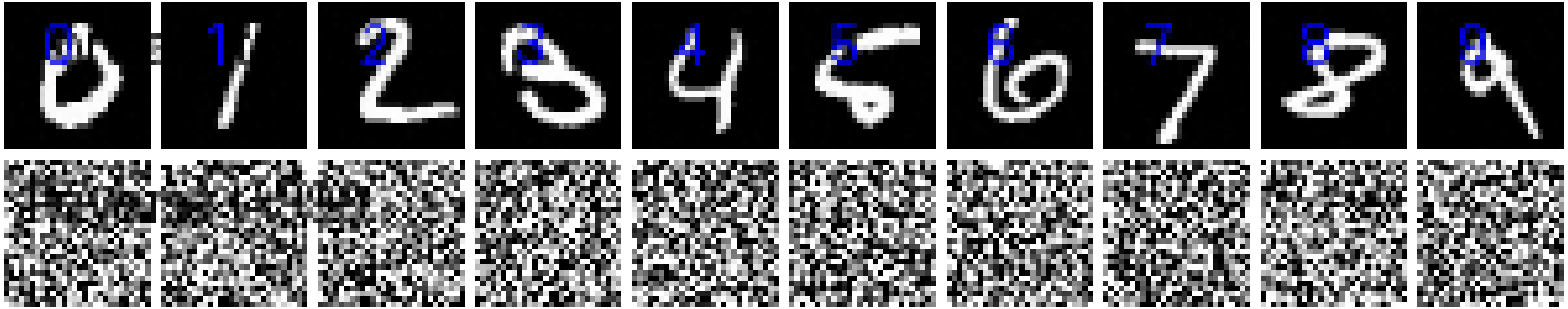
  **Reverse:**

  For “t” time steps; starting from pure noise, in each step “undo” adding noise to the current image.

# Diffusion models

Example!



# Diffusion models – (*a very quick aside*)

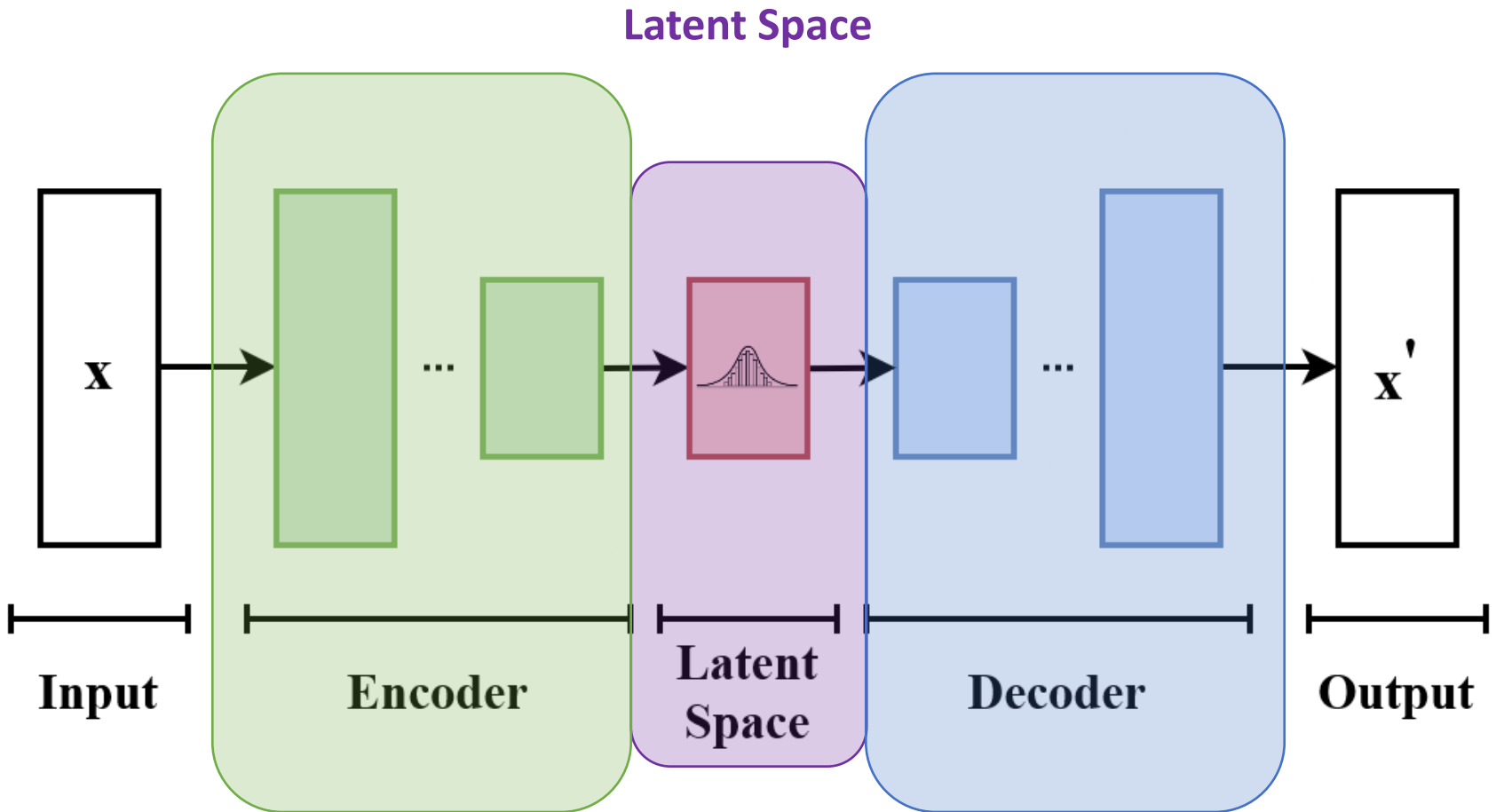
- Compared to a **GAN**, **diffusion** models can produce very high-quality data
- **Diffusion** models de-noise in-place and continual de-noising generates an image vs **GAN** where generation is trained to “trick” an upstream discriminator
- Generating from a **diffusion** model can be expensive, but can be combined with faster sampling/distillation techniques to reduce runtime compute (often with a quality trade-off)
- A key practical difference is controllability: diffusion models often support strong conditioning (e.g., text, masks, guidance) to steer the output more directly

# Last Week's Workshop

- We were building&training a VAE model to train on the numerical mnist dataset.
- This model is composed of 2 components which should be symmetric to each-other.
- There was a subtle "bug" in the Question version of the notebook ☹
- I think it's worth discussing a) what this was and b) why it doesn't actually matter ☺

# Last Week's Workshop

- At a high level we want our model to be symmetric around the LS



# Last Week's Workshop

The encoder model is composed of layers.

Each layer does something like the following:

*# input (B, 1, 28, 28)*

- **Filter** the input (via **conv2d** transformations) *# output: (*B,1,28,28)
  *extracting information*
- **Normalize** across the output (**BatchNorm2d**) *# output: (*B,1,28,28)
  *giving us numerical stability*
- **Downsample** the output (**AvgPool2d**) *# output: (*B,1,14,14)
  *picking out only the key features*

# Last Week's Workshop

The decoder model is also composed of layers.

The ideal version of the model is to run this in reverse:

*# input (B, 1, 14, 14)*

- **upSample** the output (**interpolate**) *# output: (*B,1,28,28)
  *UpSampling from previous layer*
- **Filter** the input (via **conv2d** transformations) *# output: (*B,1,28,28)
  *Adding back in fine-detail*
- **Normalize** across the output (**BatchNorm2d**) *# output: (*B,1,28,28)
  *Numerical Stability*

# Last Week's Workshop

- Incorrectly the notebook suggested you do the following:

  add-Detail → Normalize → UpSample

  rather than:

  UpSample → add-Detail → Normalize

- As every up-sample step is followed by an "add-Detail" step in the whole model chain this doesn't make "too much" of a difference.

  Aka, it still works and doesn't seriously impact the learning outcome.

  But you may find that Building your model the 2nd way leads to even more fuzzy/blurry output compared to the 1st way.

# Today's Workshop – cGAN

- Today we will be building a cGAN model.
- This is very similar to the AutoEncoder but the decoder and encoder parts of the model have effectively been reversed.
- We will also be introducing the simplest form of Embedding: "Absolute Embedding" into the model.
- As with a VAE we will start with building our model from components and checking for their consistency before beginning training.

# Today's Workshop – cGAN

- During training you should expect to see example images output per epoch.

- If your images look like pure-noise or blurry and/or your loss values remain static for >1epoch your model has collapsed during training.

  → Please ask for help in this case! ←

- Ideally you should see number-like images generated from pure noise seeds after ~10epoch or so.

# Today's Workshop – cGAN

- After training we will take some time to explore the use of Embedding layers in this model.

- It's interesting to see how the model choses to cluster the 10 classifications of images (or not)

# Today's Workshop – cGAN

- An interesting side note.

- Today's cGAN model has a *LOT* of tuneable hyper-parameters:

  - LearningRate
  - BatchSize
  - Model Width
  - Input Noise dim
  - Label Embedding dim
  - GradientClipping threshold
  - Optimizer Settings

Changing most of these parameters up/down by sensible factors causes model training to fail.

This example cGAN model is surprisingly unstable(!)

Adjusting:
Model 32->16 or
GradientClipping 1.0->2.0 you will probably see this struggle to train and/or completely collapse!

# cGAN model vs (V)AE/GAN

- Normally a **(V)AE/GAN** model takes one set of arguments:

  **ae_model.forward(data)**

  **data.shape == (batch, channels, x, y)**

- **cGAN** takes 2 arguments:

  **gan_model.forward(data, labels)**

  **data.shape == (batch, channels, x, y)**
  **labels.shape == (batch, 1, 1, 1)**

# cGAN model evaluation

- The heart of a **cGAN** is still a graph of compressing or decompressing information within the model.
- These graphs are the same as **(V)AE** and are designed to take data in a certain format.
- Labels in the case of mnist are just numerical numbers, of dimension (1,1)
- To use the labels during training we must *change* their dimension.

# cGAN model evaluation

- The best approach to do this is to embed the label data and scale the result.
- **nn.Embedding**
  https://docs.pytorch.org/docs/stable/generated/torch.nn.Embedding.html

- It’s probably possible to embed complex labels in very low dimensional space, but the more freedom the embedding has the easier it learns.
- Scaling is relatively easy, either use **.expand** to copy values, or connect everything together with a DNN.

# cGAN model evaluation

- In our example notebook we will first embed our labels into a 3d embedding space (*3d just a "choice" here*).
- Then, these 3 dimensions are then scaled up to the inputs for the "***encoder***" or "***decoder***" models.
- These scaled up inputs are then taken as a new channel of information by our model when learning how to scale up/down.
- The **encoder** here is our **discriminator**
- The **decoder** here is our **generator**

# cGAN model evaluation

Before our discriminator in a **GAN** only has access to information with the shape:

**(batch, 1, 28, 28)** when looking at mnist

Now, our discriminator in a **cGAN** accesses information in the shape:

**(batch, (1+3), 28, 28)** when using embedding

# Today's workshop

- Training a cGAN with a smaller dataset can be unstable
- Training a cGAN with only CPUs can take a very, very long time

- The implementation in today's notebook has been tested on the machines in labs and should run from start to finish in ~20min on a desktop CPU

- I've provided a lot of the training logic in code as implementing this correctly and debugging this is can be difficult and time consuming
- I encourage everyone to examine the training code and ask questions

- **I would rather see you successfully train a cGAN, generate new images & examine the behaviour of the Embedding class for trained vs untrained**

# Further Reading (ahead of next week)

- 3Blue1Brown: https://www.youtube.com/@3blue1brown/videos

What we've covered so far:

- 1) https://www.youtube.com/watch?v=aircAruvnKk
- 2) https://www.youtube.com/watch?v=IHZwWFHWa-w
- 3) https://www.youtube.com/watch?v=Ilg3gGewQ5U
- 4) https://www.youtube.com/watch?v=tIeHLnjs5U8

# Further Reading (ahead of next week)

- 3Blue1Brown: https://www.youtube.com/@3blue1brown/videos

What we will cover in the last lecture:

- 5) https://www.youtube.com/watch?v=wjZofJX0v4M
- 6) https://www.youtube.com/watch?v=eMlx5fFNoYc
- 7) https://www.youtube.com/watch?v=9-Jl0dxWQs8

- Also does excellent videos on other maths topics.
  (I'm a Physicist but I'd call these "*accessible*" to me)

# Further Reading – Other Resources

- Andrej Karpathy: https://www.youtube.com/andrejkarpathy
- Excellent resource if you are more of a coding centric person
- https://github.com/karpathy/nanoGPT
- If you have access to a GPU, you can train a basic Shakespeare LLM model yourself, from scratch in ~1weekend.
  It will spout out sentences which are utter nonsense, but it's cool what you can do to start.