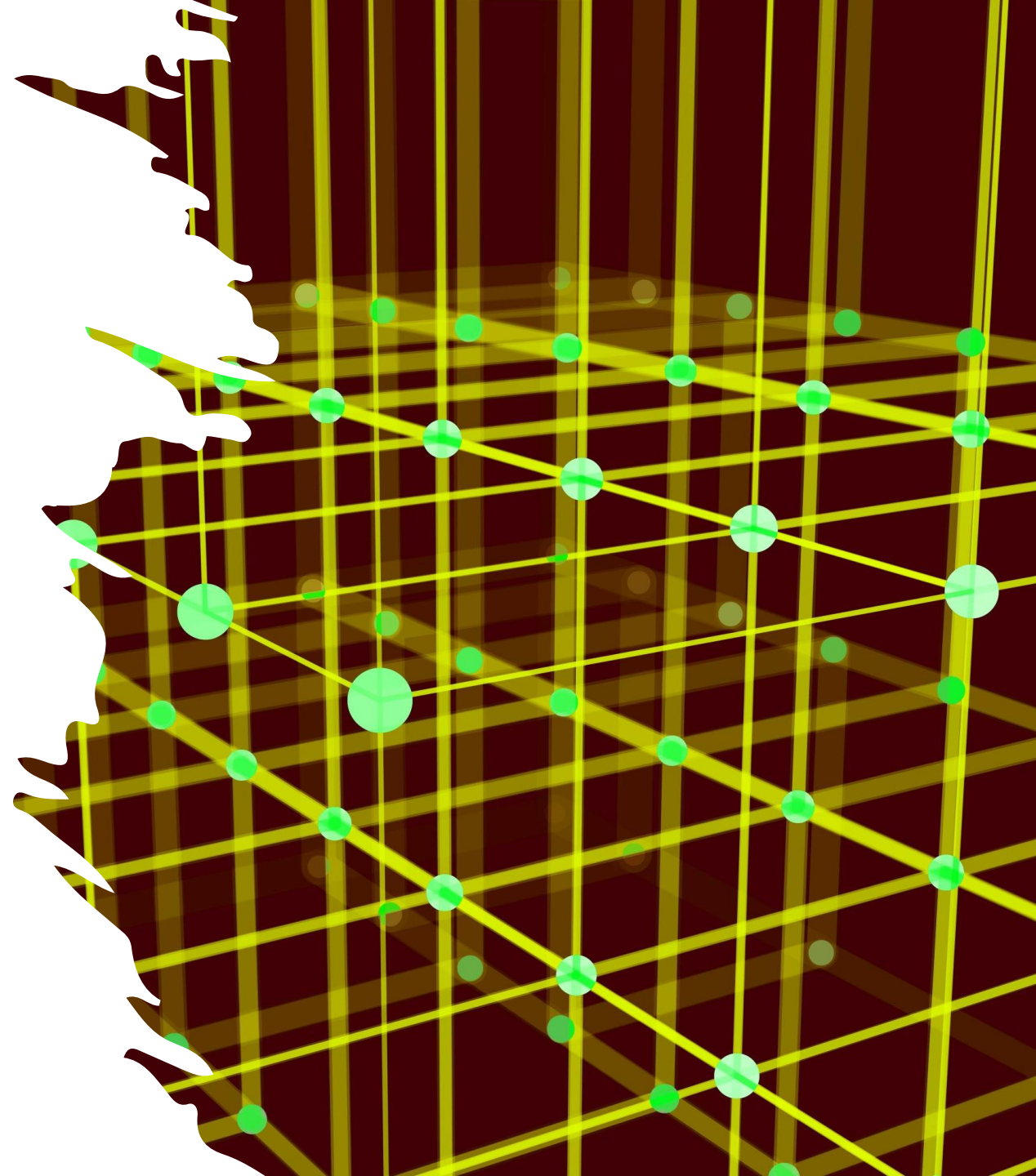


# Advanced **Neural** **Networks Design,** **Attention,** **Precision** and **building complex** **models**

AGQ Lecture4

Robert Currie



# My AGQ Lectures

- ~~Lecture 1: Embracing Python3 & Git~~

- ~~————— Intro, Intermediate Python3, Git~~

- ~~————— Getting Setup with Conda, Basics of Numpy & Pandas~~

- ~~Lecture 2: Machine Learning from the ground up~~

- ~~————— Building a simple DNN using NumPy, then the same again using PyTorch.~~

- ~~————— Constructing a Simple Classifier using PyTorch.~~

- ~~Lecture 3: More complex Machine Learning models~~

- ~~————— Building more complex graphs using PyTorch.~~

- ~~————— Building an AutoEncoder using PyTorch.~~

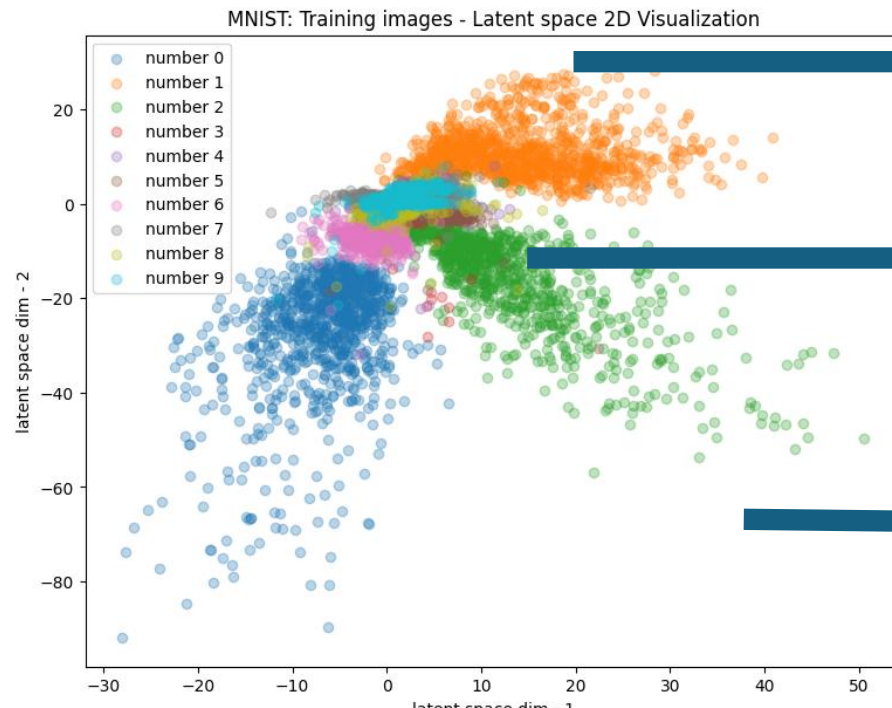
- **Lecture 4: Modern Machine Learning Concepts**

- DNN, CNN, Attention and model Precision.**

- Building a 1-bit precision Classifier using PyTorch**

# 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](https://en.wikipedia.org/wiki/Here_be_dragons)

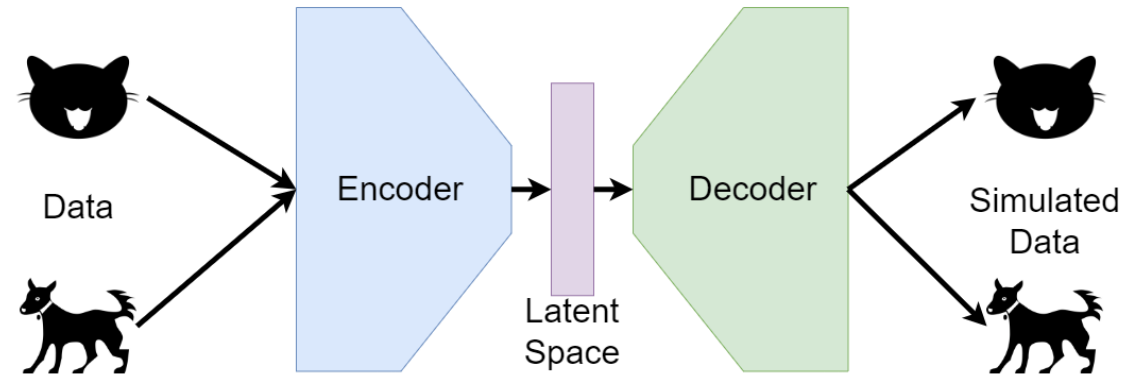
# Generative Adversarial Neural networks

- **GAN** models are a type of ML model which are better suited to 2 of the shortcomings with **AE/VAE**:
  - Generate better simulated data  
(**AE** results are often low-precision, **VAE** not “*realistic*”)
  - Able to better generate “unseen” entries indistinguishable from real data
- They also tend to:
  - Generate simulated data with finer details
  - Generate more stable image generation over wider range of inputs

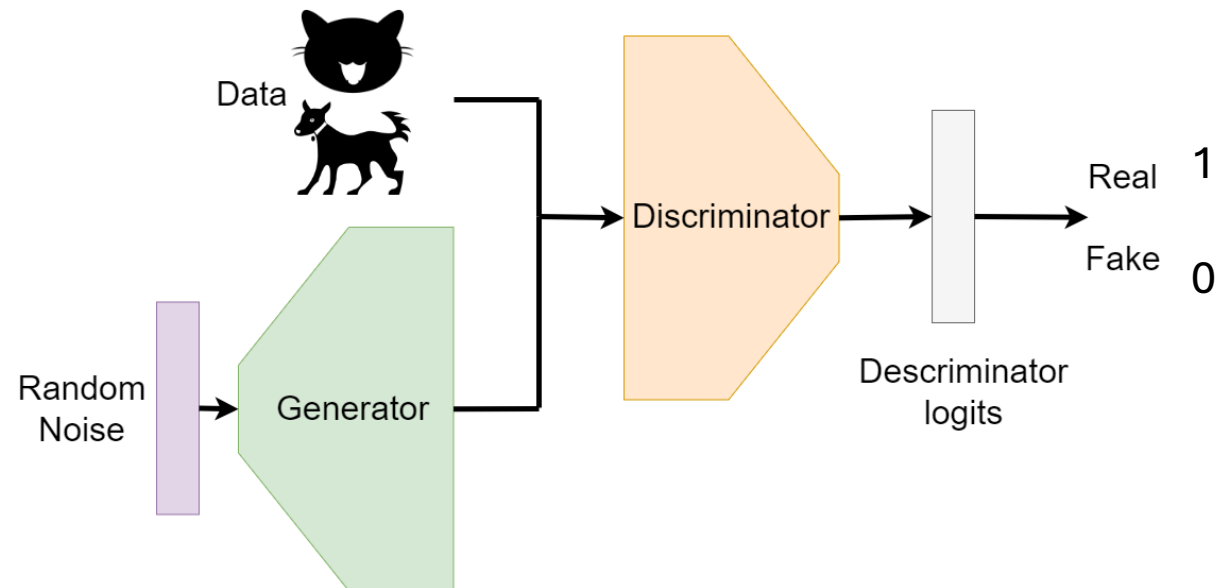
# AE vs GAN design

- 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* in 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 (*NB: DNN Classifiers*):

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

1-bit Classifier:  
*True/False*

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

# GAN loss function

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

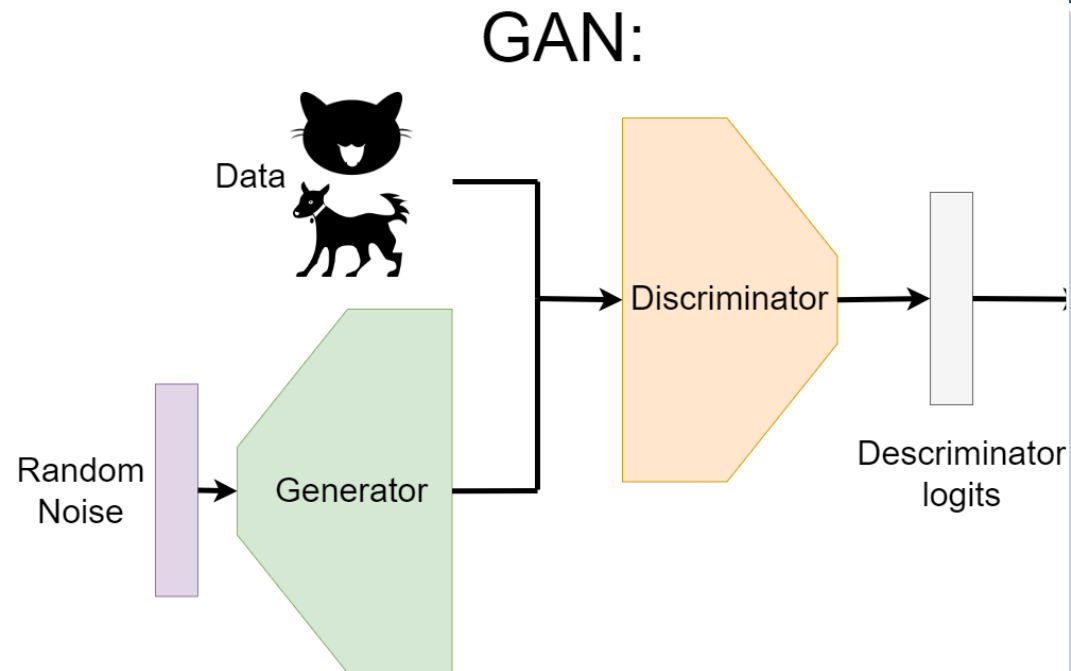
$$\min_{\mathbf{G}} \max_{\mathbf{D}} \left[ \sum_{\mathbf{x}} [\log(\mathbf{D}(\mathbf{x}))] + \sum_{\mathbf{z}} [\log(1 - \mathbf{D}(\mathbf{G}(\mathbf{z})))] \right]$$

*Real Data*

*Generation Seed*



# GAN model



## Pros

- Powerful as **anomaly detection** models
- Generated data has **higher precision**
- Well suited to **data augmentation** tasks
- Less free parameters than equivalently complex DNN

## Cons

- **Model collapse** can lead to generated data failing to fully match whole phase-space
- **Training instability** in GAN can cause the models to rapidly diverge or mis-converge
- Potentially **computationally** more **expensive** than other comparably complex models
- 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 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.

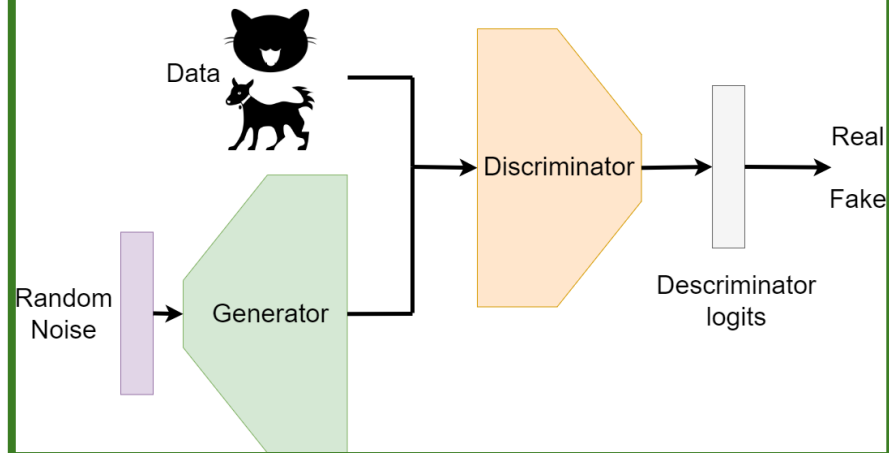
These models incorporate *tokens*, or *labels* into the model during training.

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

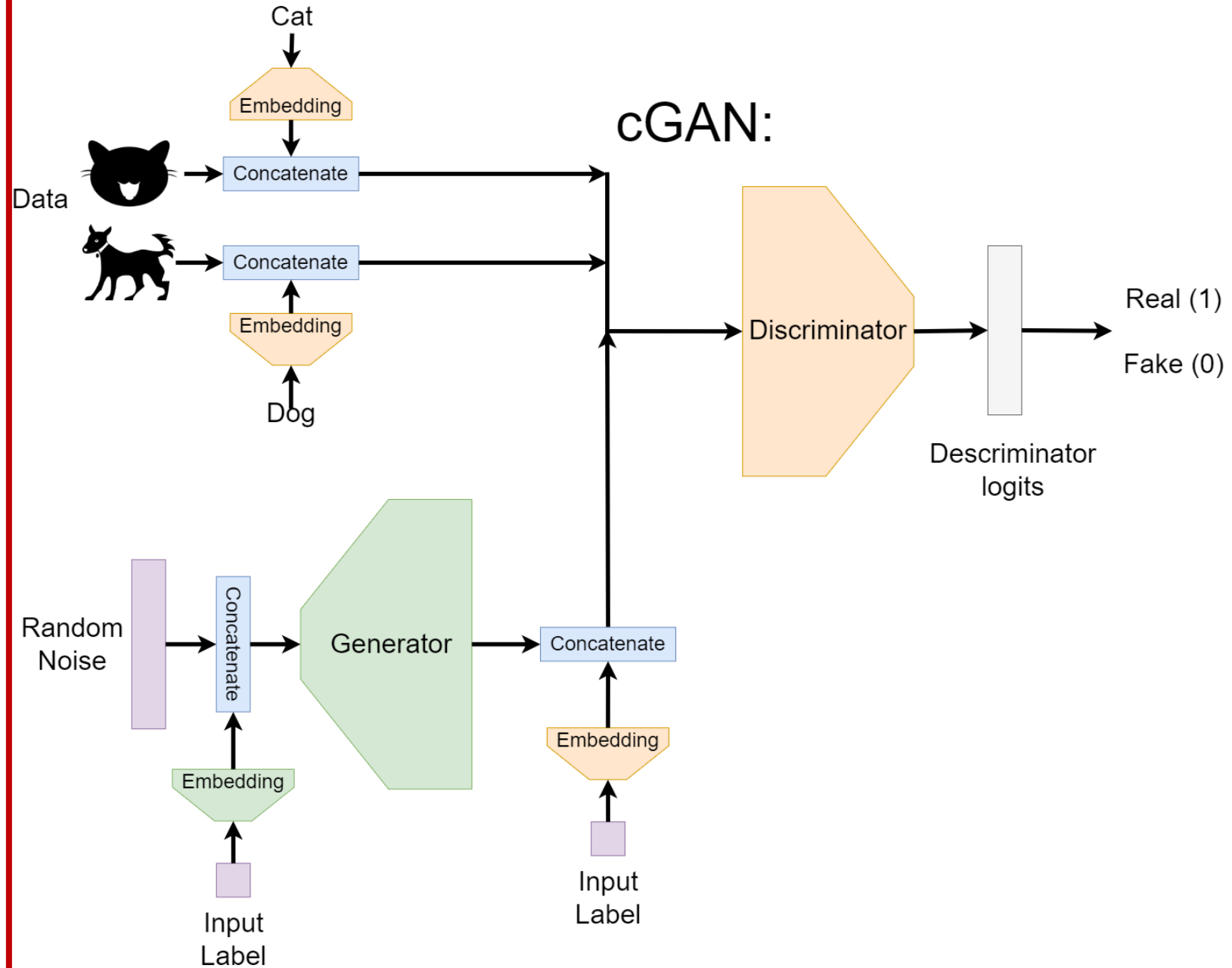
One of the most common tools for this is referred to as **Embedding**.

# GAN vs cGAN

GAN:

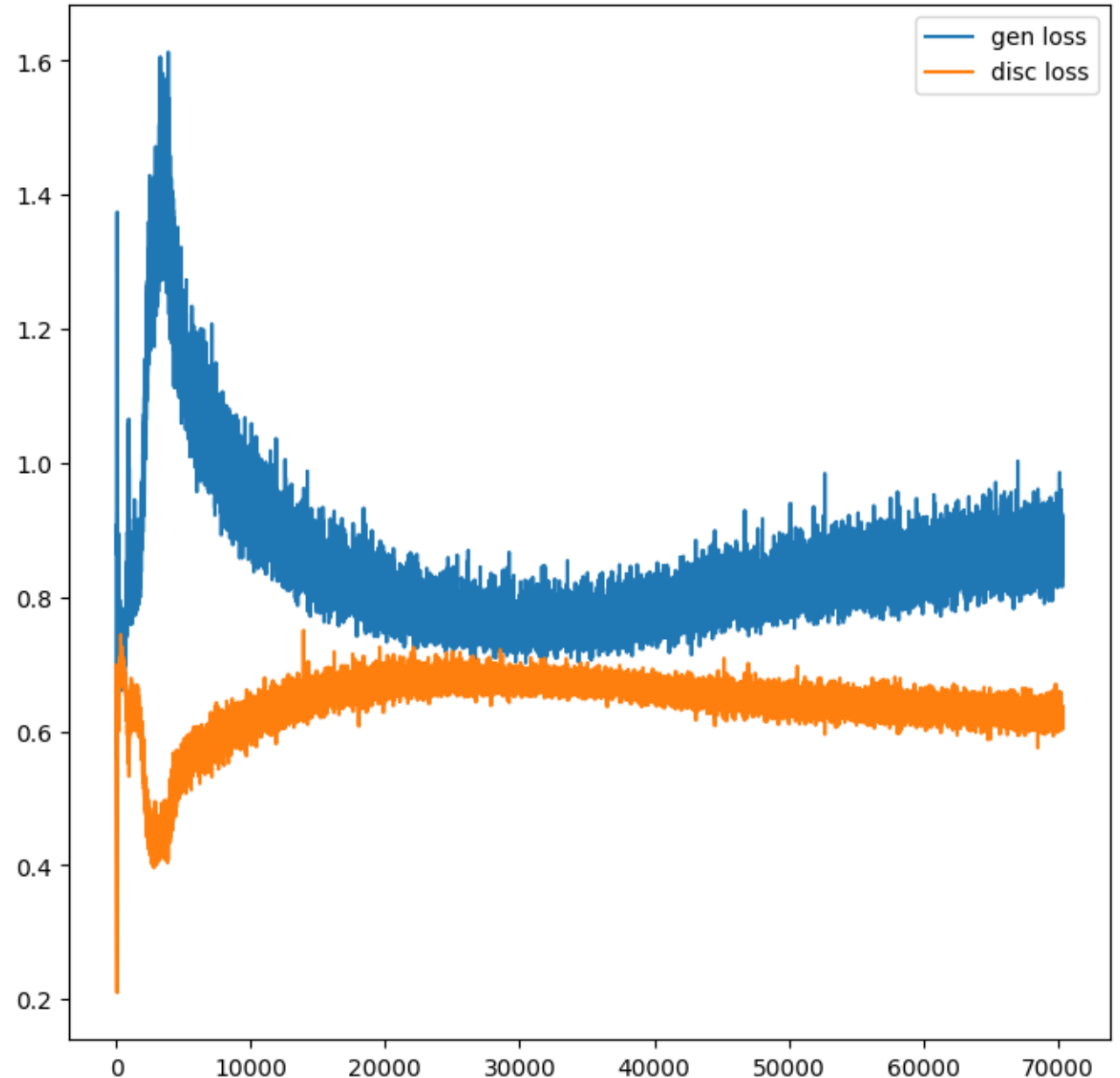


cGAN:



# 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(!).



# Diffusion models – (a very quick aside)

Simple premise;

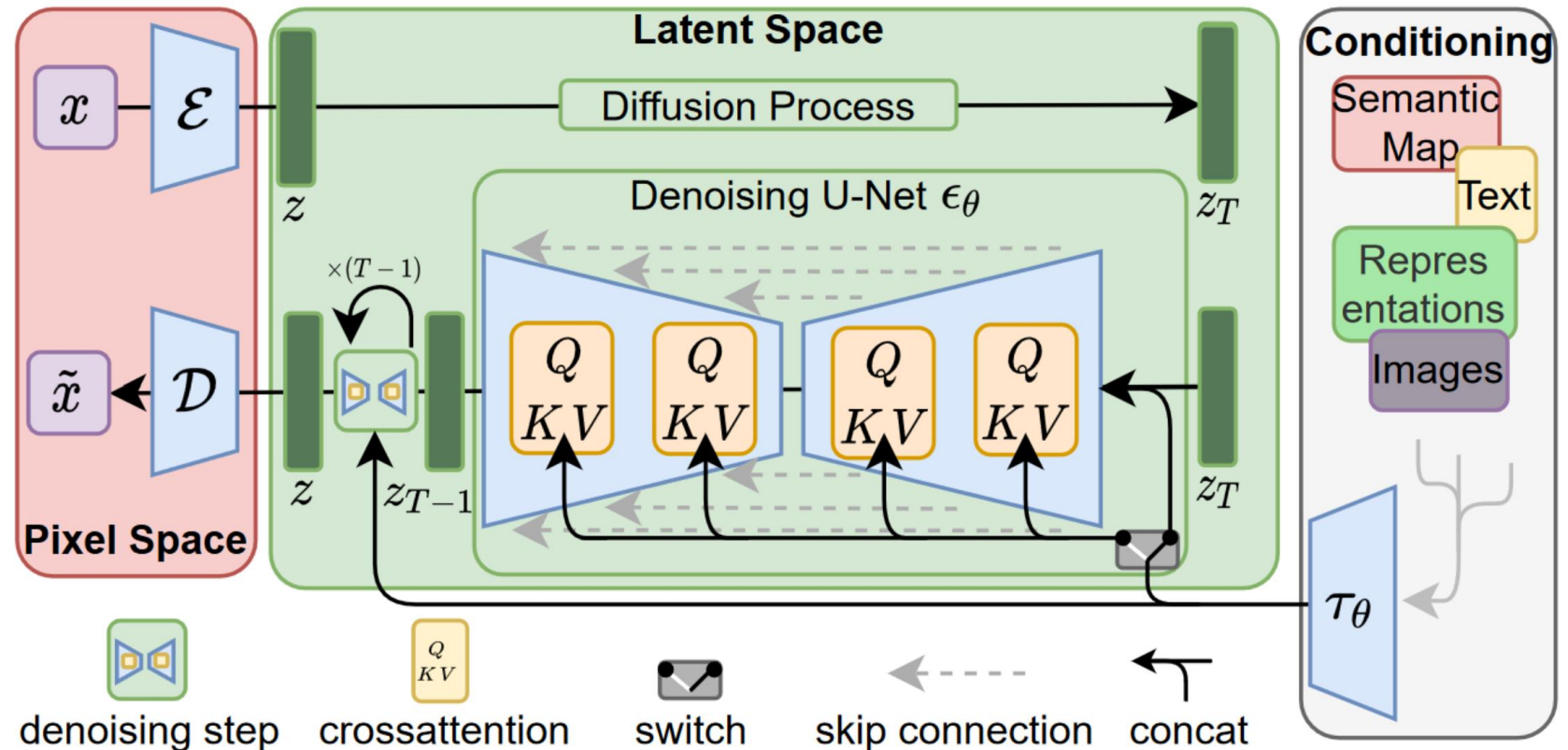
First publication: <https://arxiv.org/pdf/1503.03585>

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.

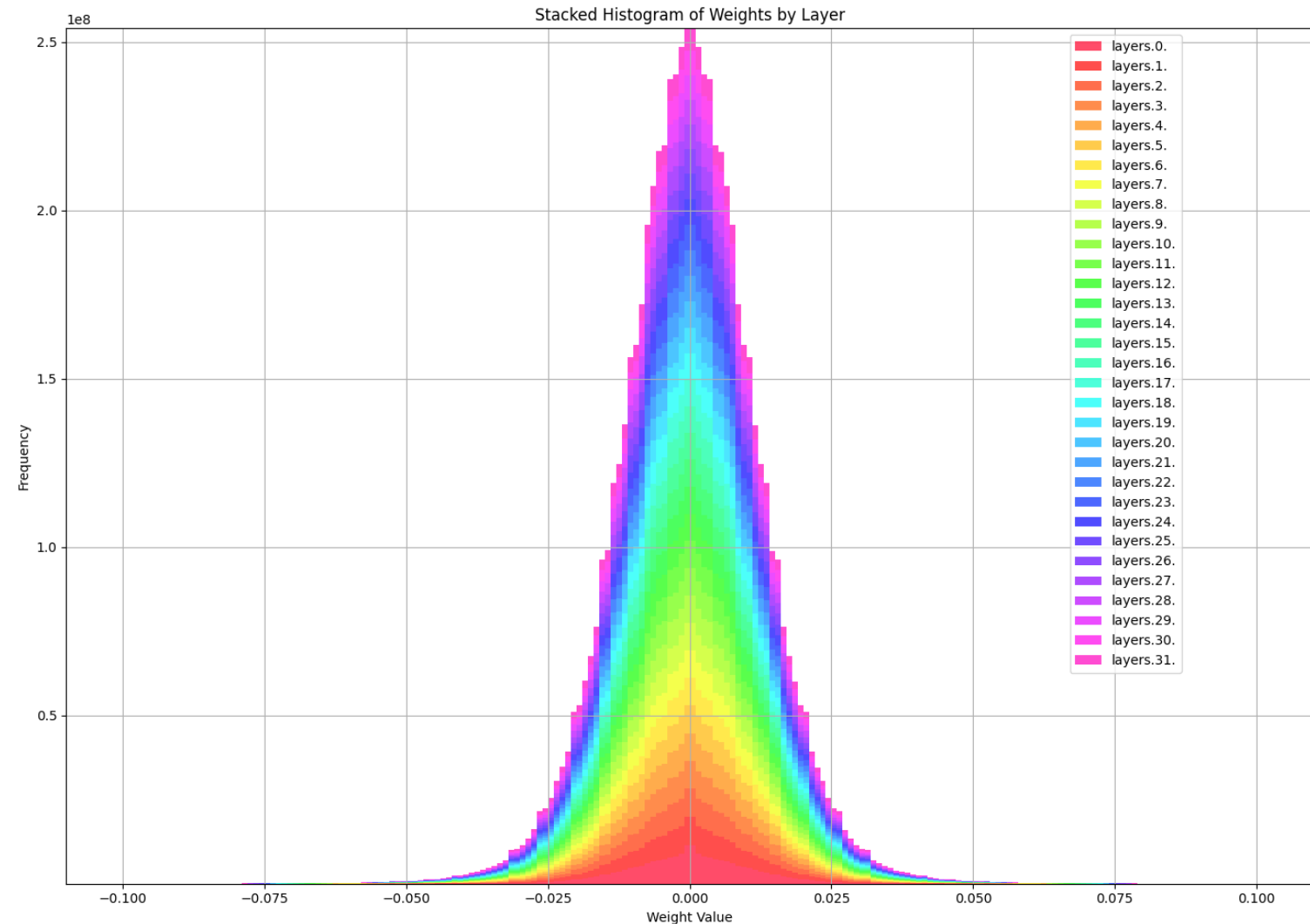
Whilst in back-propagation train the model to undo the diffusive process and to create signal out of the noise.



<https://arxiv.org/pdf/2112.10752>

# Looking at (pre-)trained models

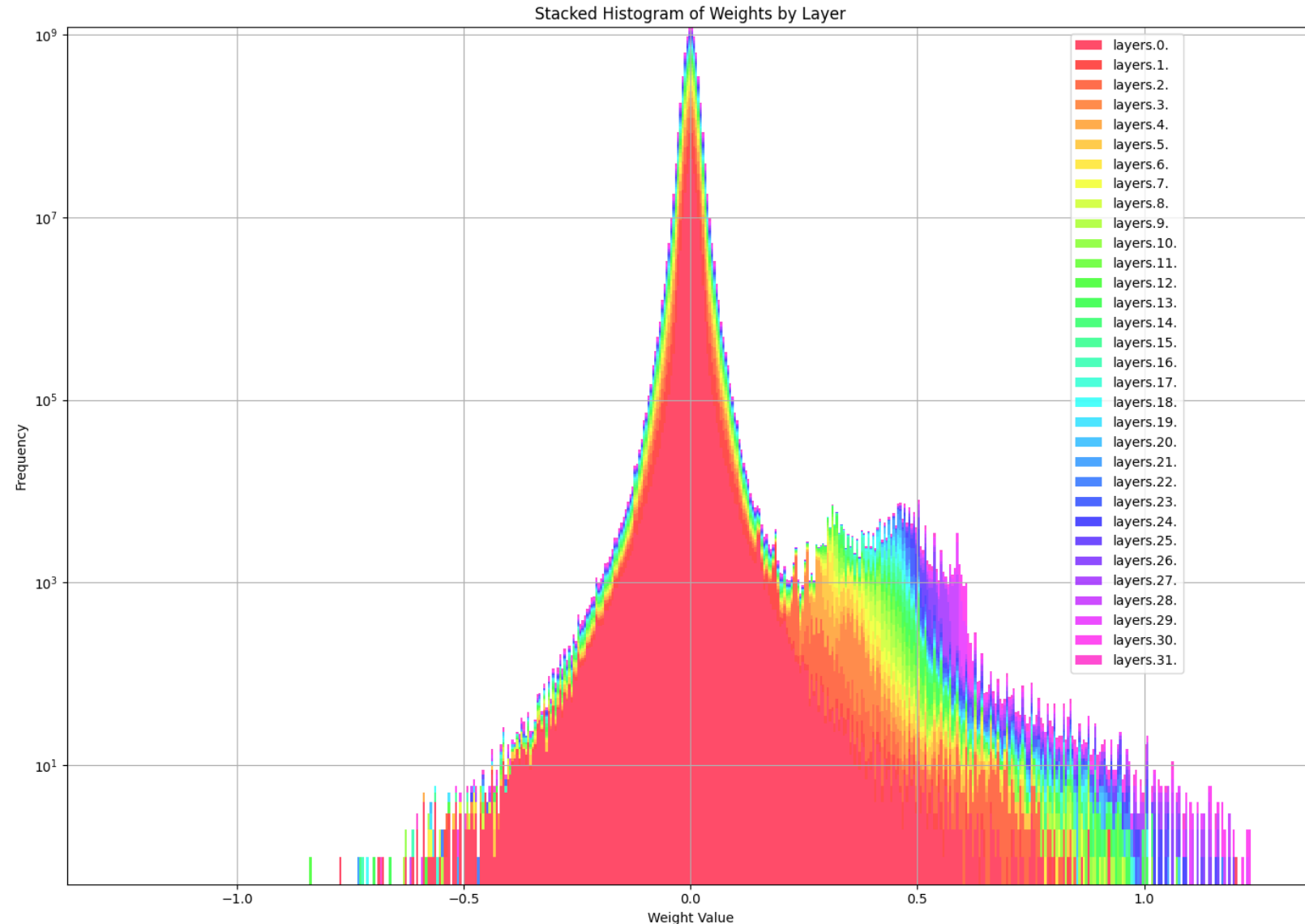
- Let's examine:  
“Meta-Llama-3-8B”
- Open model from meta  
hosted on huggingface
- 8-billion parameters
- 32 layers
- Transformer-based





# Looking at (pre-)trained models

- What do we see close-up?
- Millions of parameters which are non-centred.
- Not immediately clear why not symmetrical?
- Is this due to:
  - Structure related?
  - Fine-Tuning?
  - Model safety?
  - Something else?



# Model Optimization

- Training advanced large models is slow.
- Most of the time spent training is mostly in shuffling data around for massive matrix multiplications.
- How can we optimize or speed this up?
- **Higher level languages -> lower-level languages**  
(Fewer instructions to complete)
- **Generic code -> Accelerator specific code**  
(More instructions per cycle)
- **Reducing precision -> Less time per operation**  
(More instructions per cycle)
- **Advanced Caching/Compression -> Rewriting the model itself**  
(Fewer instructions to compute)

Easy

Difficult

Hard

# Computing Throughput and Precision

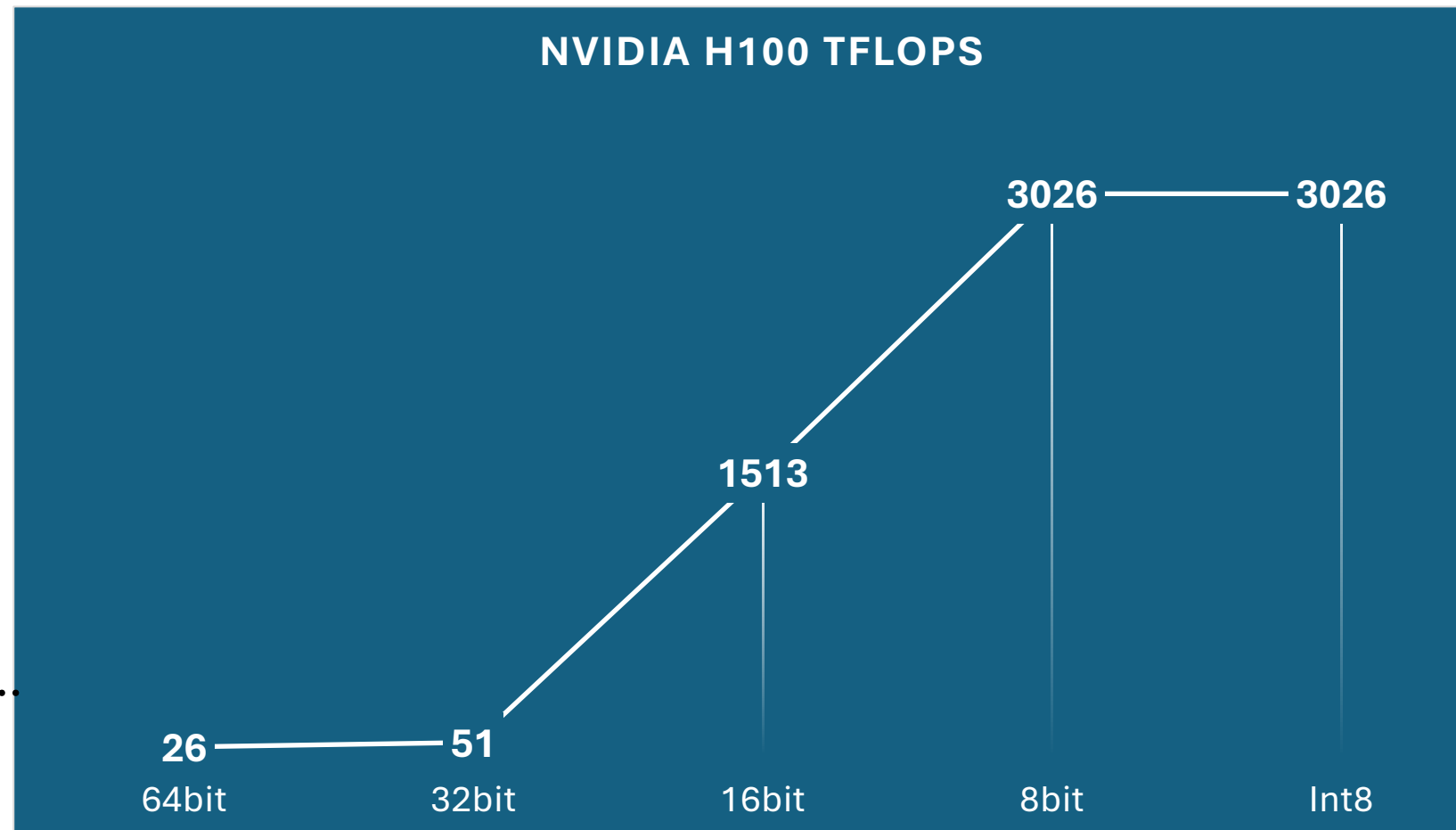
Intel i9 can perform 0.3TFLOPs @ 64bit. Best in class CPU.

GPUs can perform better.

nVidia arguably best in class.

1,500x faster by reducing precision.

Reducing precision reduces, cost, power, thermal waste, ...



# Computing Throughput and Precision

- Why can't we “just” use **8bit** models?
- Training models using only lower precision is more unstable.
- Production models may not be well-suited to using **FP8**.  
e.g. GAN accuracy can degrade rapidly with reduced precision.
- Larger models help restore accuracy of lower precision but are much more difficult to train.

# Computing Throughput and Precision

- Modern approach to combat this is to use:

**“Quantization Aware Training”**

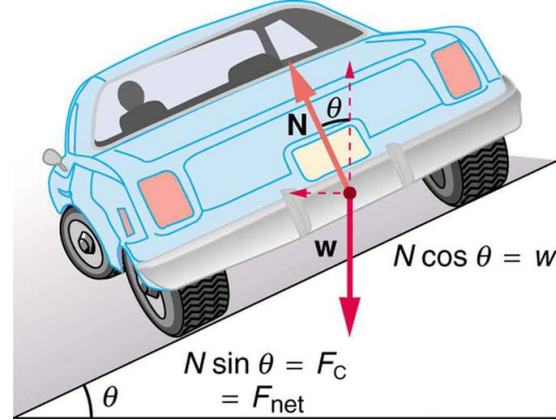
- This is when a full-precision model is trained with extra quantization aware components which gives enhanced stability.
- This also means that models need to be **“Quantized”** after training so that they can run in production.

# Computing Throughput and Precision

- **QAT** models are often good in production where resources are limited.
- However, there's no “*free lunch*”.
- Typically, these models have to be much wider in order to still store the same amount of information through training.
- This can impact their *stability*...

# Training Stability

- Training Stability is a complex topic with no “silver bullet” to solve:

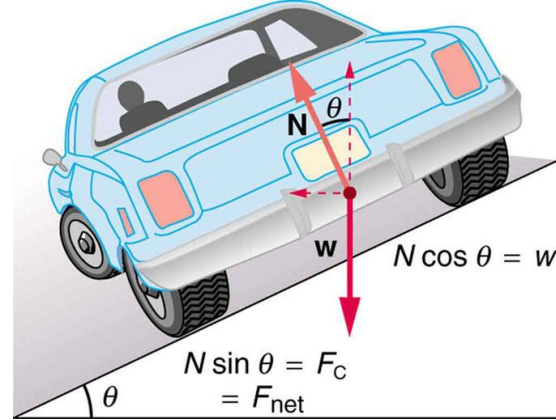


[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

1. *Dropout*
2. *Regularization*
3. *Learning Rate Scheduling*
4. *Batch Normalization*
5. *Gradient Clipping*
6. *Data Augmentation*
7. *Early Stopping*
8. *Weight Initialization*
9. *Optimizer Selection & Tuning*
10. *Gradient Accumulation*
11. *Smoothing Model Weights*
12. *Avoiding Dead Neurons*

# Training Stability

- Training Stability is a complex topic with no “silver bullet” to solve:



[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

## 1. Dropout

Extra layers dropping connections

## 2. Regularization

Normalize model parameters

## 3. Learning Rate Scheduling

Adapting the LR during training

## 4. Batch Normalization

Normalize the input data

## 5. Gradient Clipping

Reduce range of back-propagation

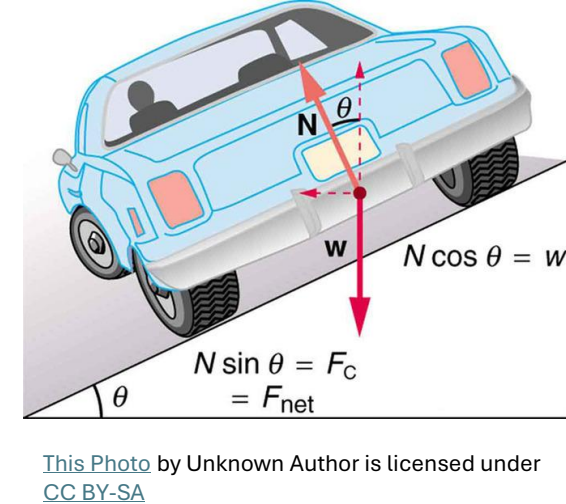
## 6. Data Augmentation

Use real and generated data



# Training Stability

- Training Stability is a complex topic with no “silver bullet” to solve:



**7. Early Stopping**

Stop training on condition

**8. Weight Initialization**

Start from a better place in fit

**9. Optimizer Selection & Tuning**

Tune Hyper-Parameters

**10. Gradient Accumulation**

Smoother Averages

**11. Smoothing Model Weights**

Avoid oscillating around a minima

**12. Avoiding Dead Neurons**

Improve use of whole model

# Model Stability

- Lots of the steps in model training rely on **computationally random** sources.

The “**random**” initialization values.

Which nodes are “**randomly**” dropped out.

“**Random**” noise added to supplement the input data in training.

- This means we can ‘**game the system**’:

*Picking the seed corresponding to the best version of this model isn’t “bad”.*

- If you build a classifier and train it from a fixed seed, then adjust it and re-train you will see different final accuracies(!).

# Model Stability

- The “*correct*” approach would be to combine these individual models in some way to get the “*best average model*”.
- This is difficult as these models are all black boxes.
- The **spread** for a *numerically 64-bit classifier* trained on the mnist dataset can be quickly shown to be of the order of **1%** (maybe slightly higher).
- The **spread** of *numerically 1-bit classifier* trained for the same task is much higher under the same conditions, **~10%** (!).

# All you need is **Attention**(!)

- Probably one of the most disruptive concepts in ML in the last 10yr or so has been **Attention**.
- Before we get there there's something we want to consider;
- How do ML models learn, and store relationships from data?
- *Let's have a quick review...*

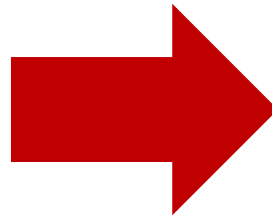
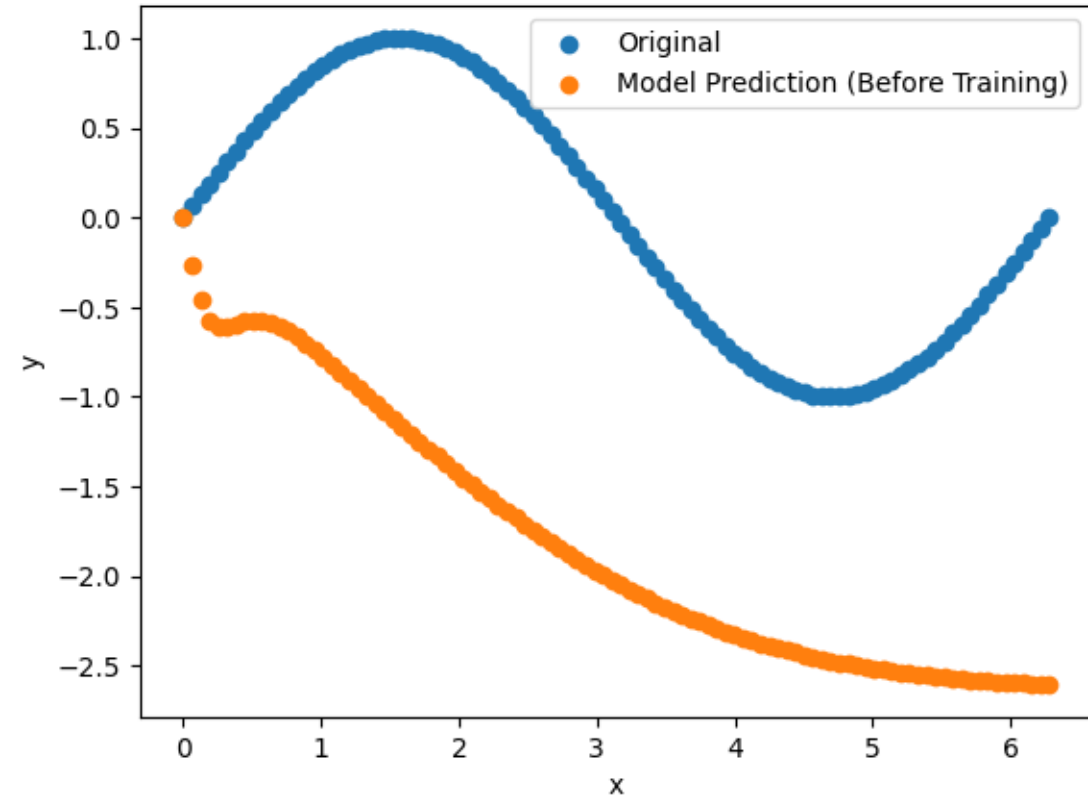
# CNN/DNN Model Test

- *Time for a ‘simple’ totally non-scientific test.*
- **What happens if I train a DNN on a sinusoid waveform?**
- Input data: “clean” single sinusoid. 100% signal.
- Model will be trained until it has converged, or hits 1k iterations over the whole dataset.
- Trained model will be used to predict data:
  - a) within the input window of the training data
  - b) then extrapolated to what will happen “*in the future*”...
- Caveat: *Data will be input into the model as best format for a given model type.*

# CNN/DNN Model Test

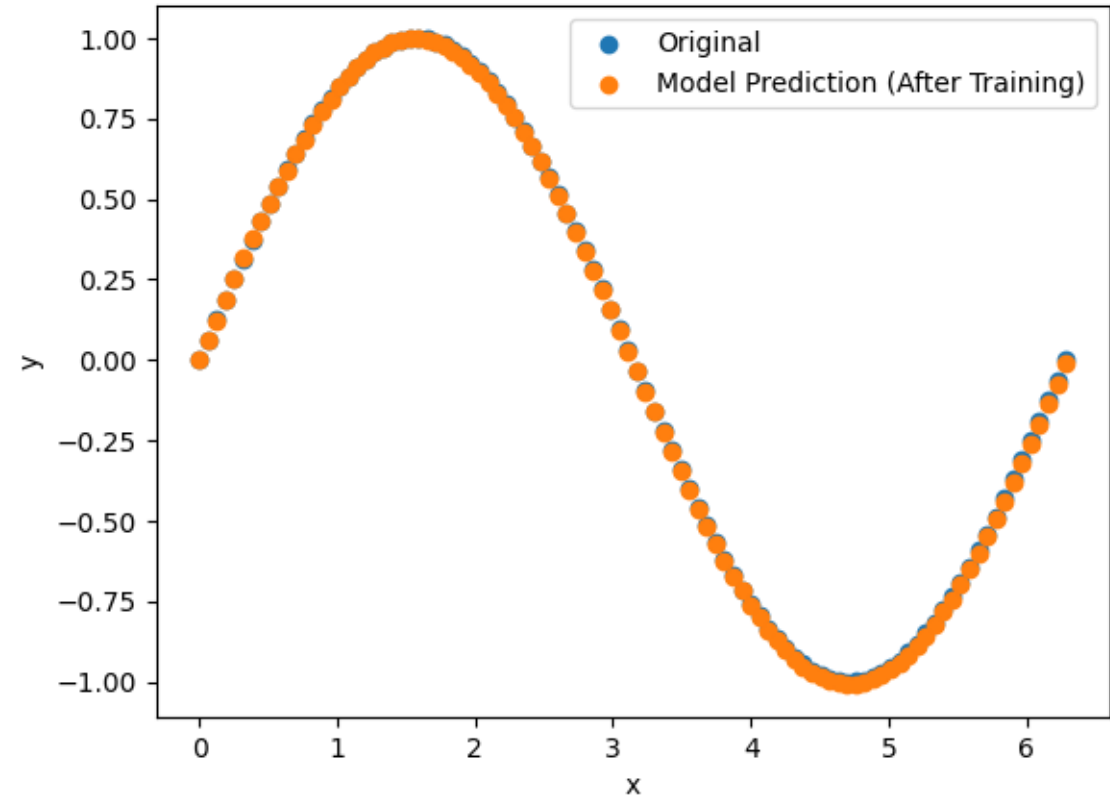
**Before:**

DNN Predictions vs Original Data (Before Training)



**After:**

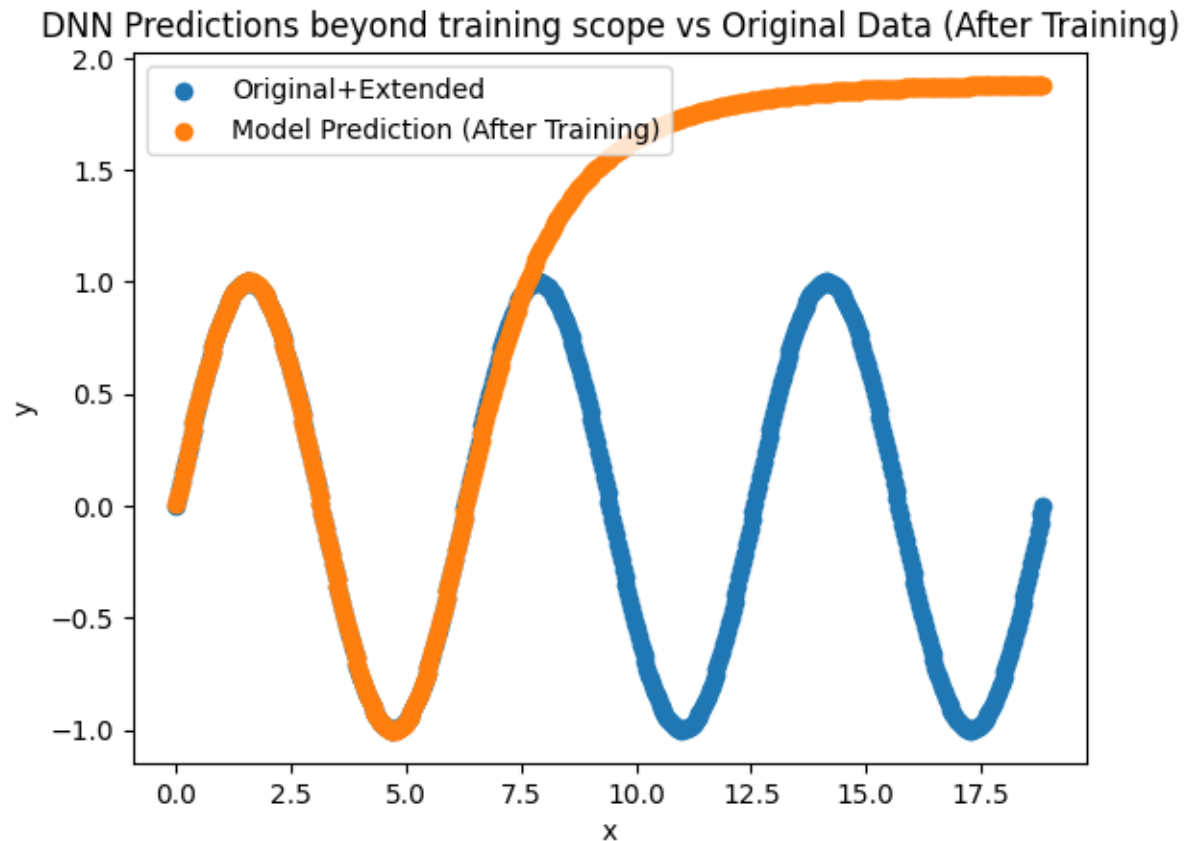
DNN Predictions vs Original Data (After Training)



# CNN/DNN Model Test

- Networks aren't designed to take ordered input(s)

- Output from model will almost certainly not predict patterns or time-based behaviours
- Models are good for identifying if a random anomaly appears compared to training dataset.



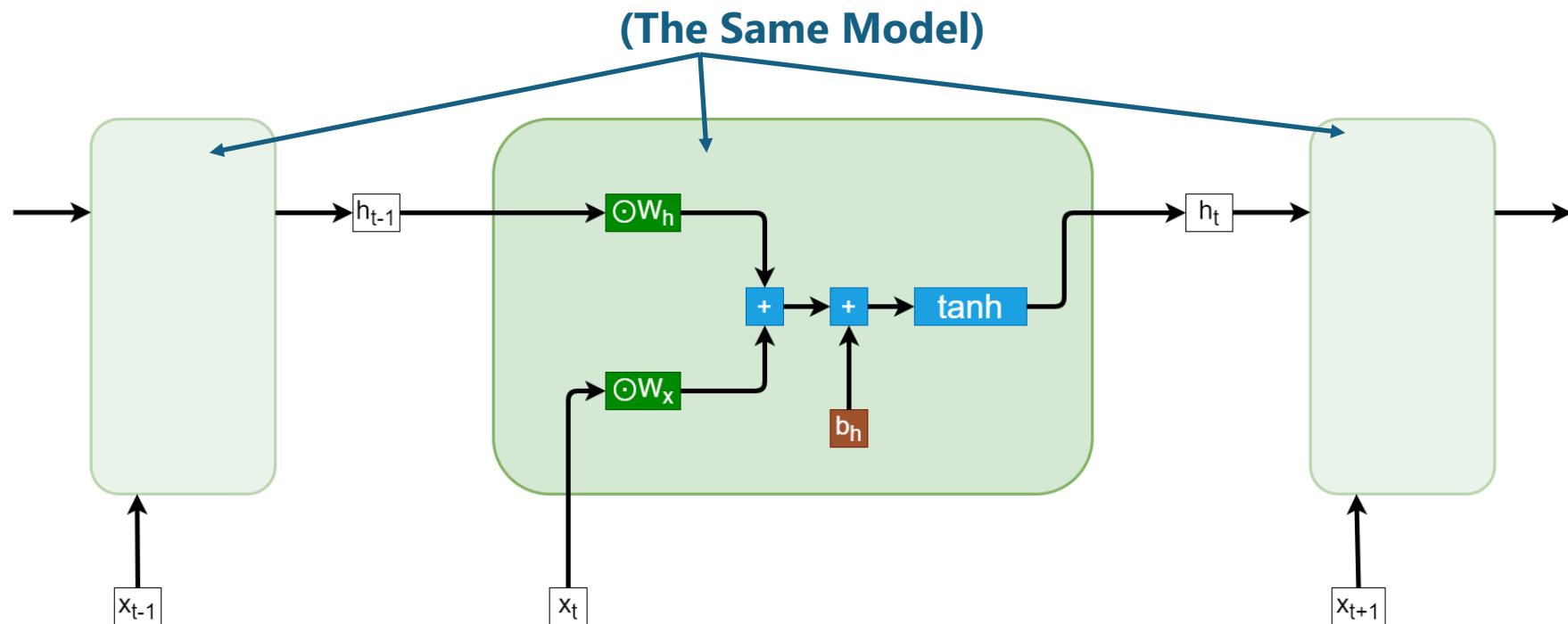
# RNN/LSTM to save the day(?)

- **CNN/DNN** models have no concept of relational ordering between inputs.
- If we modify our model to train over ordered series of inputs the model should learn these relationships.
- This requires expanding batches of data to be batches of sub-series of ordered data.



# RNN model

- Pieces of **RNN** are the same that we've seen before, only the model is evaluated over a series of  $n$  datapoints to learn relationships

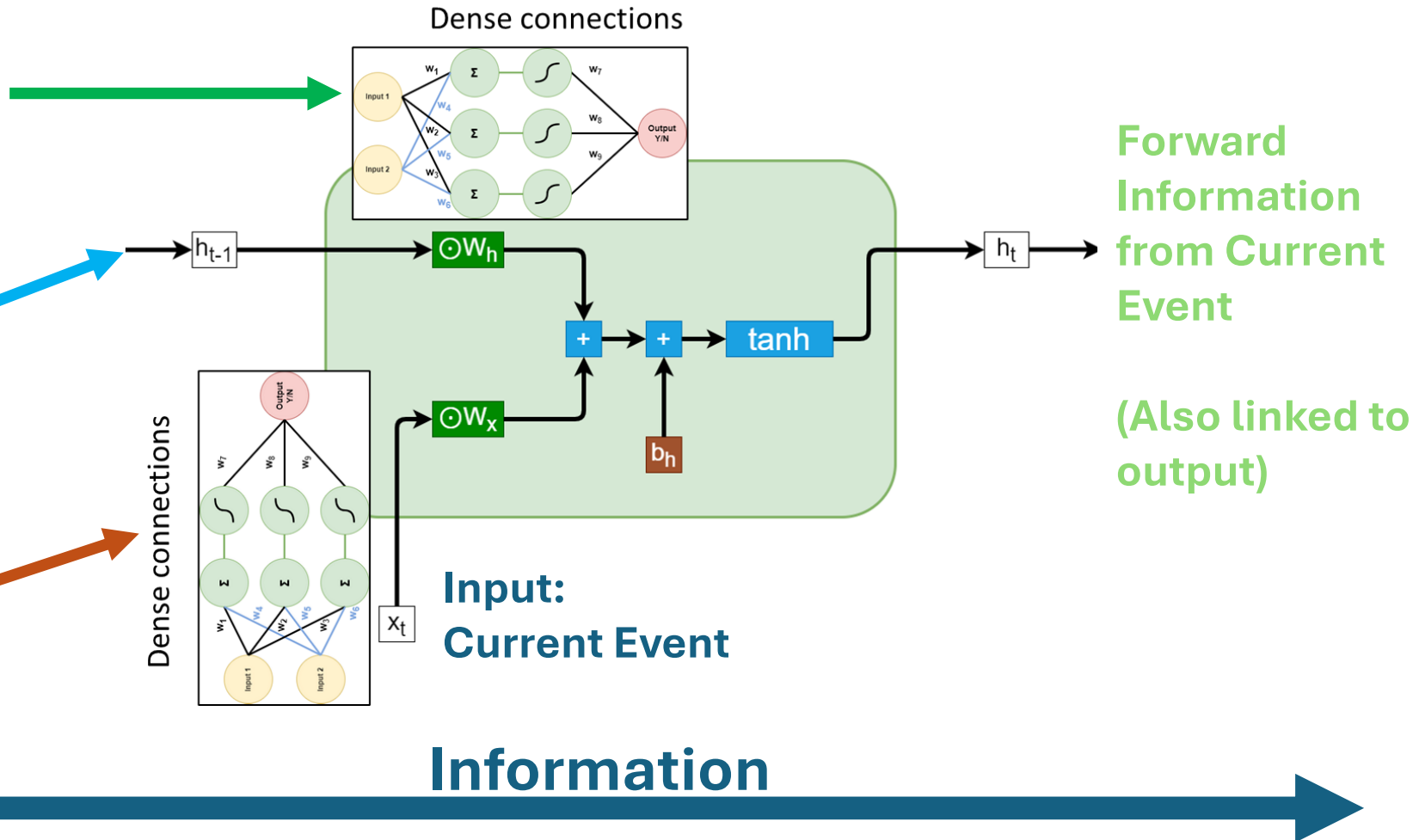


# ML Evolution & Building Blocks – RNN (2)

Learned relation(s)  
between this event  
and prior

Decision from  
Previous  
Event

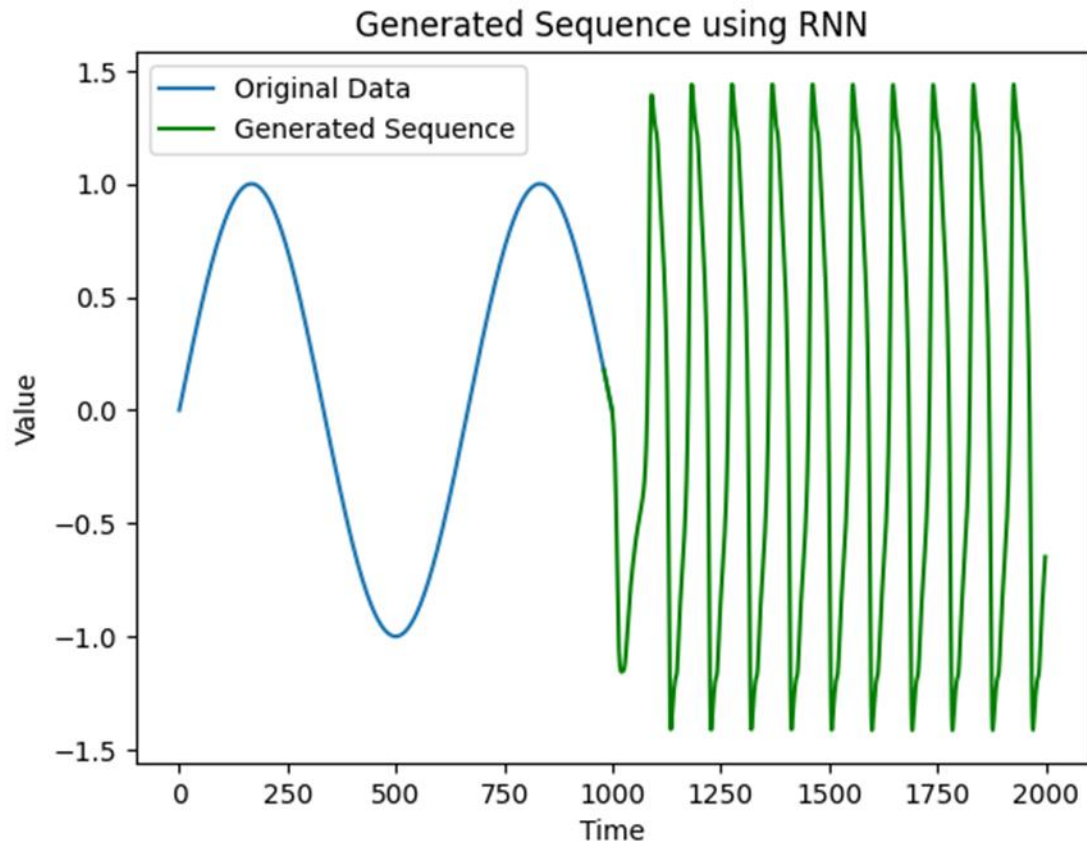
Learning Based  
on Input



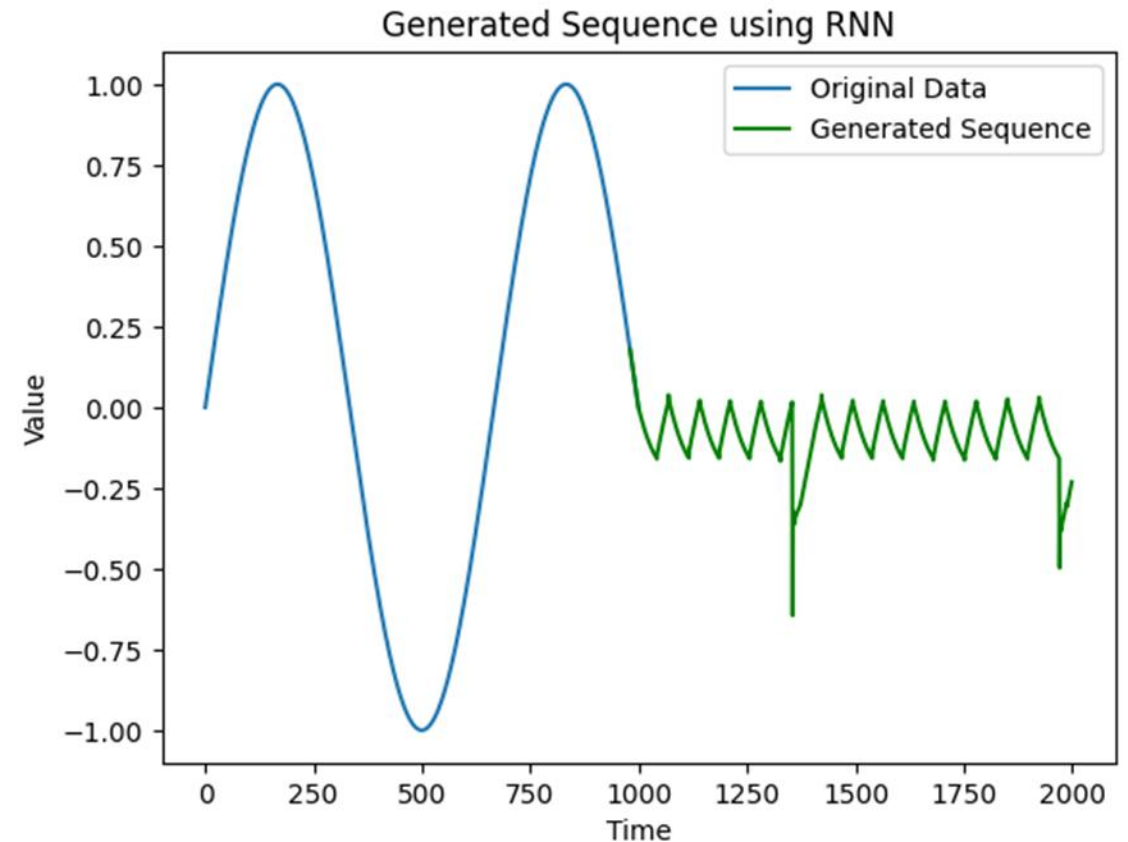
# RNN model (2)

- The results of training an **RNN** can be un-stable...

Seed: 42



Seed: 420



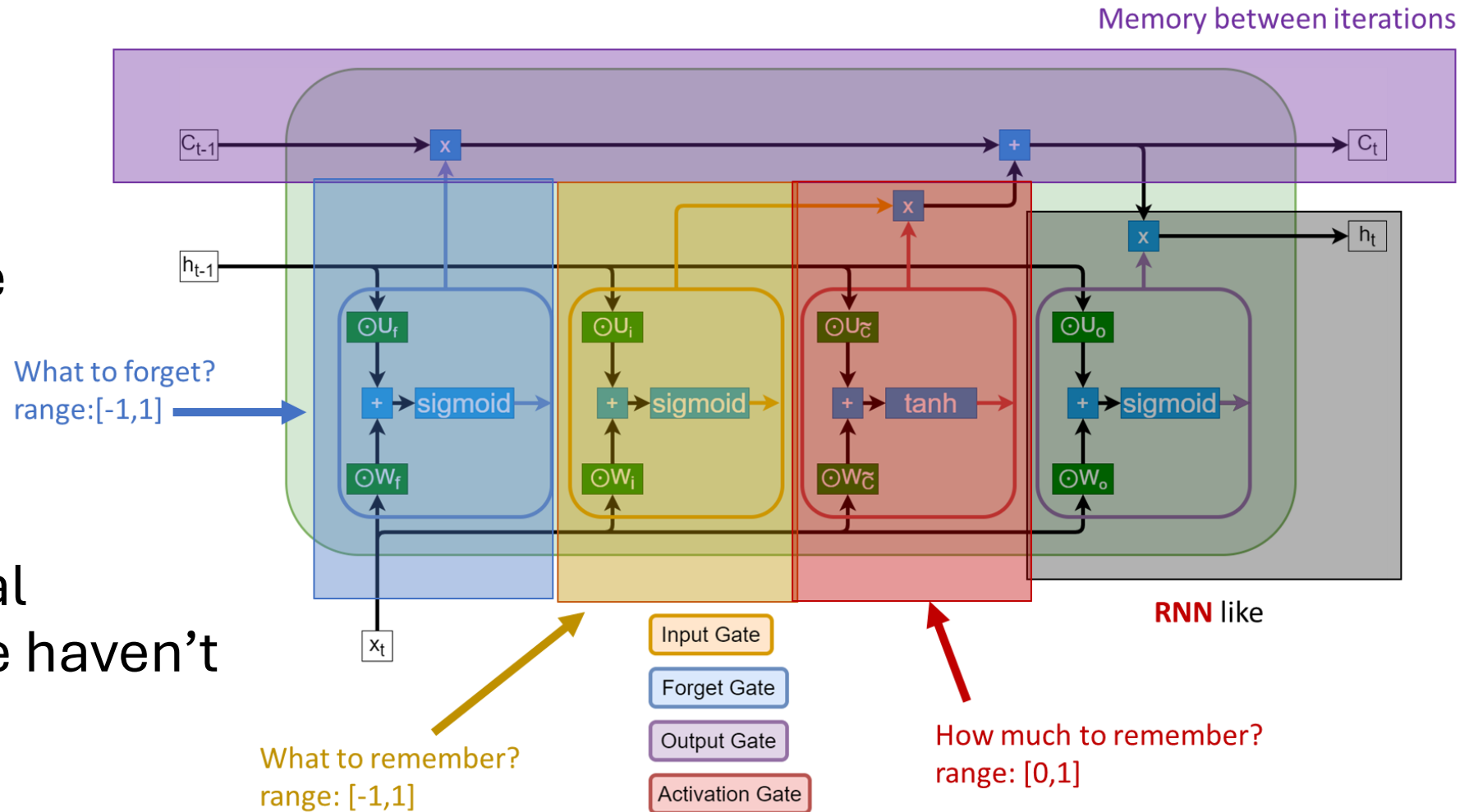
# LSTM models

- **LSTM** models introduce a vague concept of “*persistent memory*”.
- These models are able to learn some global state which contains the relationship between datapoints in a series.
- The flow of information between the models internal state and the parts of the model which train on input data are referred to as “gates”.

# LSTM models

- **LSTM** can be difficult to implement

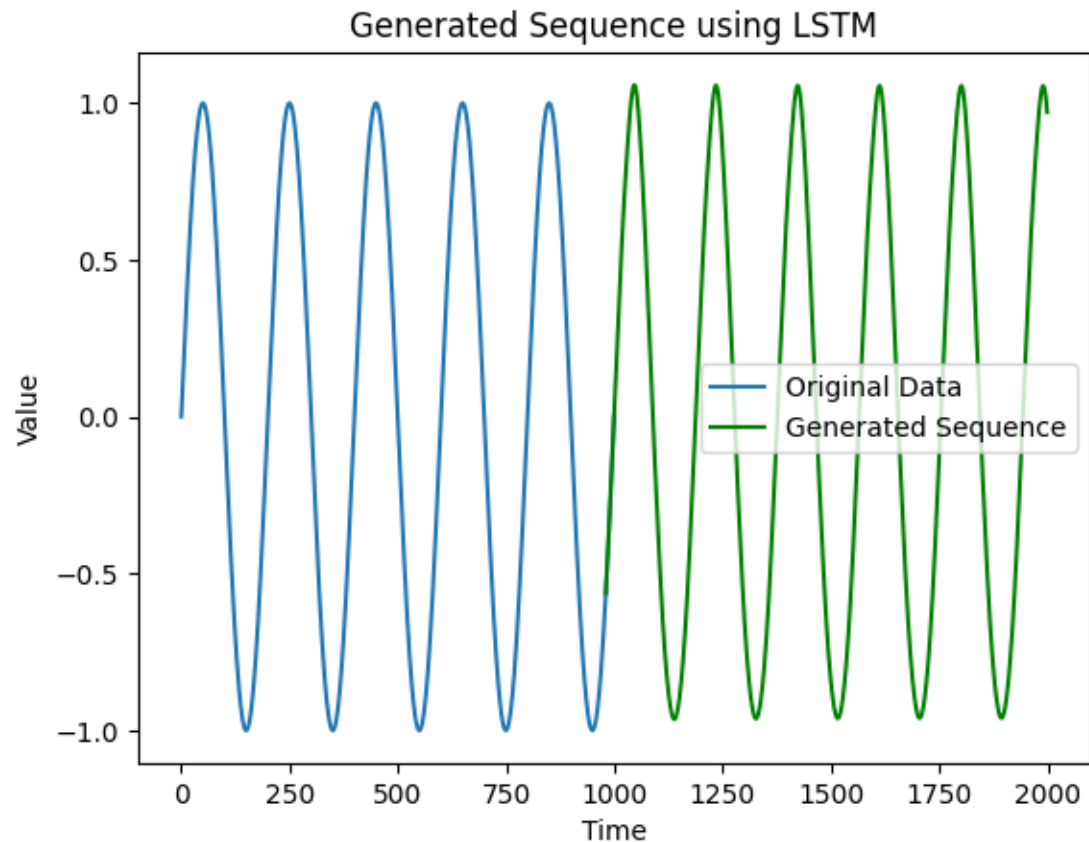
- Again, no real parts that we haven't seen before



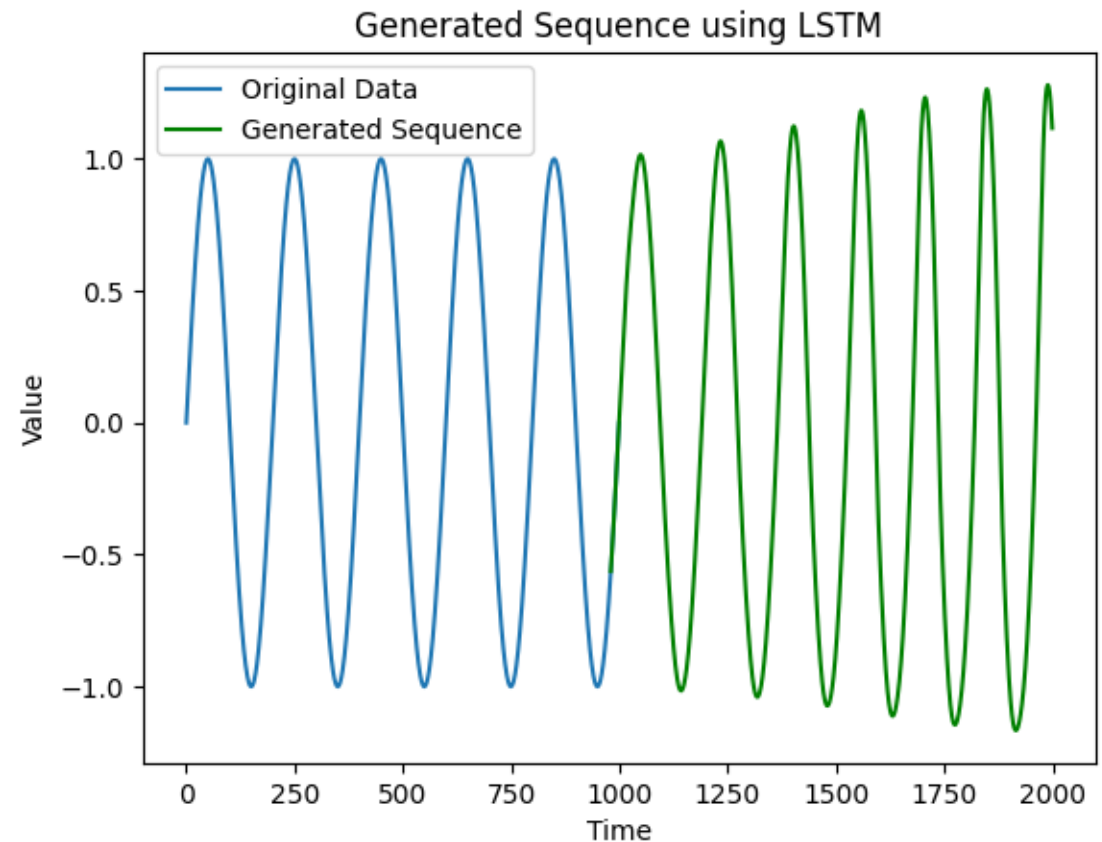
# LSTM models

- Training can be difficult, with results still slightly unstable...

**Seed: a**



**Seed: b**



# Analyzing complex relationships

- Language can be broken down into a complex set of rules governing the ordering of words
- Ideally, extract sentiment or guess what word might come next you need some model which can learn these rules or relationships and lead to a stable trained model.
- Ideally, we'd also like this model to be stable during generation too.



# Attention in Transformers

- Before we get to “**Attention** in *Transformers*”.
- What is “**Attention**”?

$$\textit{Attention} = \textit{softmax} \left( \frac{QK^T}{\sqrt{d_K}} \right) \cdot V$$





# Attention in Machine Learning

- **Query, Key, Value.** OK, that's English, but *what does it mean?*
- **Query:** What you're looking for.  
(focus of attention)
- **Key:** How well some reference matches what you're looking for.  
(relevancy score)
- **Value:** Information to retrieve from reference depending on relevancy.

# Attention in Machine Learning

- **Query, Key, Value**. OK, that's English, but *what does it mean?*

$d_k$ : Key Dimension

- E.g.: **Ranking query (web search)**
- Attention can be used to produce a ranking system.  
*For a ranking model, **Key/Value** are fixed.*

- Eg:

**Query:** “Keyword from user Search”,  
**Key:** “Keywords already pre-computed”,  
**Value:** “Indexed Web-Pages”.

**Web Query (User):**      **Ferrari**

**Keys:**      1990, blue, red, furry,  
                 mediterranean, engine, speed,  
                 leather, winter, raining

**Values (Web-Page):**      Cars of 1990s [www.cars.com](http://www.cars.com),  
                 Best Boat [www.bestboat.org](http://www.bestboat.org),  
                 Cute Kittens [www.kittens.net](http://www.kittens.net),

# Transformer **Attention** as a Concept

- It's worth looking at **Attention** to understand what is going on within these models.

$$\text{Attention} = \text{softmax} \left( \frac{QK^T}{\sqrt{d_K}} \right) \cdot V$$

- At a high-level we have:

**Attention** = "*how alike  $Q$  and  $K$  are*". "*input values*"

- This means that ignoring a lot of implementation details;

every **Attention** layer contains residual “*skip-like*” connection and a “*mask-like*” filter to control the flow of information through the model.

# Attention in Machine Learning

- OK, I can extract **Key** and **Value** from data.
- With a bit of imagination, I can make an algorithm to map “***user queries***” to “**Query**” and that gives me a search system which can find results in ranked web-pages.
- This is in a “*hand-wavy*” way how Google, search engines and complex fuzzy database queries work.

# Attention in Machine Learning

- So how do we extract this information from our data?
1. Build a billion-dollar company.  
Hire the worlds best computing experts.  
Spend millions each year.  
Keep updating refining many algorithms.
  2. Get the Machine Learning model to do it for us.

This is called “**Self-Attention**”

Aka, map the **input data** to **Q**, **K** & **V** and have the model learn the relationships itself.

# Transformers

- Now have all the pieces to build the transformer model. Famously introduced by Google in 2014
- This model uses 3 very new and key advances:
  1. **Self-Attention**  
Extracting **Q**, **K** & **V** directly from input
  2. **Feed-Forward masks**  
Only the past is considered when generating
  3. **Multi-Headed Attention**  
Multiple attention calculations in parallel

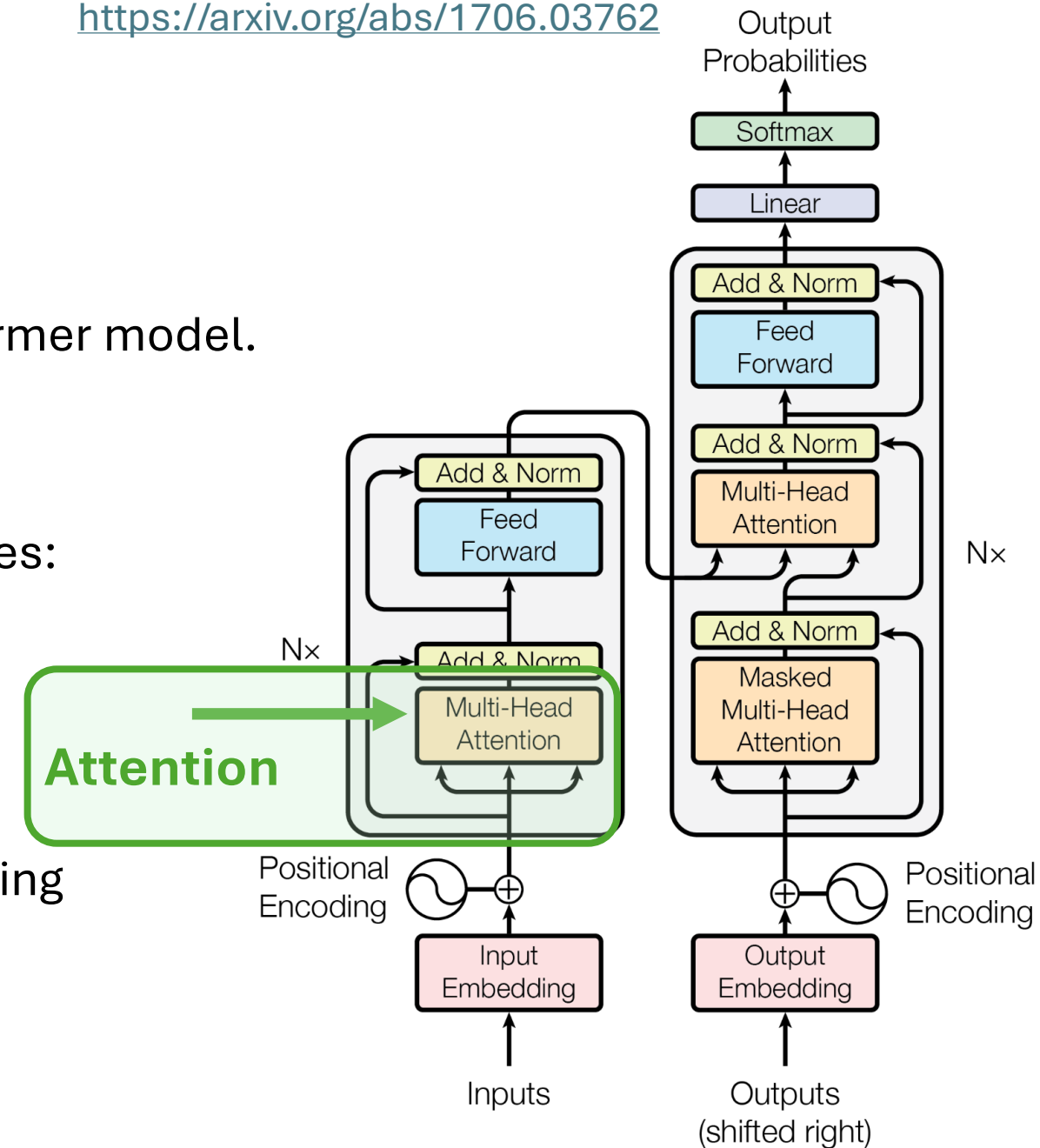


Figure 1: The Transformer - model architecture.

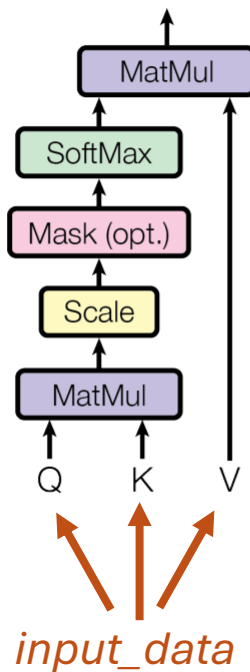
# Transformers

- **Self-Attention** means we can extract relationships from our data.
- **Feed-Forward masks** help improve the model stability.
- **Multi-Headed Attention** means we can extract short- and long-range relationships in parallel easier.
- The graph of **MHFFA** shows this approach allows more information to flow through the model.

# Attention as a Graph

- Input data is “*projected*” into the model.
- Like embedding, but **preserves positional ordering of input.**

Scaled Dot-Product Attention



Multi-Head Attention

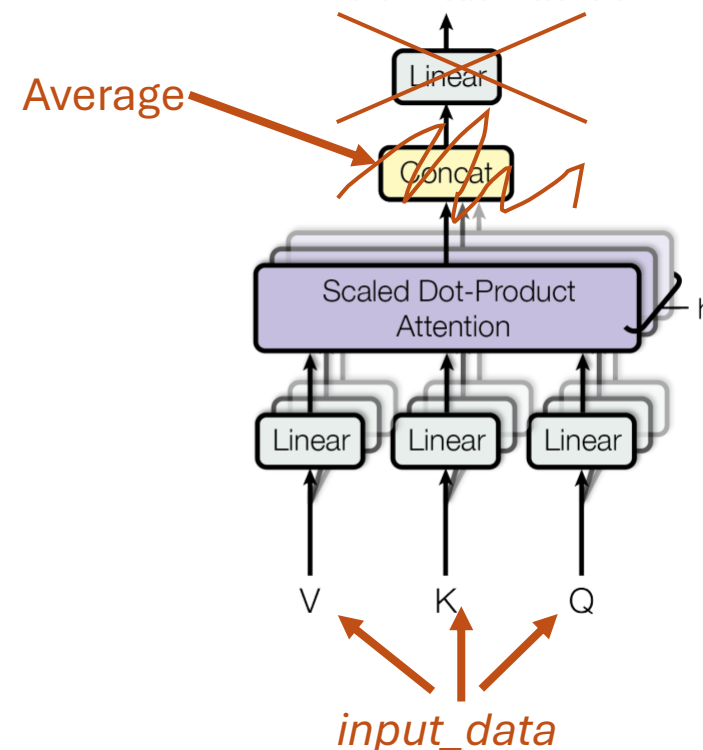


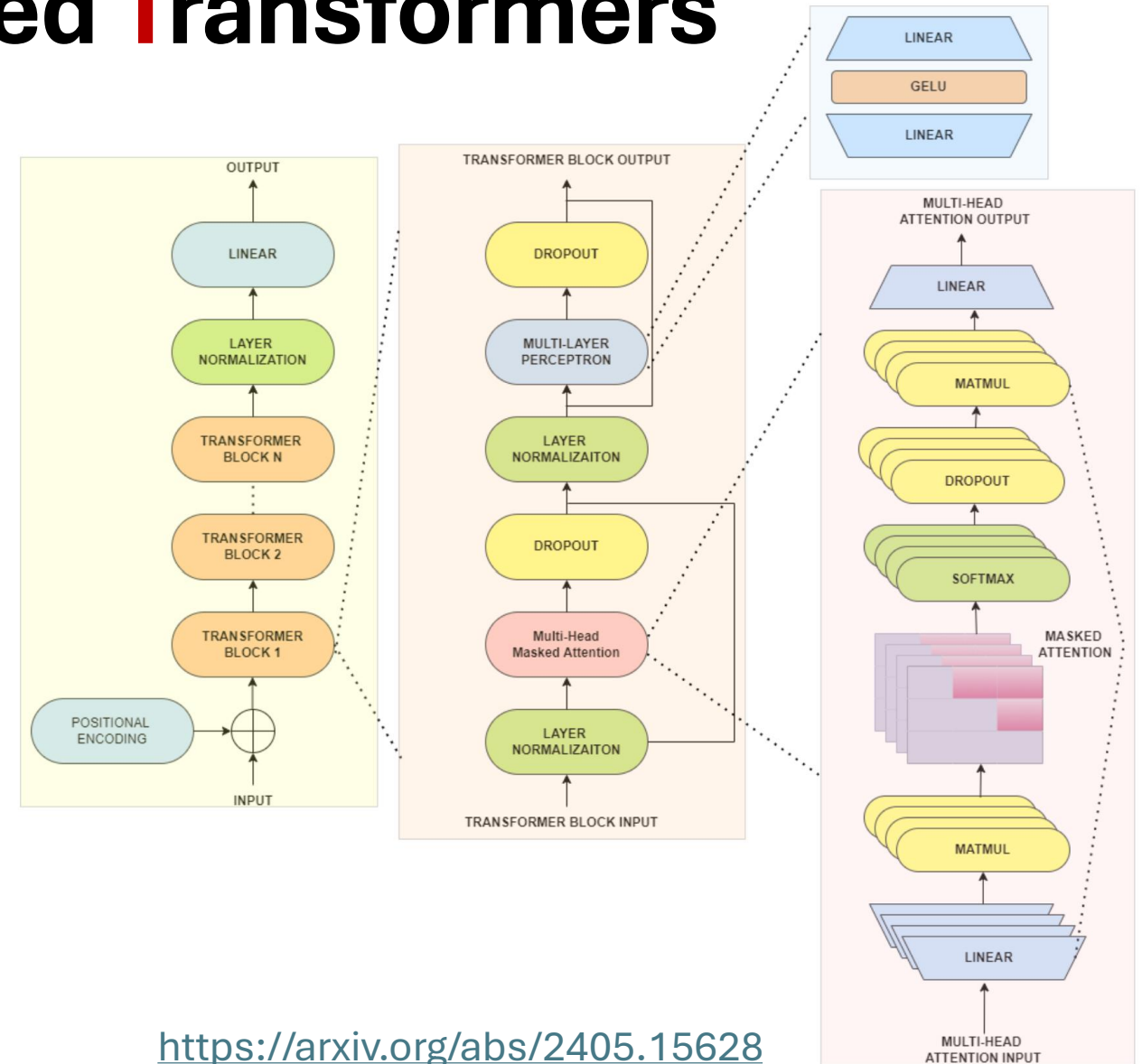
Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.



# Generative Pre-Trained Transformers (GPT)

**ChatGPT** was built using the transformer components first introduced by Google.

ChatGPT2/3 were mainly massive **Feed-Forward Self-Attention** based Transformer models.

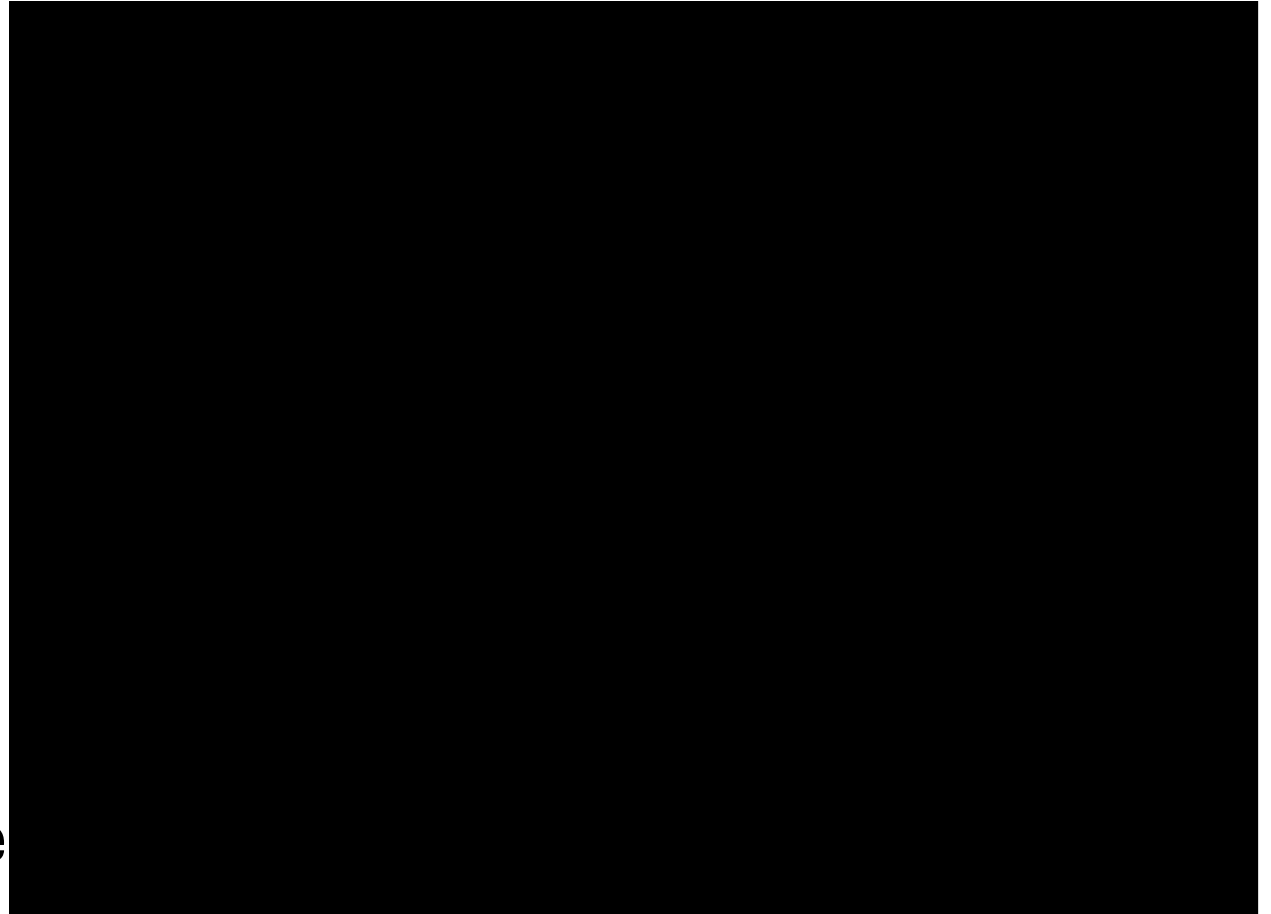


<https://arxiv.org/abs/2405.15628>

Fig. 2 GPT-2 Architecture

# DeepSeek – new best in-class(?)

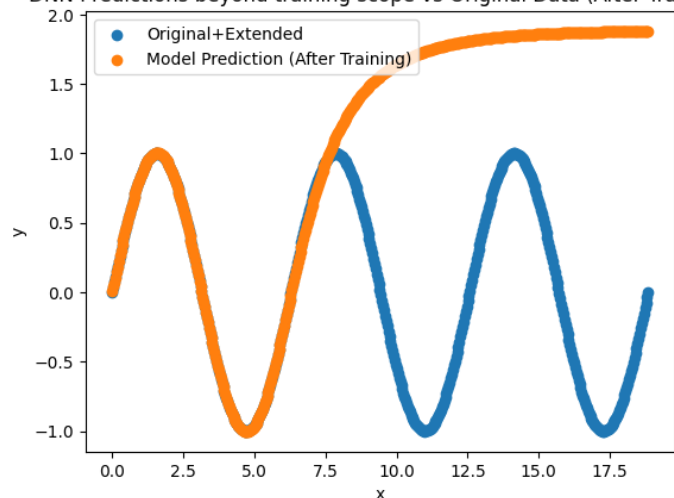
- **MHA** models perform a huge amount of complex matrix multiplications in pipelines.
- This causes an extremely large amount of data to be shuffled.
- **Multi-Headed Latent Feed-Forward Attention** is an attempt to reduce this whilst preserving model performance



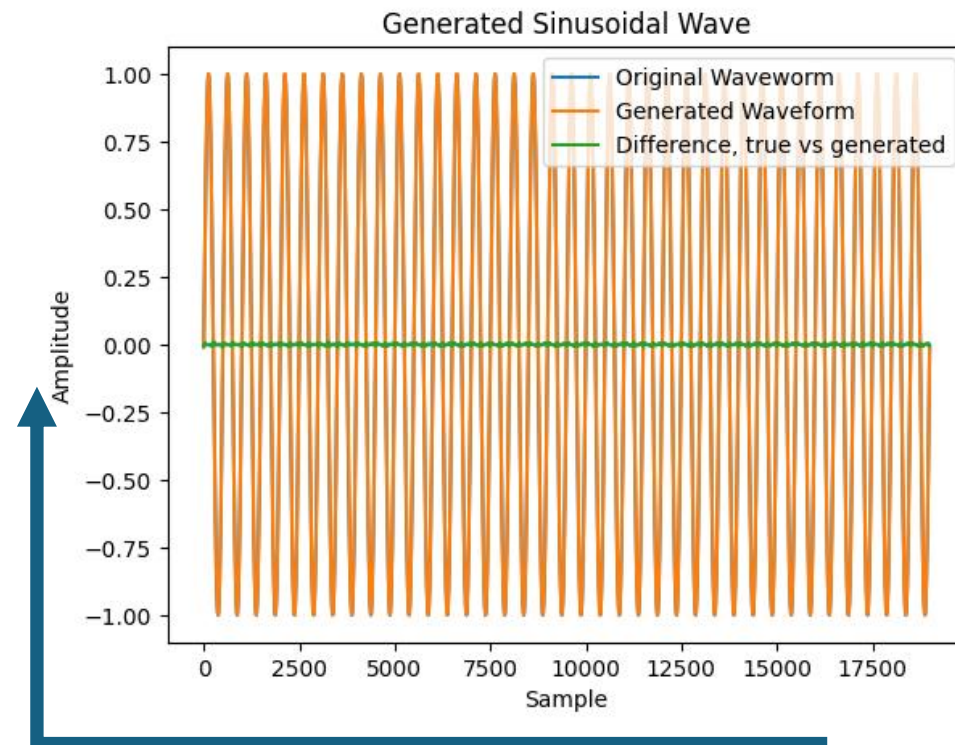
**DNN/CNN**  
**1958**

# Model Performance

DNN Predictions beyond training scope vs Original Data (After Training)

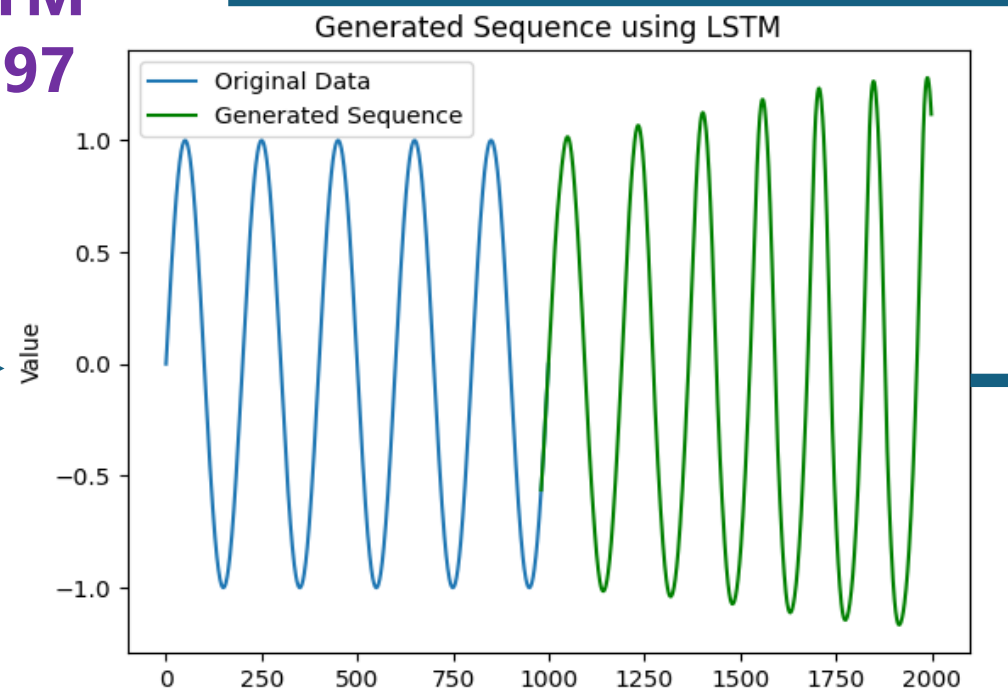
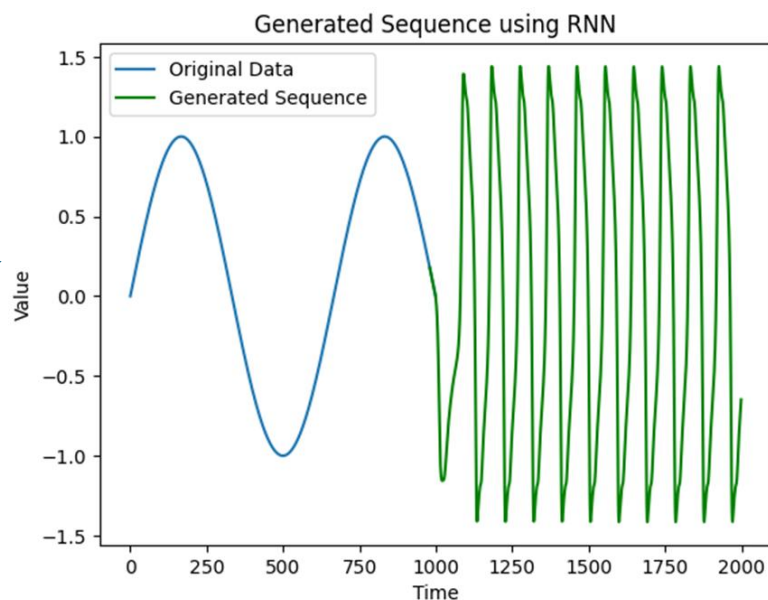


**GPT**  
**2017**



**LSTM**  
**1997**

**RNN**  
**1986**



# Transformer Models

- Transformer based models are models capable of extracting context from data. Hence, we say they are capable of “learning meaning”.
- Pretty much all Transformer based models are now based on Attention in some form given the performance of this type of model design.
- One of the often-touted advantages of ML models is “**learn once apply everywhere**”.

This means that once a single model has been designed and trained it can be used repeatedly at much lower cost.

- Transformers are also often touted as a class of model where you can:  
“**Throw hardware at a problem and reduce the expert person-power needed to solve it and only verify the result(s).**”

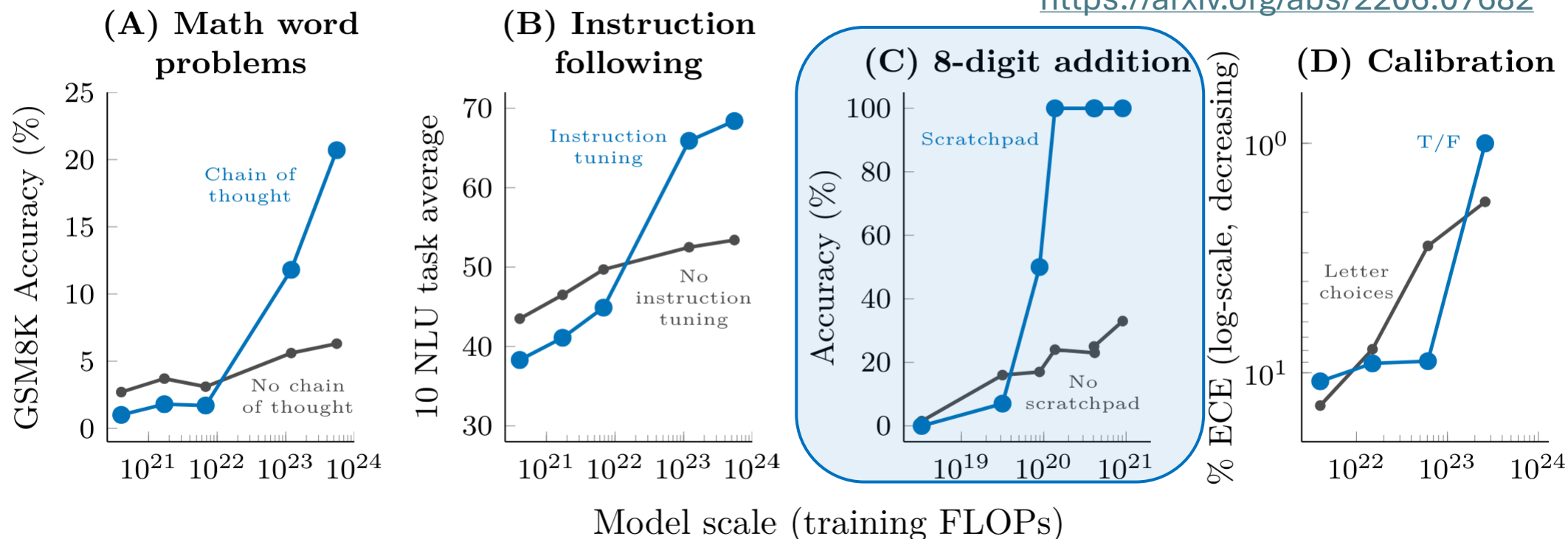
# Emergent Properties

Emergent properties observed at around  $\sim 300$ ZFLOPs of processing in modern models. (2024)

Most powerful ‘*single*’ HPC system in the world is capable of 0.001ZFLOP/s ...

→ Minimum training times of days on extremely expensive & power hungry (**1MW!**) kit.

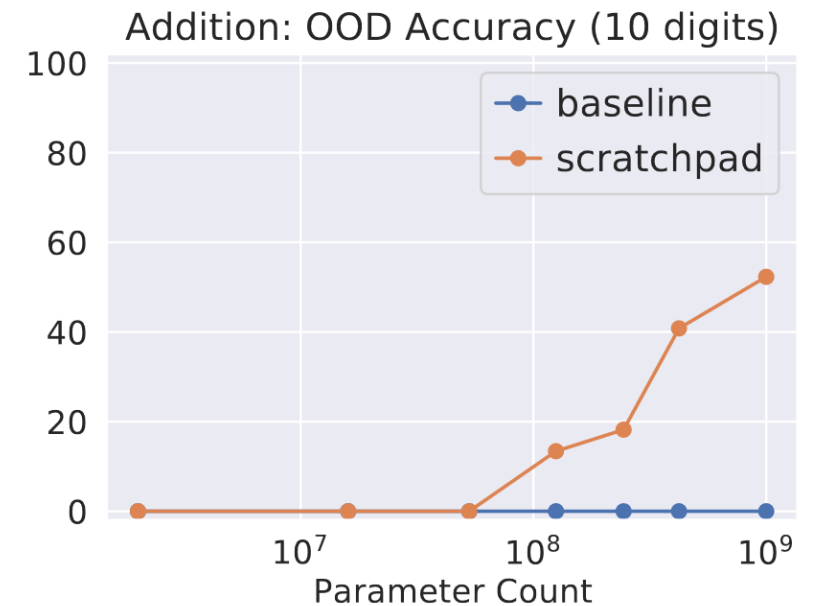
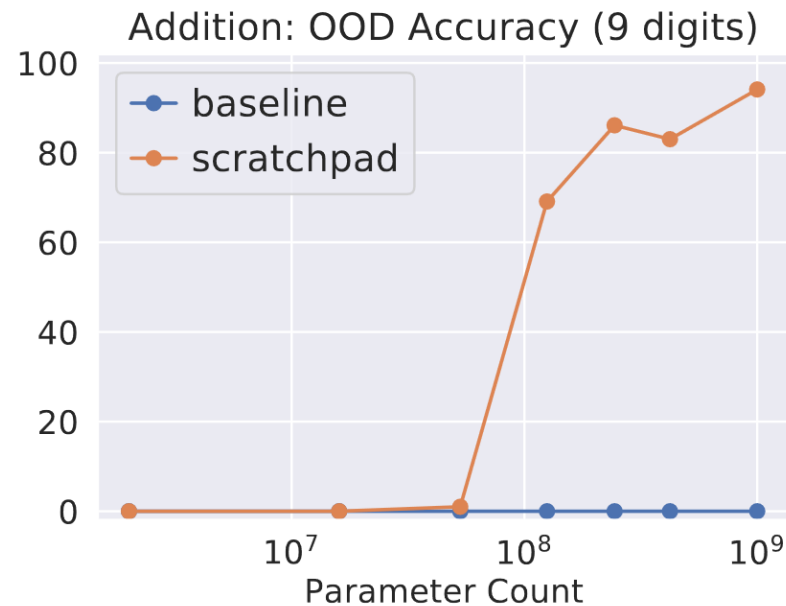
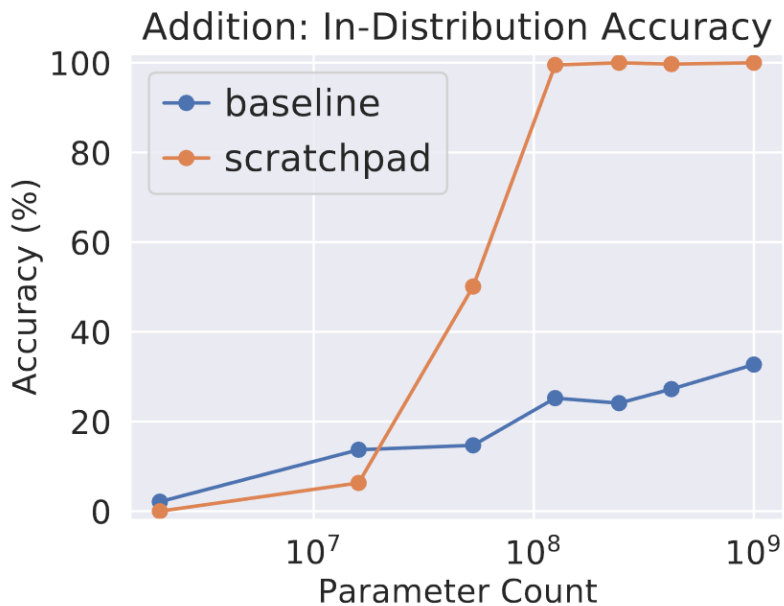
<https://arxiv.org/abs/2206.07682>



# Emergent Properties (2)

- A model capable of 8-digit addition can perform this level of calculation when the model:
  - a) has been sufficiently trained
  - b) has sufficient complexity

<https://openreview.net/pdf?id=iedYJm92o0a>




# Emergent Properties (3)

- Another example of an emergent property is that NLL models trained on English language can explain how/why a joke is funny.
- Or a model trained on coding websites can annotate, understand code, context and even identify basic coding problems and offer solutions.
- **It's not fully always understood why/how these emergent properties emerge.  
Or even why these properties suddenly emerge.**

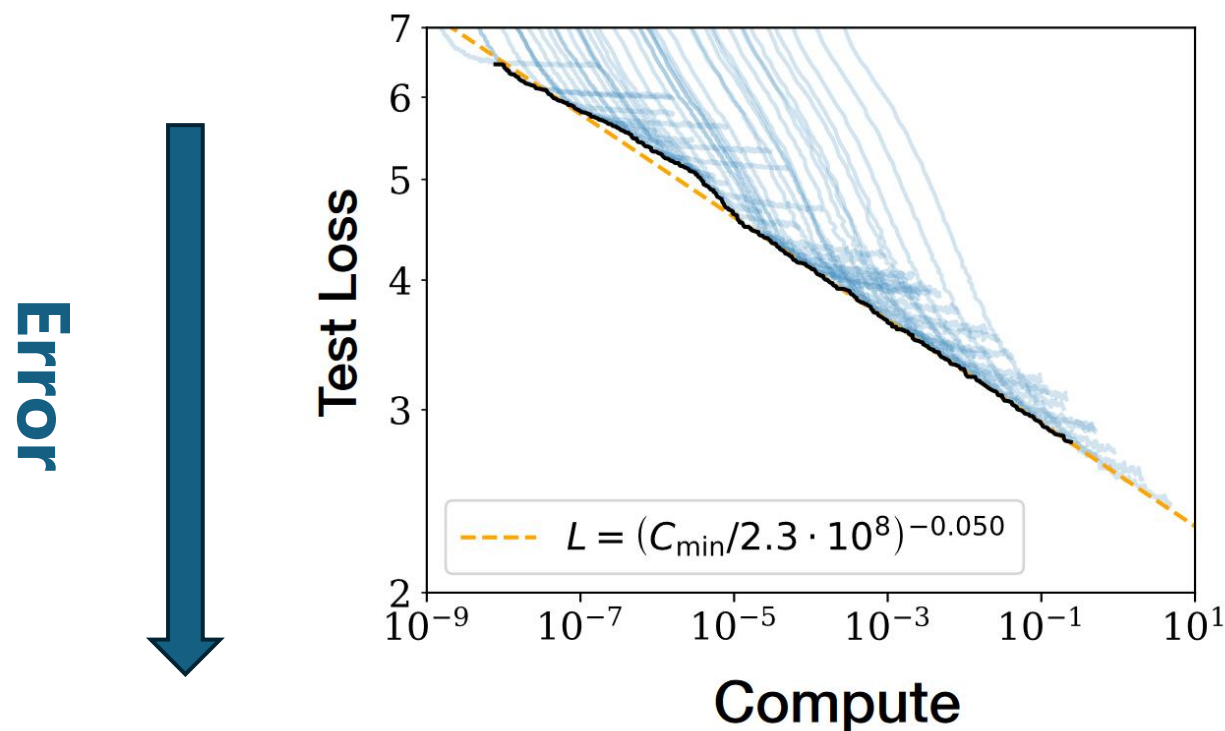
Properties from trained models are well demonstrated, but it's not obvious during training when/how these properties have emerged.

There are philosophical, mathematical & computational suggestions for how/why these properties arise...



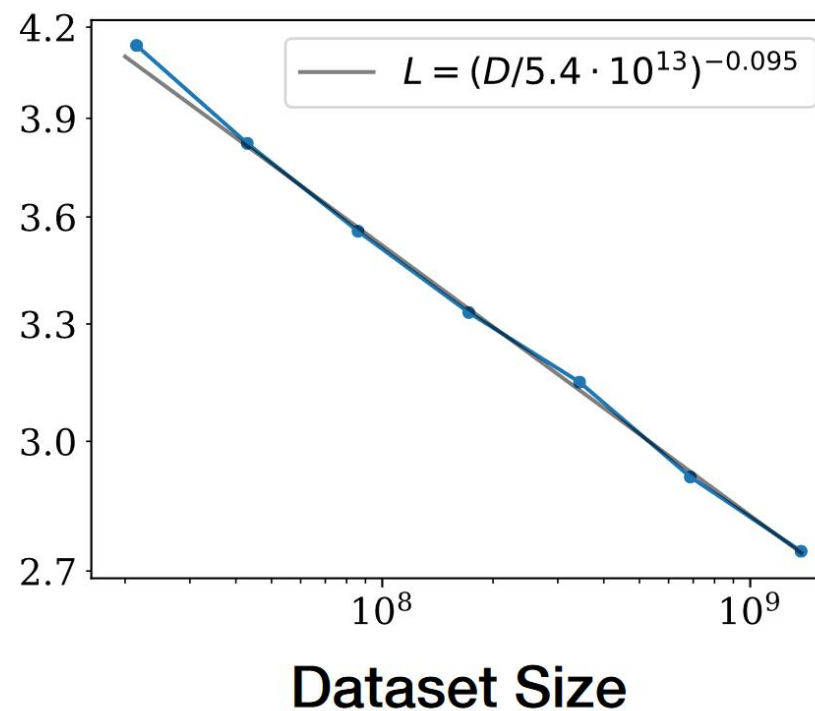
# Will things just keep getting better?

- LLM model scaling is already being strongly investigated by **OpenAI** and others.



**Model  
Size**

PF-days, non-embedding



tokens

**Dataset  
Size**

<https://arxiv.org/abs/2001.08361>



# Will things just keep getting better? (2)

- Trends observed from **ChatGPT3** training predicted **ChatGPT4** performance.
- *Not from empirical laws but observed trends.*
- LLM model scaling potentially follows the relationship:  $\alpha \propto \frac{4}{d}$ , <https://arxiv.org/abs/2004.10802>  
(here  $d$  is the dimensionality of the training data (text)...)
- OpenAI have observed their GPT models to follow a scaling of:  $\alpha \propto 0.095$   
(vs English text dataset size)
- This *could* imply that the dimensionality of English text is\*:  $d \approx 42$

*\*I claim this potentially shows there's too many physicists at OpenAI ...😊*

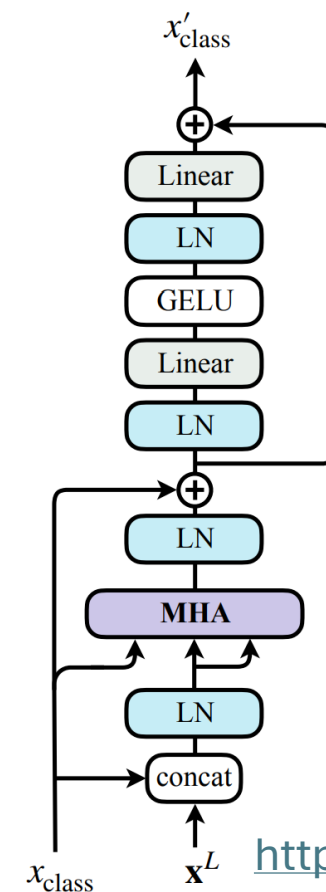
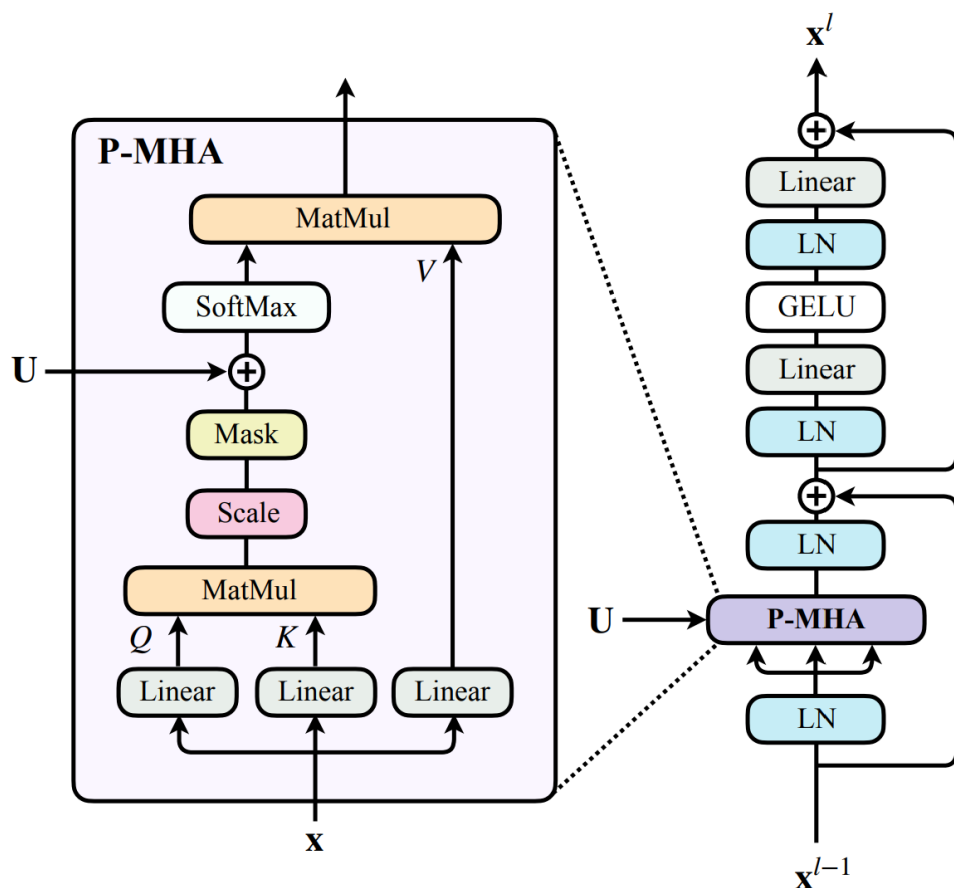
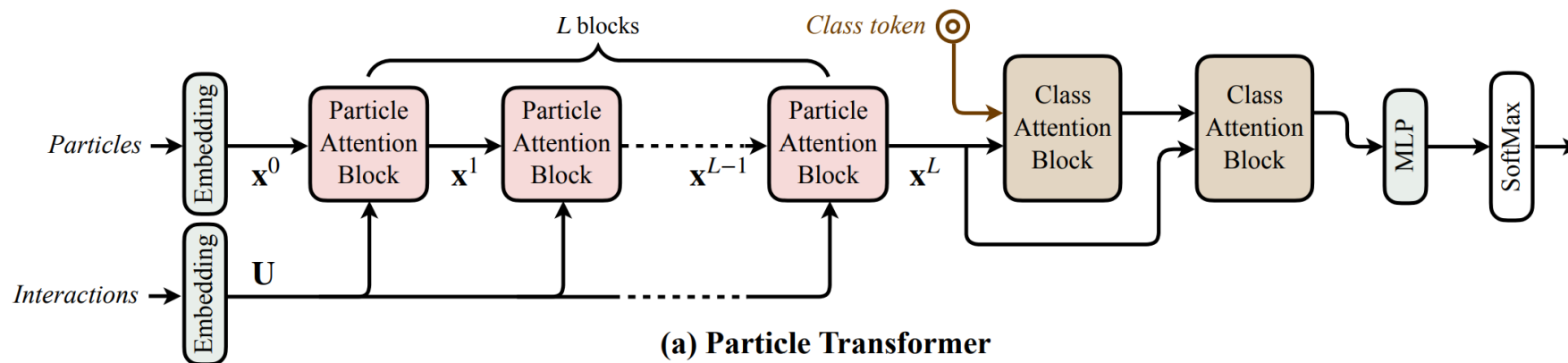
# Will things just keep getting better? (3)

- Models will improve with a combination of more parameters, more data, or, longer training times.
- However, the scaling laws that these follow are not in our favour:
- Performance  $\propto (\text{Processing Time})^{-0.050}$  ← More Power  
 $\propto (\text{Dataset Size})^{-0.095}$  ← More Data  
 $\propto (\text{Model Parameters})^{-0.076}$  ← More Hardware
- Reducing errors by 50% for a (perfect) LLM requires either:  
~10<sup>6</sup>x CPU, ~1,000x Data, or ~1,000x Parameters (!!!)

# ParticleTransformer

- Transformer based models have demonstrated a huge amount of potential over normal DNN/CNN based models.
- One example model within Particle Physics is the ParticleTransformer model. <https://arxiv.org/abs/2202.03772>
- This transformer can take un-structured information of observed particle interactions and can be trained to

**P:**



<https://arxiv.org/abs/2202.03772>

Figure 3. The architecture of (a) Particle Transformer (b) Particle Attention Block (c) Class Attention Block.

# ParticleTransformer

- Particle Transformer as an example model has 2 “interesting” features not found in all Transformer based models:

1. Additional **U** connection between all Attention layers.

This adds some global relationship between all the particles in the decay.

2. A **Class Token** injected toward the end of the model.

Research from Facebook on multi-layer attention-based transformer models has shown the performance of the models as classifiers is better when the class tokens are injected later into the model graph.

# Workshop Today

- **1bit\_Classifier\_Workshop.ipynb :**

This notebook will guide you through training a 1-bit DNN Classifier model.

This is the quickest problem to run computationally today.

- **ModelCompare\_Workshop.ipynb :**

This notebook will guide you through looking at a problem I illuded to last week. How well do different models react to unseen data?

This takes a reasonable amount of compute as you will be building/training a DNN, CNN and Transformer based classifier.

# Workshop Today

- **CNN\_TF\_Best\_Workshop.ipynb** :

This is the most computationally intensive notebook I'm going to present you.  
It does the following:

1. Build and Train an AutoEncoder on the mnist dataset
2. Take the Encoder component from the AE and build a new Transformer-based classifier
3. Train this classifier using the “pre-trained” encoder
4. Further fine-tune the complete stable model

This is a way of building an **extremely stable** classifier which can cope well with un-seen inputs not present in the training dataset.

It scores **>90% on artificially supplemented mnist test and training datasets** and **~99% on the original test mnist test image categorization dataset.**