More complex Machine Learning models

AGQ Lecture 3:

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

DNN – Some Caveats

- **DNN** don’t perform well with noisy inputs or with small signals

- We have the problem of ***Vanishing Gradients***
  (e.g. **DNN** of depth ~10 are *very* hard to train)

- Doesn’t work well with input changing i.e.
  - Not very good at extracting features if input is translated/transformed

- Requires only single numerical inputs per-neuron
  - Doesn’t scale very well with larger datapoints/datasets

- Effectively a black-box.
  We don’t know what a *single weight* deep inside 2 fully interconnected dense layers might represent in the *real world*...

ANN using 2D inputs

- Take a 2D photographic image as an example.

  (e.g., telescopic pictures, particle tracks, cracks in 2D materials, …)

- For a **4k colour** image we need to use,

  3 * 3840 * 2160 * n = 24,883,200 * neurons weights to use the input with a raw DNN.

- If this is a picture of a cat, we want to extract key features for instance such as *'number of whiskers'*, *'length of fur', or 'size of teeth'.*

- Ideally, we want to extract this automatically without having to label every feature in every image manually.

This can be achieved via Image Processing…

ANN & Image (pre-)processing

If we want to use a **DNN** to analyse a sample dataset we want to make sure that it's extracting correlations between **signal** features in the dataset and ***not background or noise***.

An obvious step in this case is to consider pre-processing *all* our input data so that the **DNN** works well with it.

Doing this manually is time consuming, expensive and in-efficient, and is what humans do to verify that they're not robots on websites.

Larg(er) ANN datasets

- **ChatGPT3** *as an example* as 4k token input and ~10 layers

- Building a comparable model with pure **DNN** nodes would require 4k **neurons** per layer and 10 layers.

- This would give 4x10^{4} neurons each with 1x10^{7} weights or **~4x10^{11} free parameters** which need optimizing.

- Facebook's *Llama* models which are still best in class for certain tasks only has ***~1x10^{7} free parameters*** *total*.

Larg(er) ANN datasets

- We want/**need** a way of *automatically* extracting the important bits of information from our input.
- At a very high level this means we need to **filter** our data in some way.
- Keep/emphasise/extract the important parts (signal) and ignore background/noise.
- *Easiest* to discuss this with pictographic data

Image Filtering

Before we start applying filtering techniques to our data, let's take a step back and look at the example of image filtering.

One of the most famous examples of this is edge detection filtering or applying ***Sobel*** operators to an input image.

This gives us the advantage of extracting out the features (edges) of a scene without having to care about the background of an image.

Image Filtering – Convolutions

Image filters are applied by convolving an operator with an input to give an output.

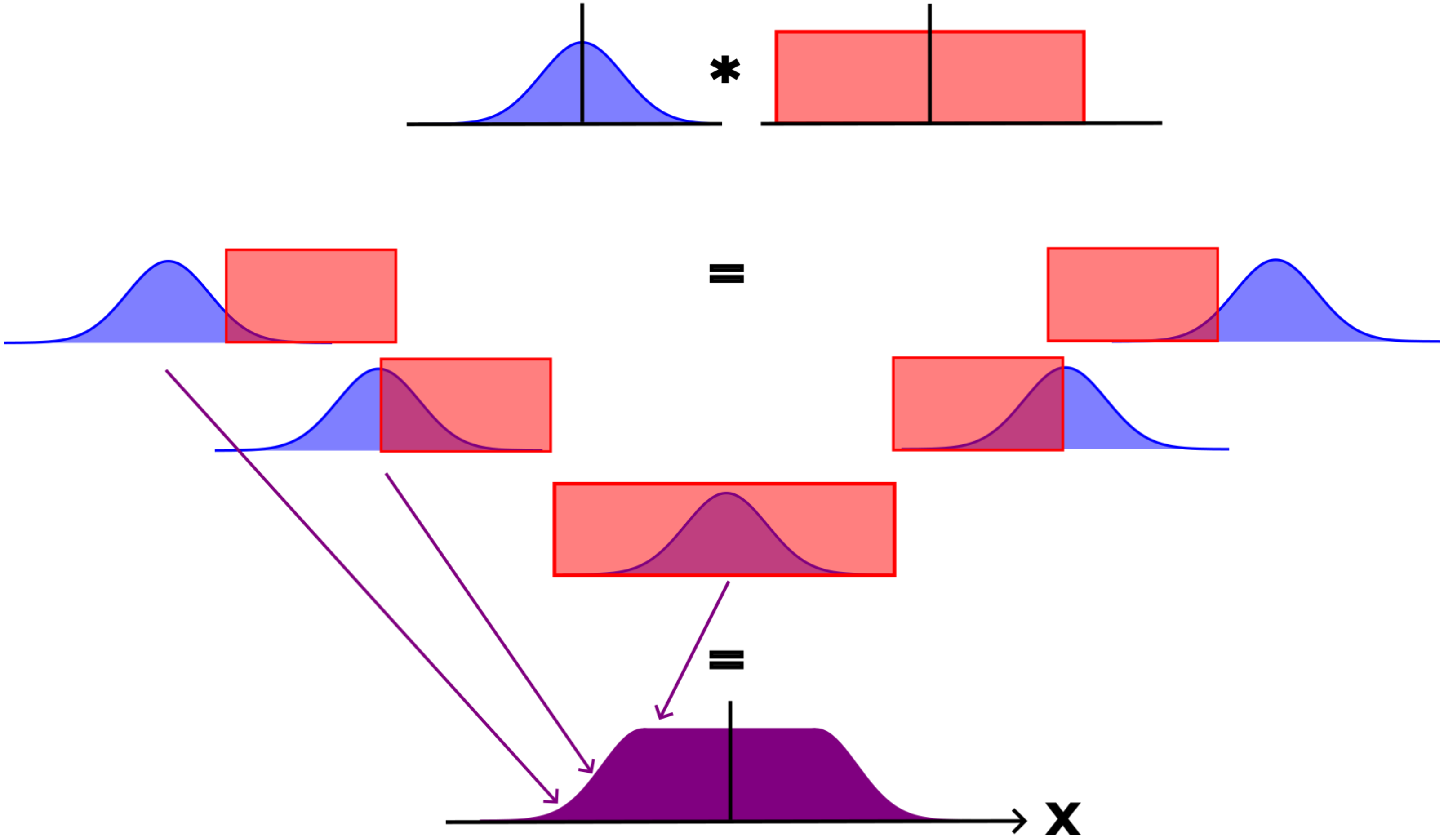
i.e. $Input * \ Operator = Output$

$$(f * g)(t) = \int_{-\infty}^{\infty} f(t) g(t - \tau) d\tau$$

These operators can be used to sharpen, blur, up/down-sample data.

Convolutions – 1D example

$(f \quad * \quad g)(x)$



Convolutions – 2D

The 2nd dimension!

Now we want to convolve 2D functions/operators with 2D data.

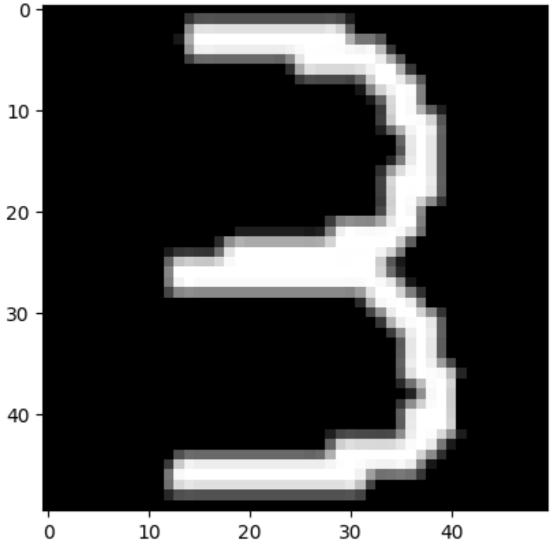
One of the most common 2D functions to demonstrate as I've mentioned is the Sobel Operator*(s)*.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

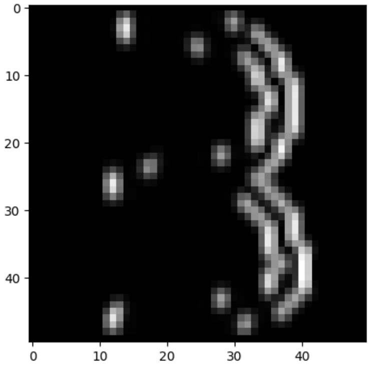
$$\mathbf{G} = \sqrt{G_x^2 + G_y^2} \qquad \Theta = \mathrm{atan}(G_y, G_x)$$

Convolutions – 2D example

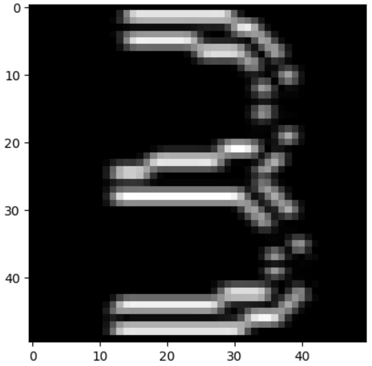
Input=



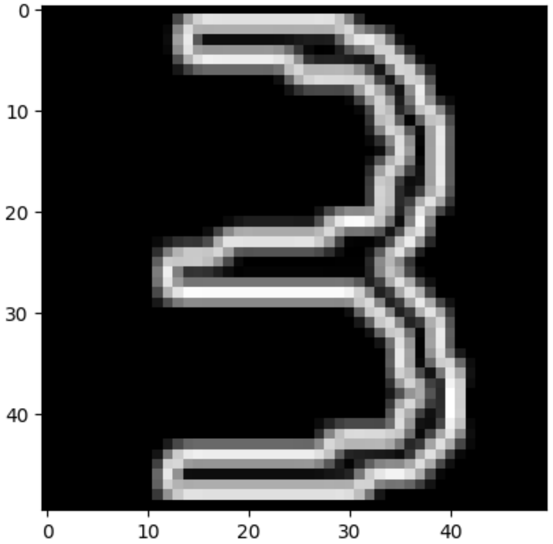
$\mathrm{G_x} * Input =$



$\mathrm{G_y} * Input =$



$\mathbf{G} * Input = Output =$



ANN – Image Processing

- **Sobel** operators are a specific example of a using a kernel matrix of dimension **(3,3)** to extract features from the input.

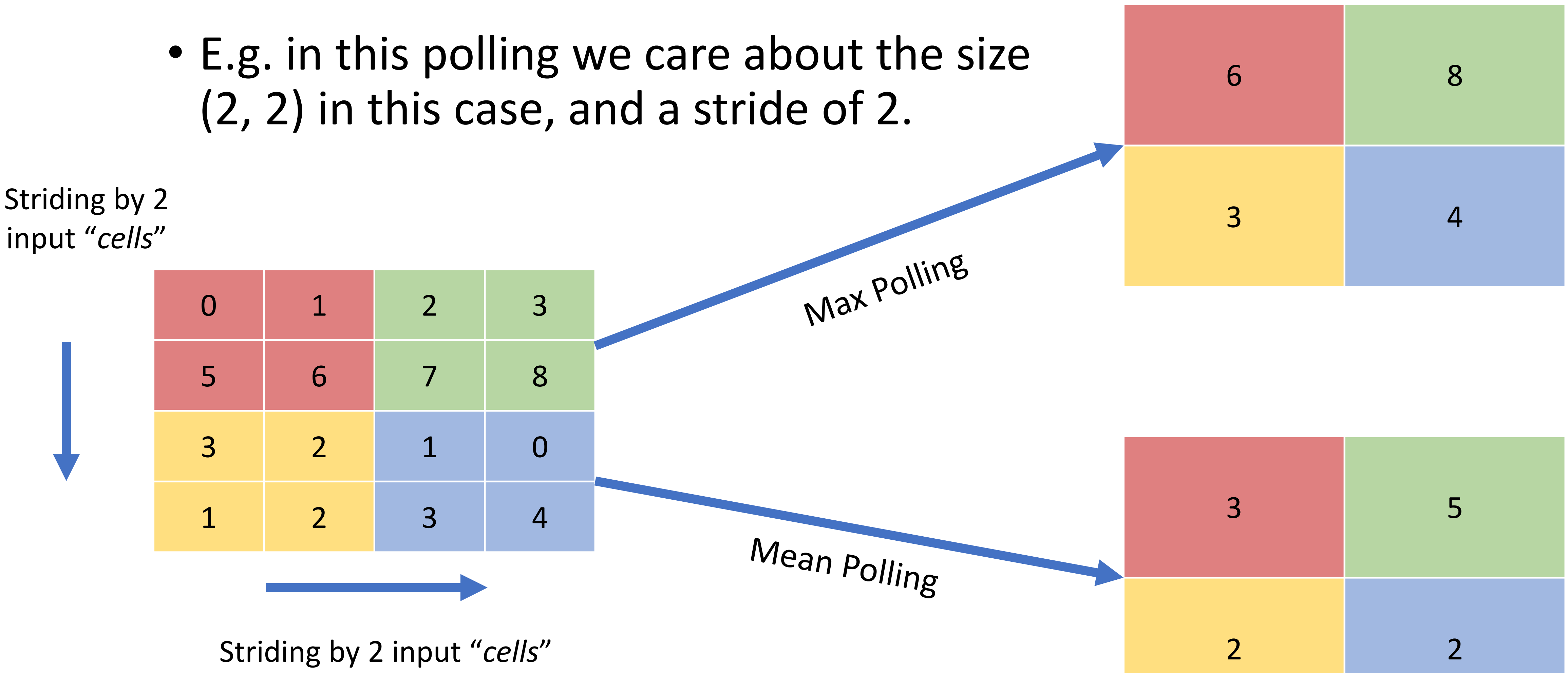
- Other common image manipulation filters are; MaxPolling, AveragePolling, Flatten, Bounding-Box, Sharpening, Skew, …

- Let’s go over **MaxPolling** and **MeanPolling**.

- Flatten and Resize, are hopefully self-explanatory

Image Manipulation – Other filters

- E.g. in this polling we care about the size (2, 2) in this case, and a stride of 2.



Striding by 2 input *“cells”*

0	1	2	3
5	6	7	8
3	2	1	0
1	2	3	4

Striding by 2 input *“cells”*

Max Polling

6	8
3	4

Mean Polling

3	5
2	2

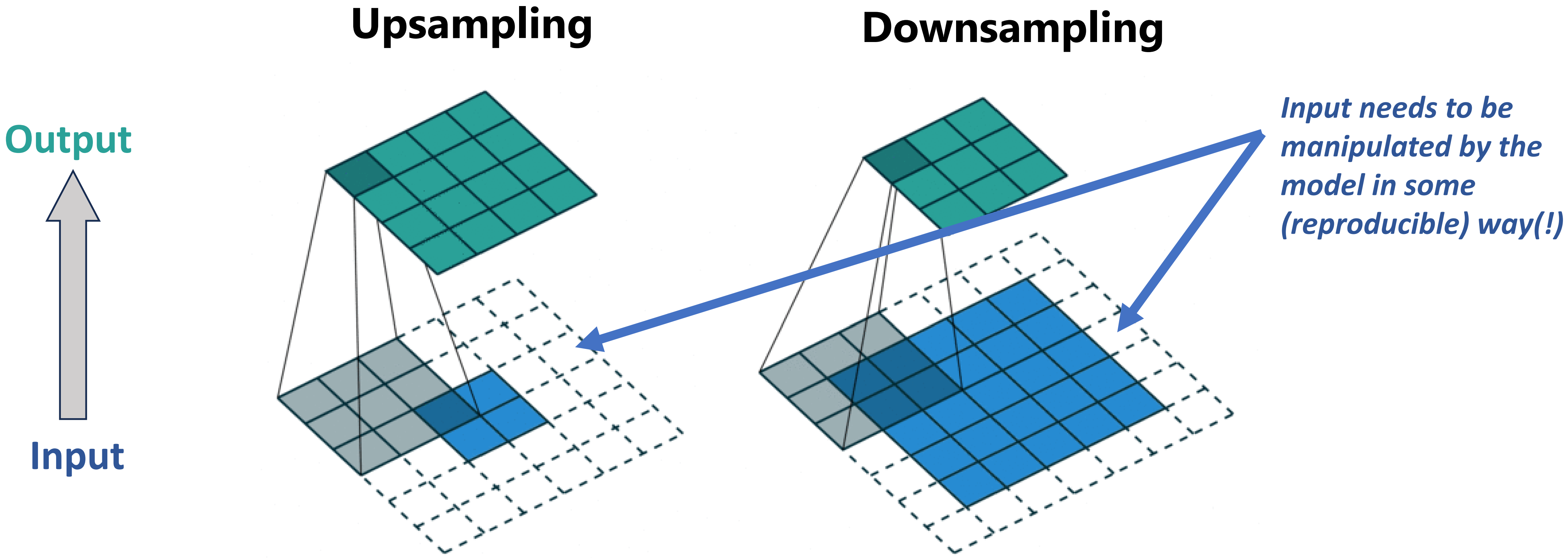
Image Manipulation – Common Nomenclature

- **Kernel**: This is a matrix of size (n, m). Usually square, but not always.
- **Strides**: This governs the ‘step-size’ of a kernel as it’s applied to input.
- **Padding**: This determines the behaviour of the algorithm at the edges of the input/output.
- **Filters**: This is the number of independent kernels which are to be trained over a given input step.

Image Manipulation – Common Nomenclature

Different **CNN** transformations allow for accessing/manipulating data in different ways.

https://github.com/vdumoulin/conv_arithmetic



Upsampling

Downsampling

Output

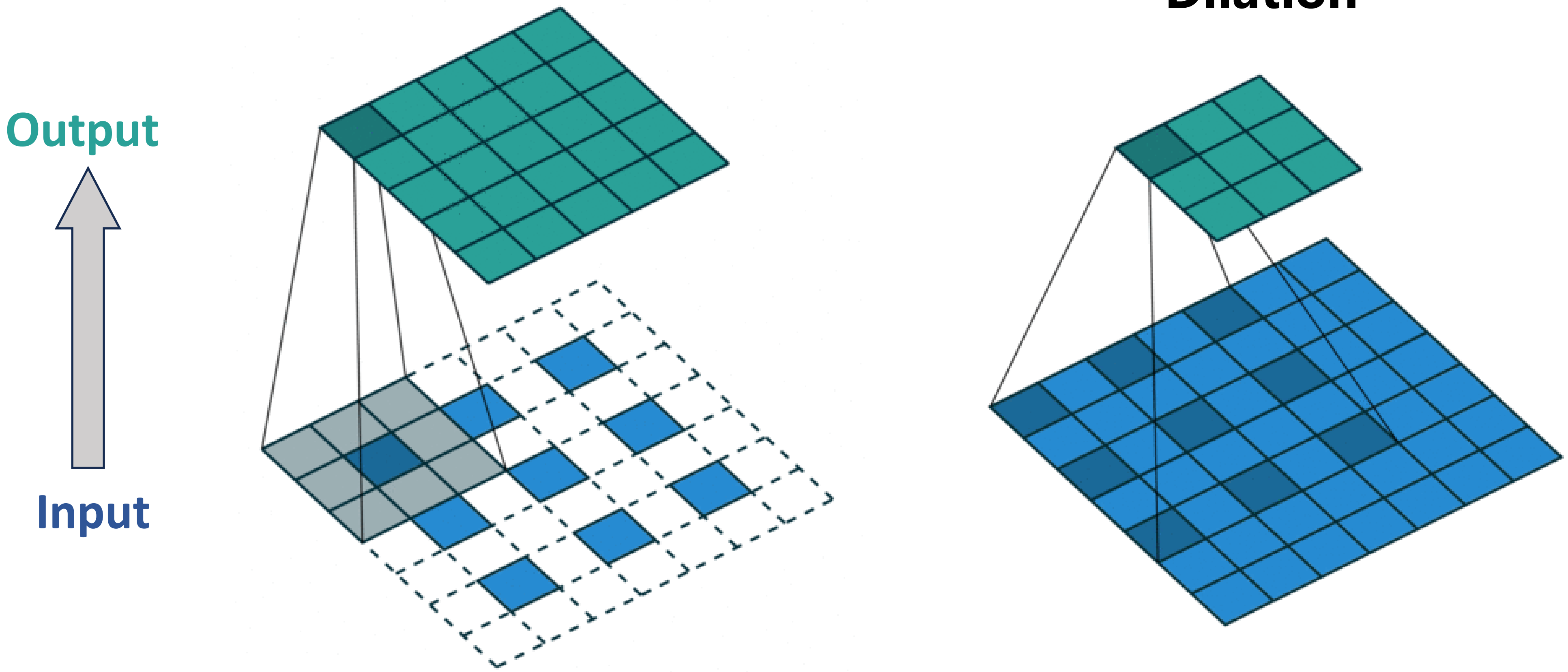
Input

Input needs to be manipulated by the model in some (reproducible) way(!)

Image Manipulation – Common Nomenclature

Different **CNN** transformations allow for accessing/manipulating data in different ways.

https://github.com/vdumoulin/conv_arithmetic



Transposing

Dilation

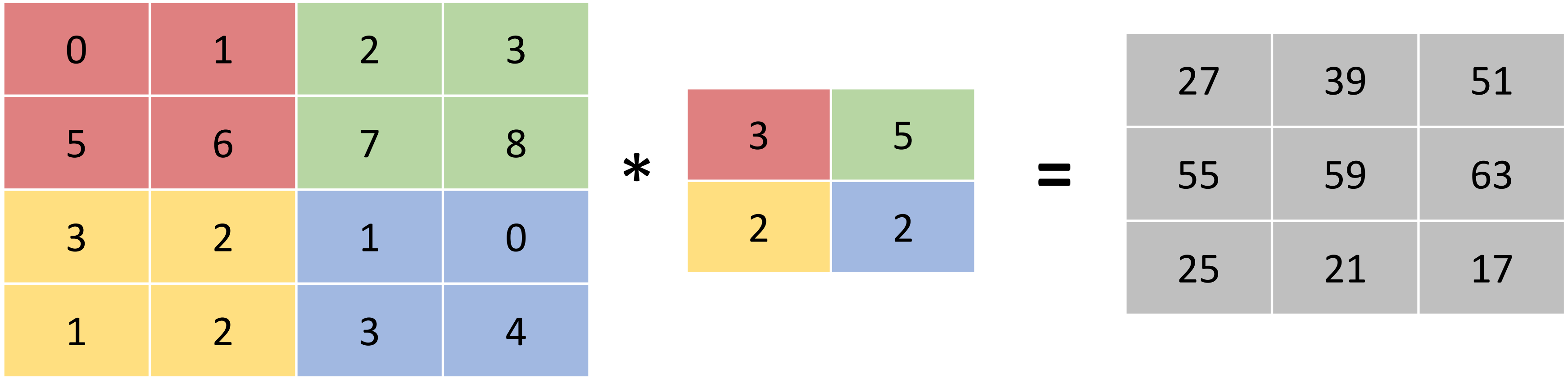
Output

Input

Conv2DTranspose – An Aside

Forward convolution typically decreases-dimensions/downsamples.

This extracts information from the input image.

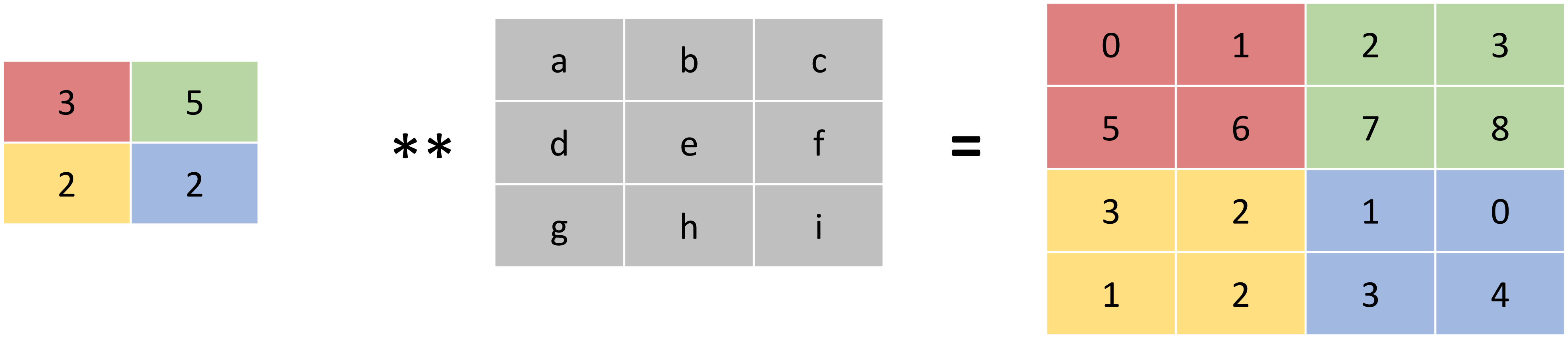


$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix} * \begin{bmatrix} 3 & 5 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 27 & 39 & 51 \\ 55 & 59 & 63 \\ 25 & 21 & 17 \end{bmatrix}$$

Conv2DTranspose – An Aside

Backward/transposed convolution increases-dimensions/upsamples.

This means using a different kernel to the forward convolution we can recreate the original image.



$$\begin{bmatrix}3 & 5\\2 & 2\end{bmatrix} ** \begin{bmatrix}a & b & c\\d & e & f\\g & h & i\end{bmatrix} = \begin{bmatrix}0 & 1 & 2 & 3\\5 & 6 & 7 & 8\\3 & 2 & 1 & 0\\1 & 2 & 3 & 4\end{bmatrix}$$

CNN neurons

- By taking what we’ve just learned about Image Processing algorithms how do we now using a ML neuron?

- Each neuron now contains a ***Convolution*** not a **multiplication**.

- This means instead of 1 *numerical weight* per neuron we now have **1 *matrix of weights*** meaning multiple model parameters per-neuron.

- We still want to use ***Activation functions*** and ***biases*** as these are independent of our neuron’s internals.

CNN neurons

To better examine/parse dense data we can replace scalar multiplications with matrices.

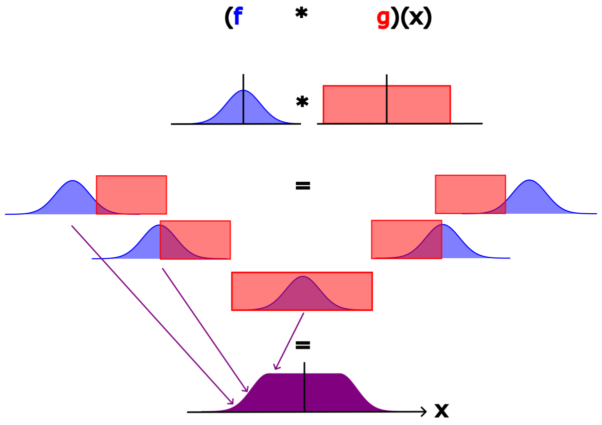
Moving from linear maths to matrix maths.

Matrix of weights

Input Matrix of values

$$\begin{matrix}\mathit{Neuron}\\ \mathit{Output}\end{matrix} = \mathit{Activation}\left(\mathit{Conv2D}(\mathit{Weights}, \mathit{Input})\right)$$

Output Matrix of values



CNN neurons

- This means we’re now applying kernels of different sizes to extract different features from our dataset.

- **Sobel** operators extract just flat edges, but a kernel could extract more complex features depending on its definition.

- We leave the content of the kernel to be determined by training on a given dataset, this allows us to use a **CNN** to extract whatever image features best describe or classify our input data.

CNN Classifier network design

- One of the most famous **CNN** designs is from *Yann LeCun et al.* 1998

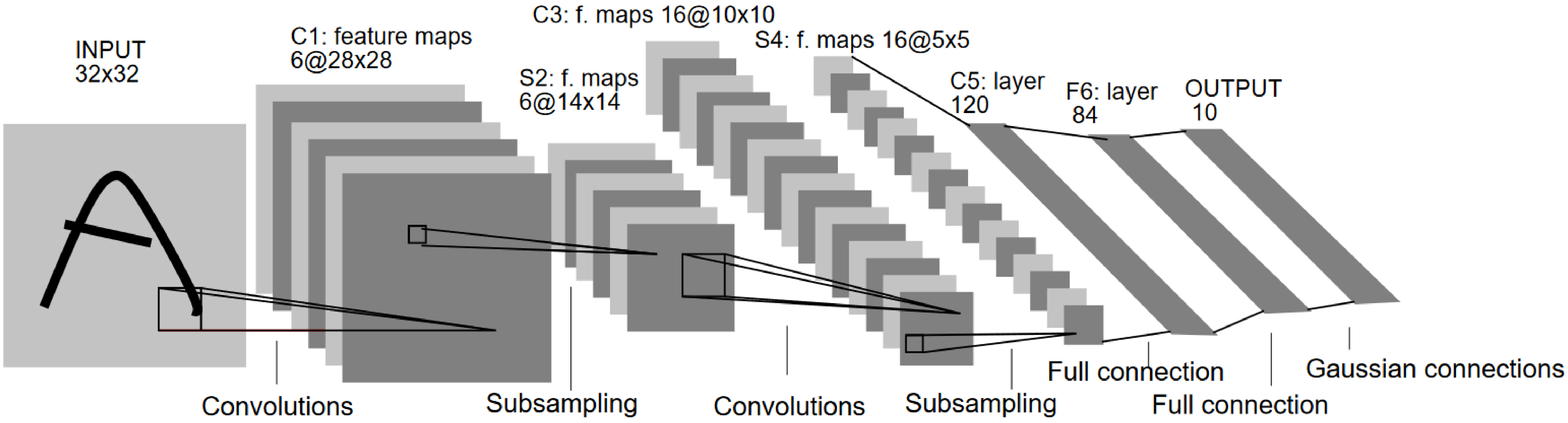
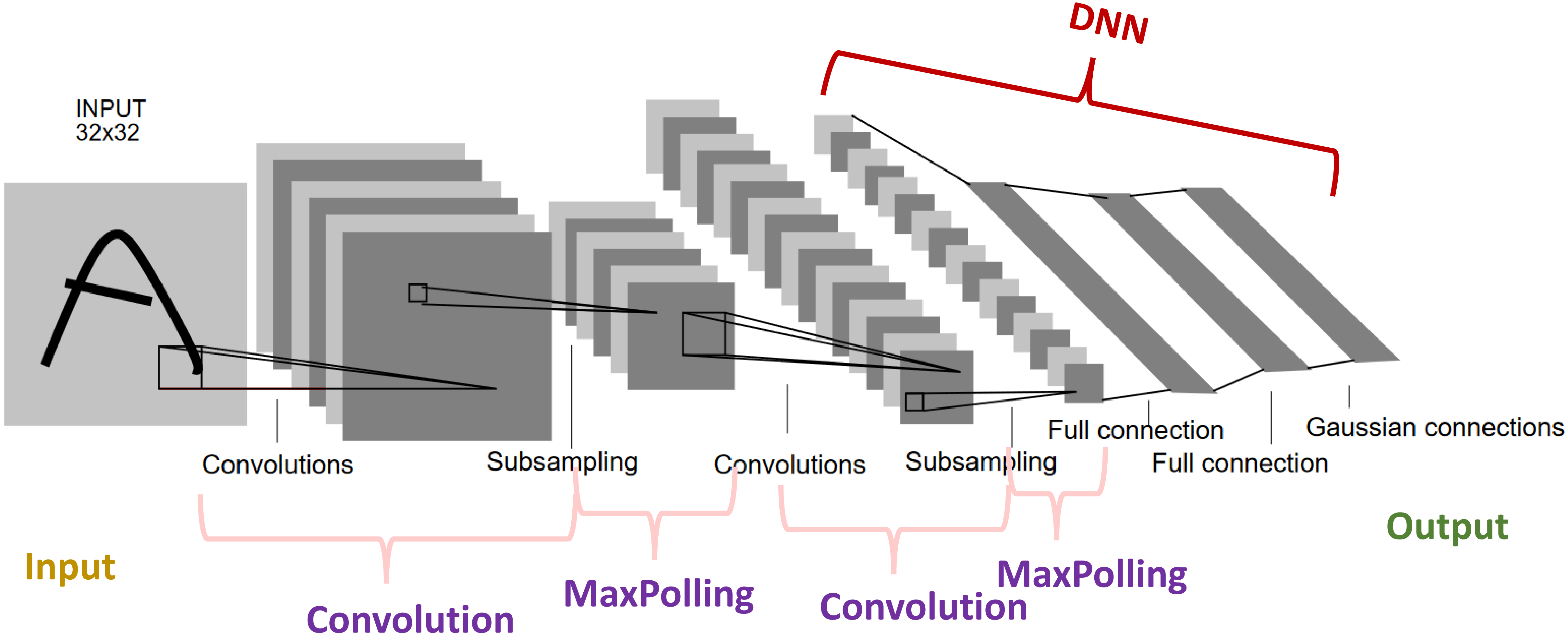


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

CNN Classifier network design



CNN – Network Design

- **CNN** networks begin with image pre-processing steps (**convolutions** and **max-polling**).

- The *output* from this **Convolution** layer is a reduced image format which is passed through successive pre-processing layers with potentially many kernels.

- Eventually for a classifier the output is reduced then passed into a **DNN** which we've already seen.

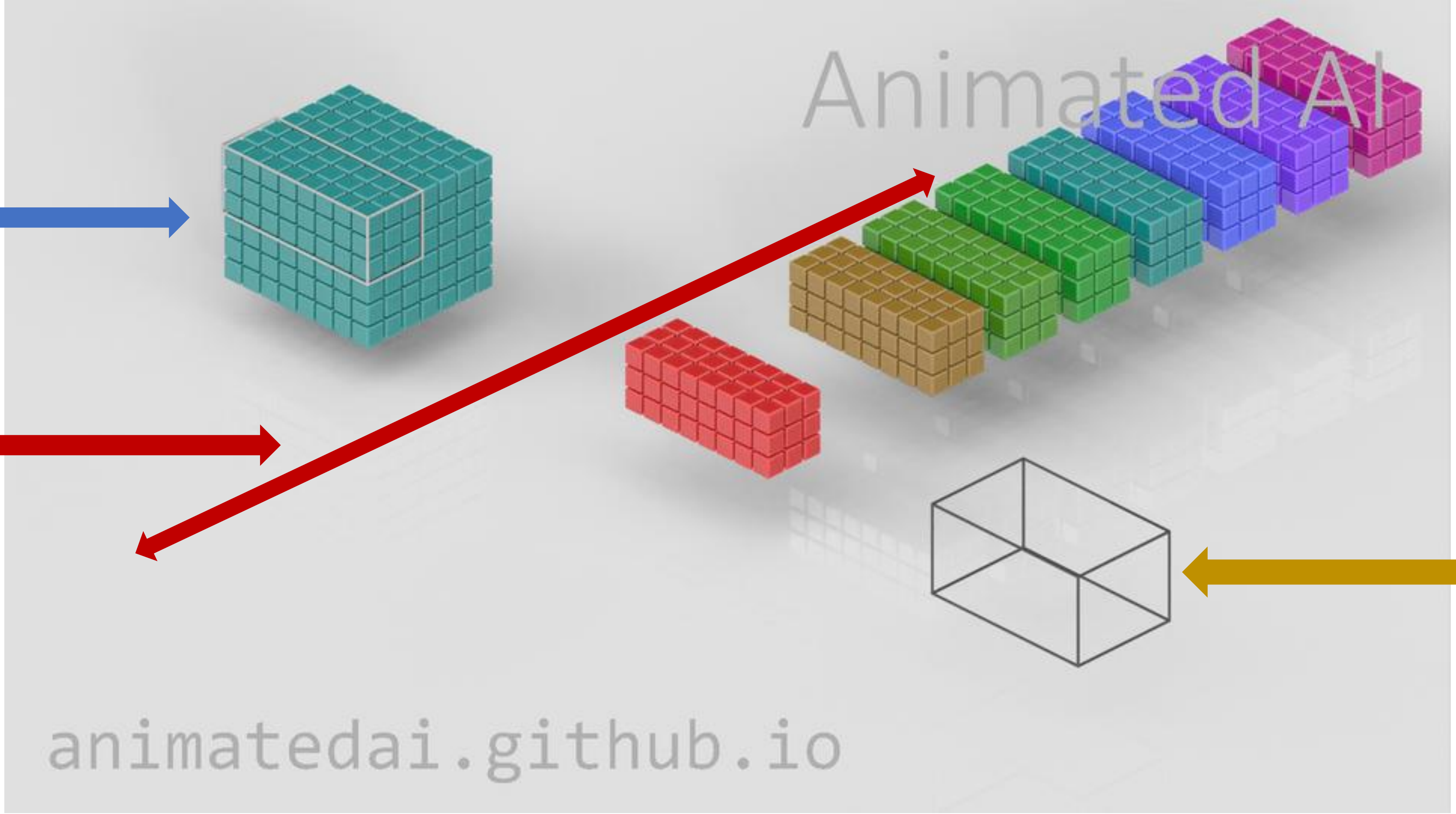
- This reduces the amount of data needing to be passed to a **DNN** and for the **DNN** to more easily separate ***key features*** from background in the data.

  E.g. *4k image may be reduced to 256x256x1 inputs after extracting features…*

CNN – Network Design

Inputs are typically n-dimensional with n-dimensional transformations…

https://animatedai.github.io/



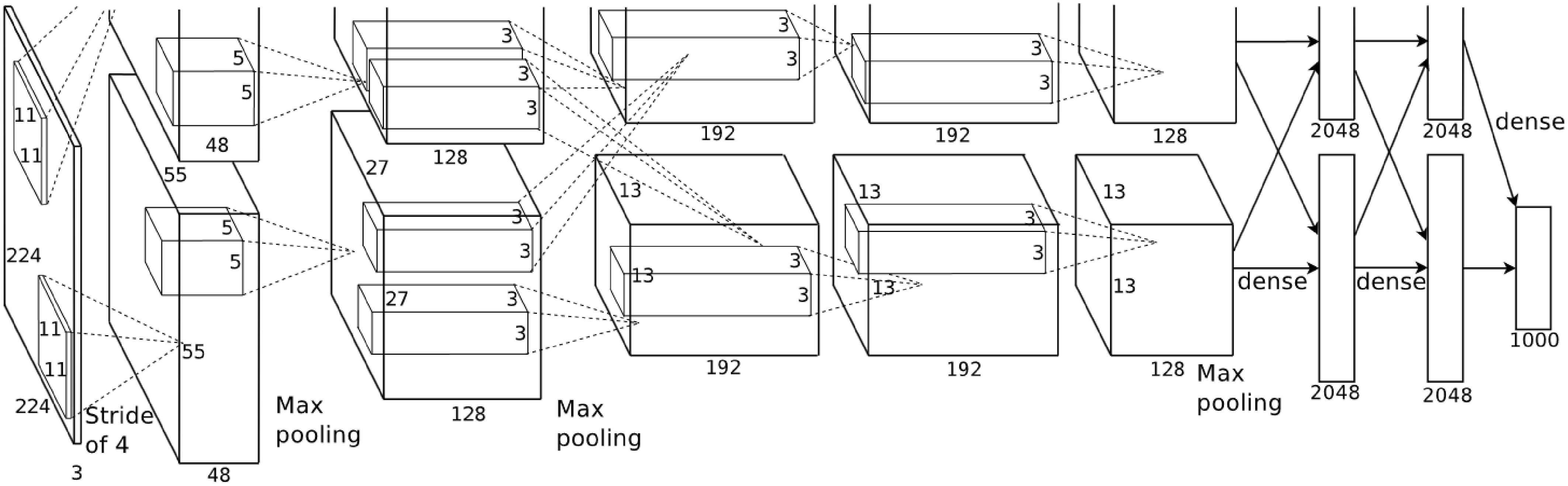
Layer INPUT

Trainable Model Matrices

Output from this Layer

CNN Classifiers – AlexNet

- **AlexNet** developed by Alex Krizhevsky et al, and published in 2012 represents what is now taken to be the basis of modern **CNN** classifier designs.
- This design was much more complex than **LeNet5**:



CNN / CDNN – Network Design

- **C**onvolutional **N**eural **N**etworks or **C**onvolutional **D**eep **N**eural **N**etworks are a combination of various pieces of technology.

- This design allows us to extract information from a given set of labelled inputs using supervised learning.

  i.e. building a classifier

- However, this model design allows us to consider the use-case of un-supervised learning.

  i.e. using the model to *automatically* extract common features.

  Does this "9" look like a "7" to you?

Uses of CNN networks

- **CNN** networks ***extracts key features*** from a dataset in an automated way.

- This means that a **CNN** can encode information and extract the key features through training on a given dataset without the need for labels.

- We can then store this information in a latent space representing the input information the model has been trained on.

- This is an example of "**Dimensionality Reduction**".

- One of the most common network designs for this is an **AutoEncoder**.

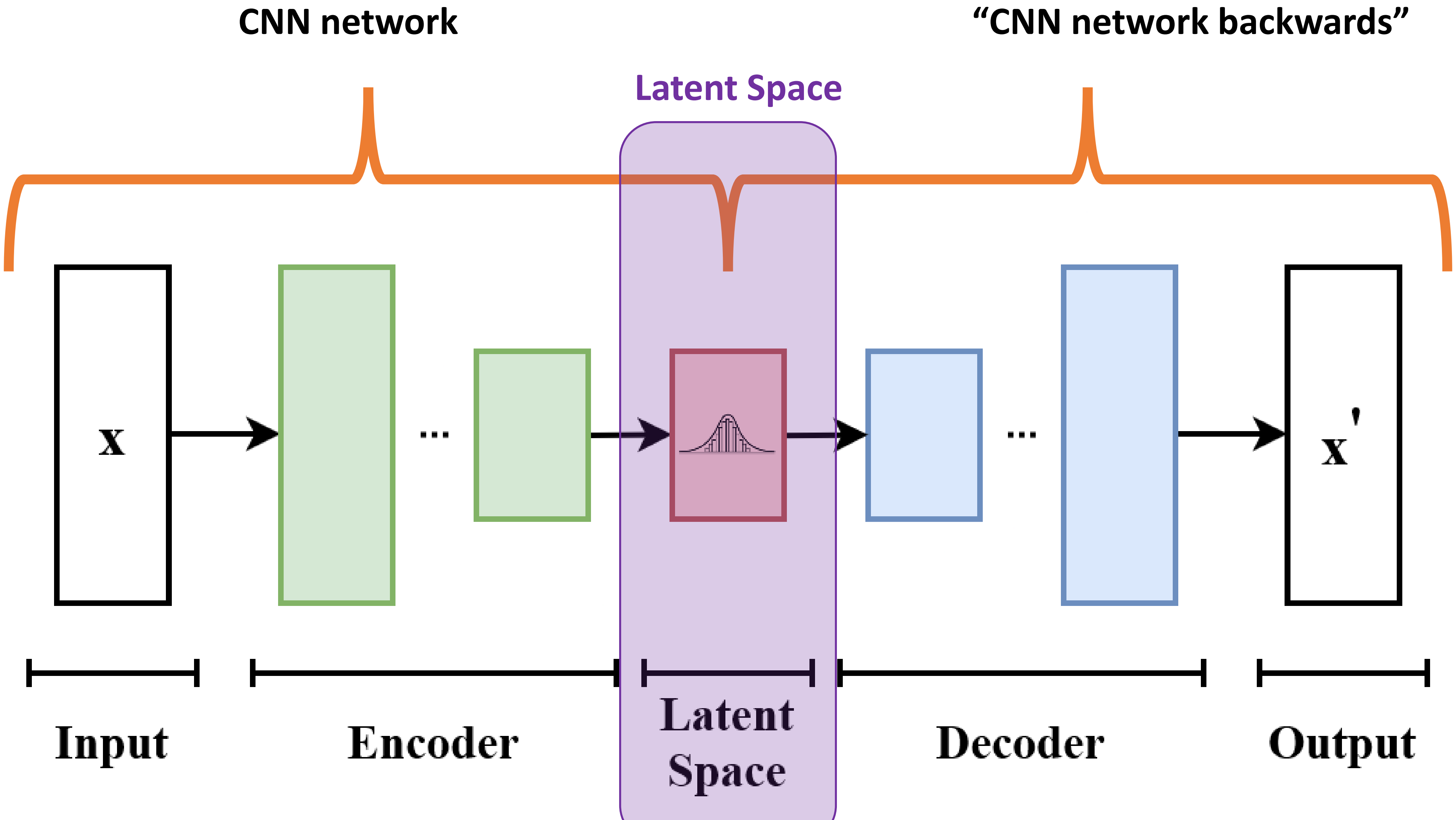
Uses of CNN networks

- Down-sampling or Up-sampling is achievable through many different mechanisms.

- Normally we use **conv2d** to down-sample and **transposeconv2d** to up-sample in PyTorch.

- In this case the sampling is normally achieved through the size of the stride in applying our convolution.

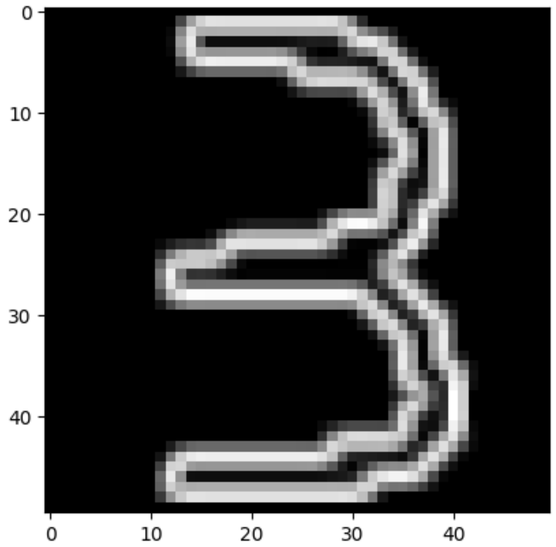
AutoEncoder – Network design



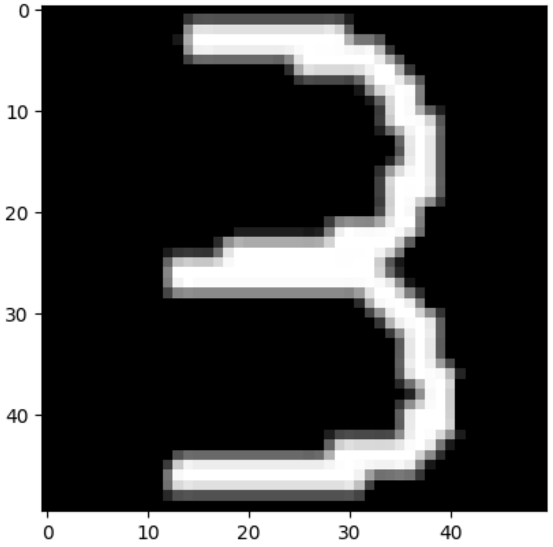
Conv2DTranspose – An Aside

- Not always trivial to know what the form of the kernel is that we need to get from input to output.

- Thankfully, we know our input and output, so we can train this kernel to find the optimal solution using training tools ☺



$$** \begin{bmatrix} ? & \cdots & ?? \\ \vdots & \ddots & \vdots \\ ???? & \cdots & ??? \end{bmatrix} =$$



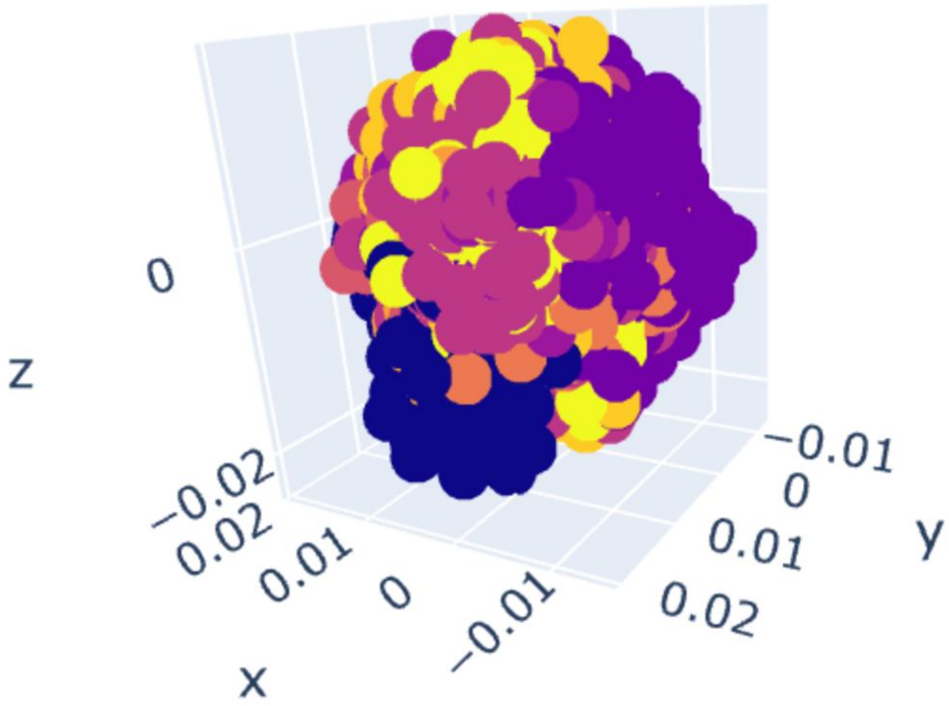
AutoEncoder – Latent Space

- An **AE** is a good example of a **semi-unsupervised** neural network.
- This network design has a bottleneck which
  reduces the amount of information which can flow through it.
- The idea behind this, is this constraint forces information to be encoded into a latent-space with coordinates representing the dataset being trained on.
- This is still effectively a ‘black-box’ style of network as we don’t have a clear interpretation of what a single weight in the system may represent in the real world.

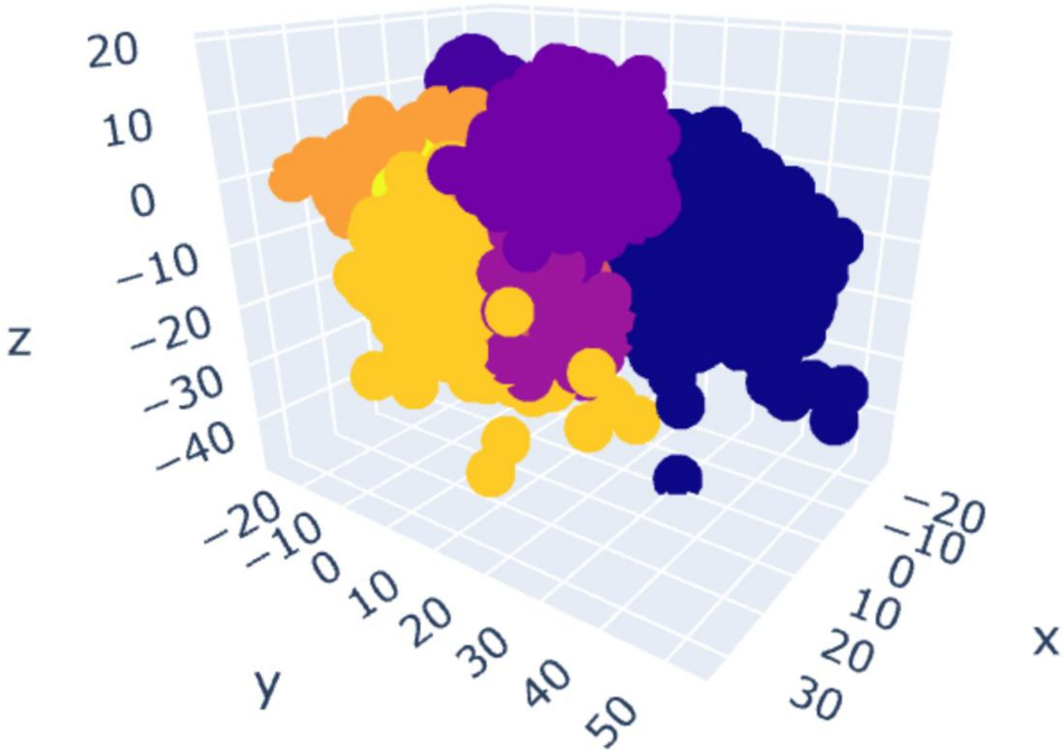
  We also don’t know ahead of time what structure our data will take in latent space!

AutoEncoder – Latent Space

Before training a network, latent space distribution is effectively random



After training the distribution of data within the latent-space is more strongly grouped & structured



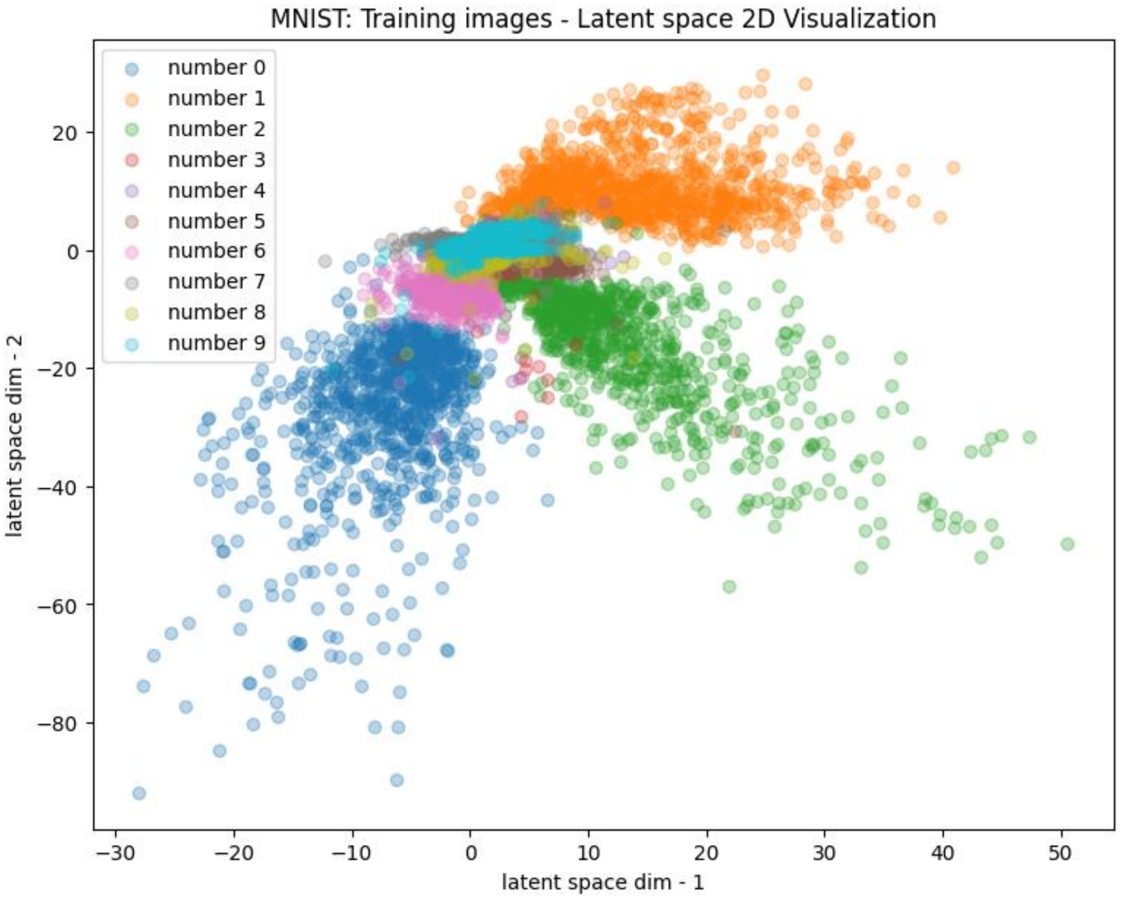
AutoEncoder – Latent Space

We want the latent space in an **AE** to allow us to share features between similar species.

e.g.

The numbers **9** and **4** are often ***similar***, so we expect them to potentially be close in the latent-space.
If this is the case our model has encoded most of the 'shapes' in a way that extracts out '**key features**'.

An overly-optimized or overly-constrained model may have these 2 numbers widely separated. In this case the model has learned about the 2 numbers, but not that they look similar.



AutoEncoder – Model uses

- **AE** models are some of the most widely used components in complex model design.

- This model design can be trained to:

  - Extract key features from a dataset for further analysis — ***Dimensionality Reduction***

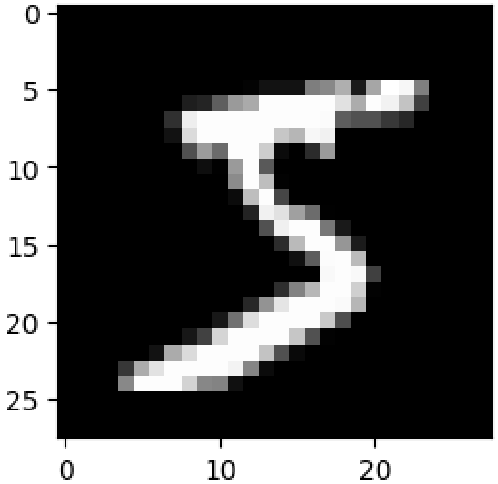
  - Regularize or pre-process raw data
  - Reduce noise on input data — ***Data Processing***

  - Supplement incomplete datasets for further use — ***Data Generation***

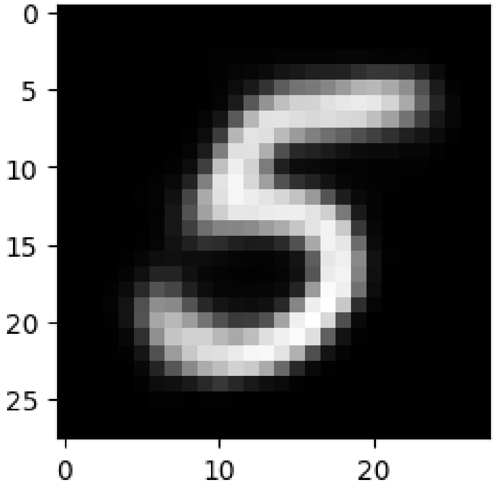
AutoEncoder – Noise Reduction

- One of the main uses of an **AE** network design is to perform noise reduction.
- Reducing an image to its key features and re-constructing it allows us to do this.

Input Image

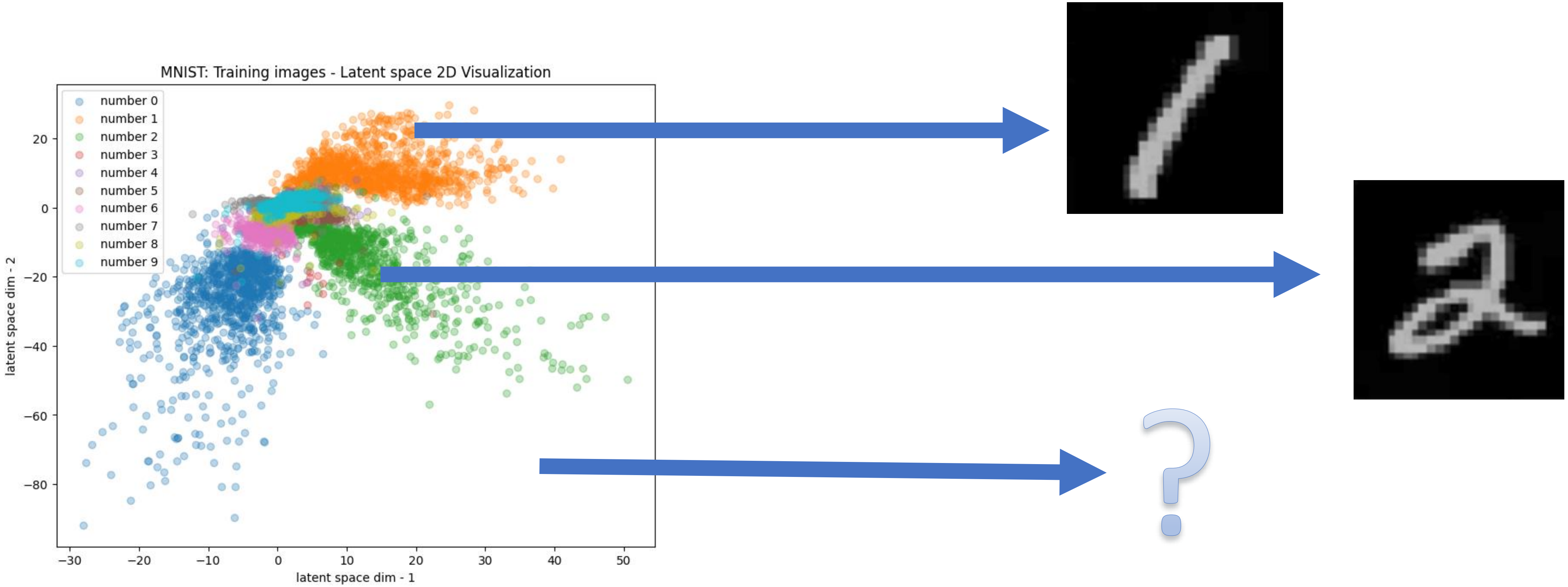


Re-Generated Image using
Latent-Space representation



AutoEncoders – As Generative AI models

- **AutoEncoder** models already have a **generator/decoder** component
- Problem with a pure **AE** is how do we generate reliable output?



AutoEncoders – Latent Spaces

- **Un-Supervised AE** model training ➔ Captured data features,

  Sampling **LS** generates random output

Used for generating:

image masks, random unlabelled data, denoised output, …

- **Supervised AE** model training ➔ Captured labelled data features,

  Sampling **LS** generates semi-structured output

Used for generating:

augmented data, labelled data, better de-noising, …

AutoEncoders – Latent Spaces

- **AE** model **latent-space** is normally ***completely random***, the structure changes as the model changes...

  NB:

  Generated by black-box and then used as input to a 2nd black-box.

- **VAE** models are a modified **AE** design and intended to address and partly fix this problem.

- Main difference is that the encoder is forced to encode information onto a **probability-space**.
- This allows us to train a **VAE** model with a decoder component to ***generate new output*** based on this **probability-space**.

Variational AutoEncoders

- **VAE** model training is not the same as **AE** model training.

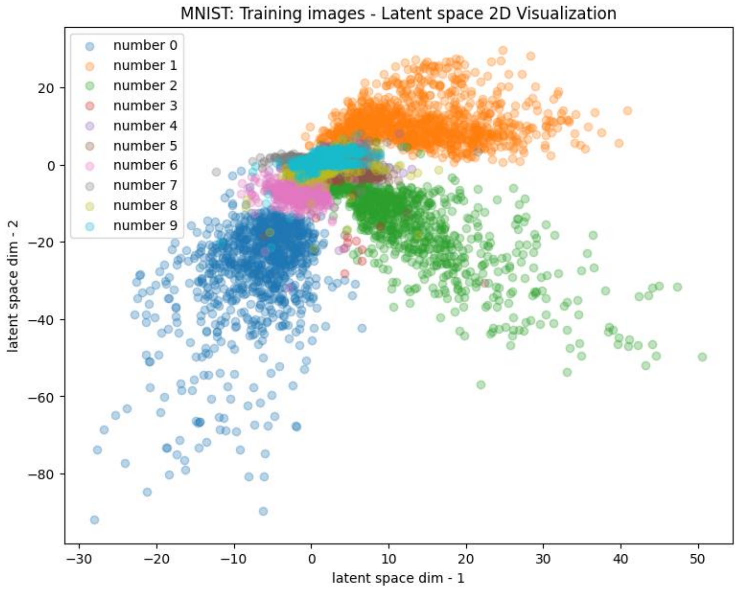
- **AE** model training makes use of a loss function typically comparing the output from the generator to the input using a loss function such as **BinaryCrossEntropy** or equivalent.

- **VAE** places do the same with an additional requirement:

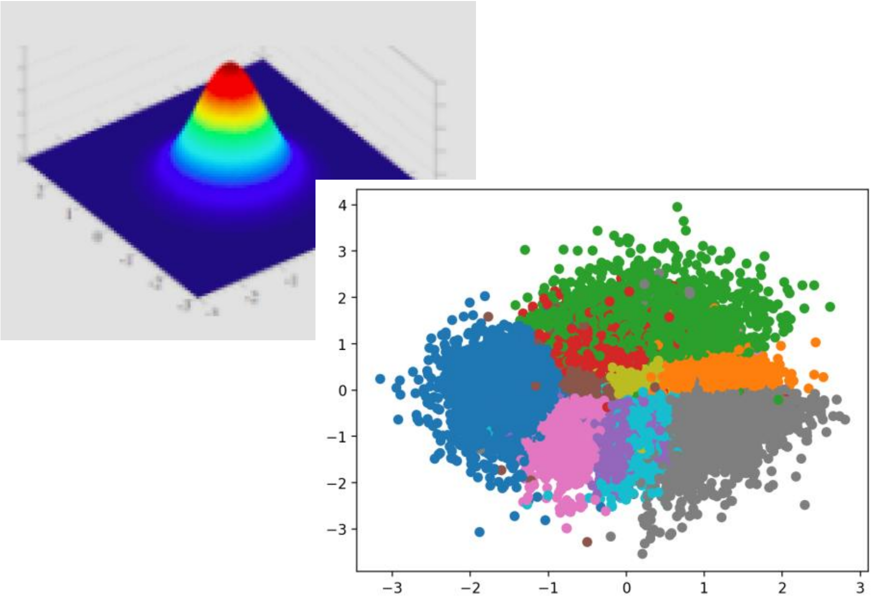
  "***We want to encode latent-space vectors onto a probability space***".

Variational AutoEncoders

What AE have:



What VAE want:



To get a probability distribution in latent space, we need to know “how far we are” from this ***ideal*** distribution.

Kullback-Leibler Divergence: $D_{KL}(P||Q) = \sum_{x\epsilon\chi} P(x)log\left(\frac{P(x)}{Q(x)}\right)$ where x is within the latent space χ .

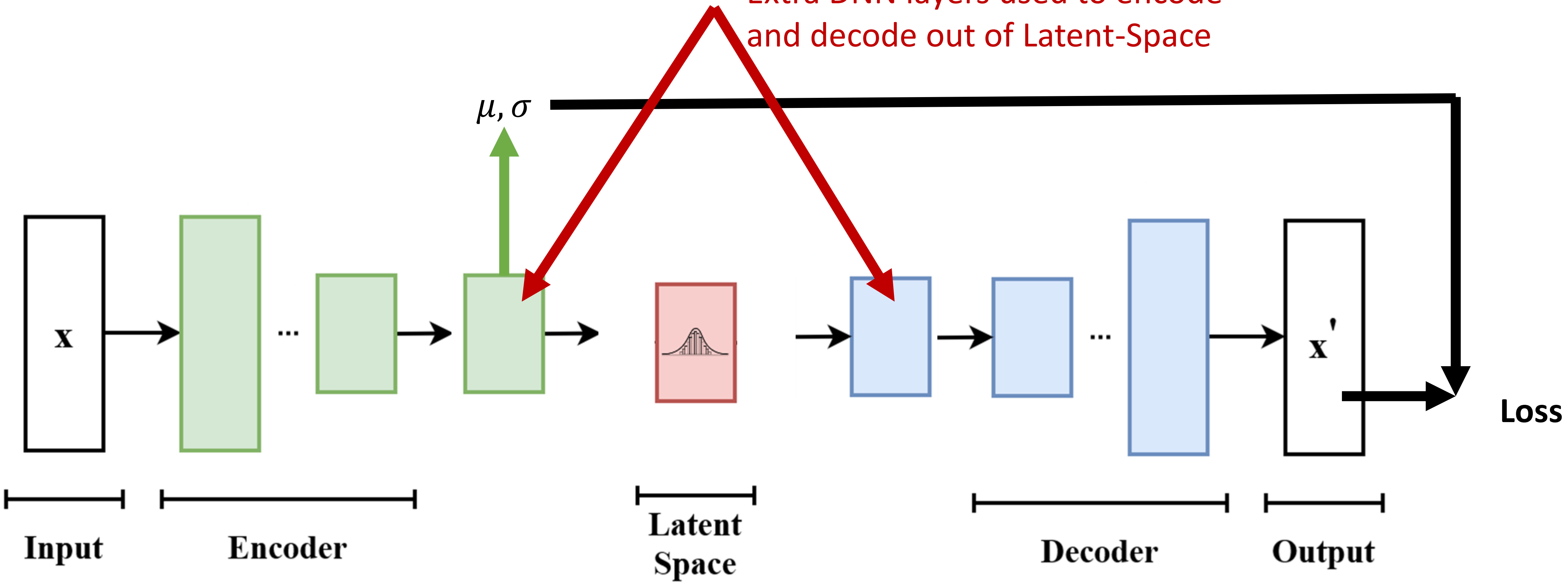
This is a measure of the entropy/information difference between the distributions of $P(x)$ and $Q(x)$.

Loss Function for training: $\boldsymbol{Loss_{KL}} = \sum_{j=1}^{J}\frac{1}{2}\left[1+\log(\underline{\sigma_i^2}) - \sigma_i^2 - \underline{\mu_i^2}\right]$

Excellent guide how to get from KL-Div to loss function: https://arxiv.org/abs/1907.08956

Variational AutoEncoders

VAE model is a modified compared to raw **AE** model with extra values from the encoder fed into **Loss** function.


Extra DNN layers used to encode and decode out of Latent-Space

Variational AutoEncoders – Loss functions

- The implementation of a VAE in different ML frameworks can be difficult.
- Typically, best approach is to use the functional API in TensorFlow which is like the API used by PyTorch.

- Amending the loss function requires adding additional term(s), i.e:

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

- Extracting the required parameters $(\boldsymbol{\mu}, \boldsymbol{\sigma})$, requires modifying the model so that these parameters are available to the Loss functions.

CNN – Other Applications

- Other examples of **CNN** usage is in anomaly detection.

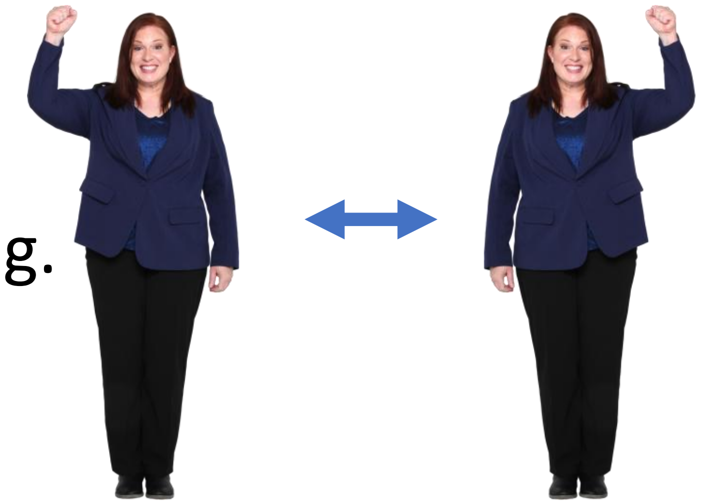
- This relies on understating what the latent-space of a trained model looks like.

- This relies on the fact that a trained model will give an unexpectedly large value in this space for an anomalous datapoint which isn’t described well by the dataset the model was trained on.

CNN – Model Training (1)

- CNN or CDN(N) models are complex models which still have many free parameters to be optimized when training on a given dataset.
- This means that datasets that give the best results for these models we want the largest datasets possible.
- In the case that the input data is translationally invariant we can apply translations on input images to make sure that our model doesn’t over-optimize.

CNN – Model Training (2)



- Imagine a dataset was composed of people waving.

  People tend to wave with their dominant hand.
  ~90% of people are right-handed.

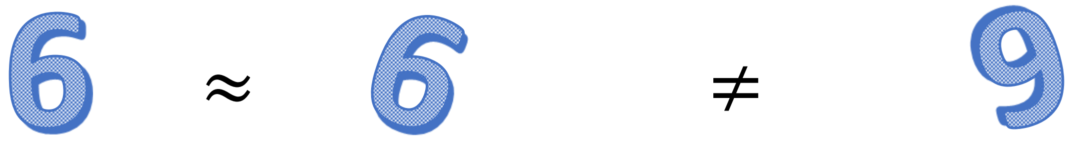
  Model might assume that only right-handed people wave.
  (or that people waving are right-handed…)

- Given that a person in a mirror still tends to look like a person.

  We can translate our data randomly to avoid this bias.

CNN – Model Training (3)

- In the case of certain models and certain datasets invariance is a good and desirable characteristic.
- However, if we're looking to build automatic number recognition for documents scanned with the *correct* orientation…



- Translations are good & useful when used correctly, but they can also cause problems when not checked.

Training – Quality of a fit

- When training our model, we want to make sure that we are avoiding over-training on features only present in the training data.

- We also want to not waste computing resources over-training a model which has effectively converged.

- The best way of doing this is to split our data into 3 datasets as with Decision Trees:

Training, Validation & Testing

Training – Model Design

In the workshop following this lecture we will be building models which are designed to work with hundreds of input parameters.

In turn the **DNN** designed to work with this uses hundreds of thousands of free weights.

If it is possible to reduce the number of free weights by being clever in our nodes, then the model is mathematically simpler to work with.

This typically means that we need to do something more complex in our node than just ‘summing’ over the products…

Training – Subtleties and Pitfalls

Training a model to data can have additional problems other than just if a fit converges.

Remember the training dataset may be large *but it is still finite*.

It can suggest features which aren’t apparent in the real world, and it can mean that not all cases in the real world are seen by a model during training.

To understand this better we want to know about the quality of a fit, as well as if the model has potentially been over-trained on the training dataset in an unhelpful way.

Training – Over-training

When training a model to data we also run the risk of over-training on our input dataset.

This can be problematic as the model now likely doesn’t represent **reality**.

How can we prevent this?

- No good single answer to this problem.
- Training a model isn’t, exactly, the same as extracting optimal physical parameters needed to describe a dataset. *
- When we have enough data, we can set aside a fraction as a validation dataset to determine if we have over-trained.

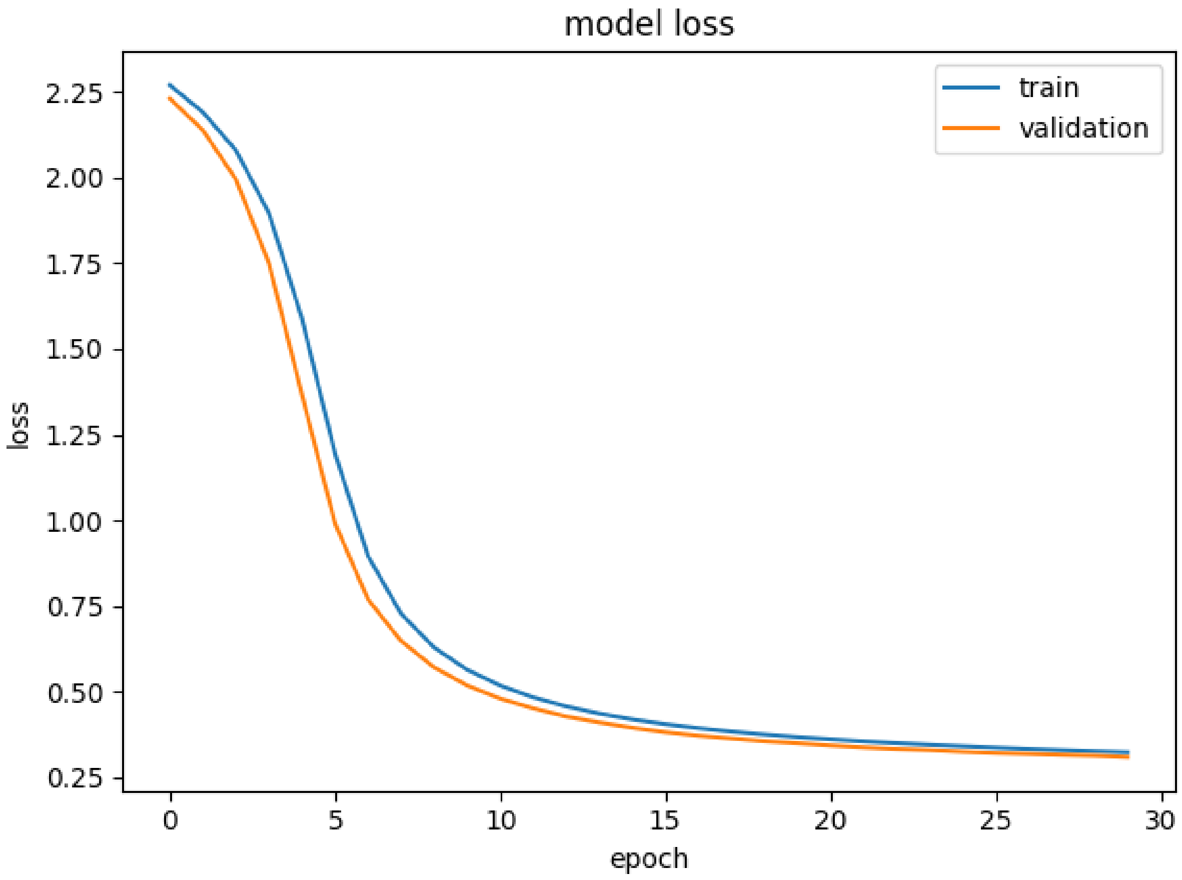
*In Physics analyses you would often use measures such as $\chi^2/nDoF$, here, our DoF is potentially millions of parameters…

Training – Over-training – Validation DataSet

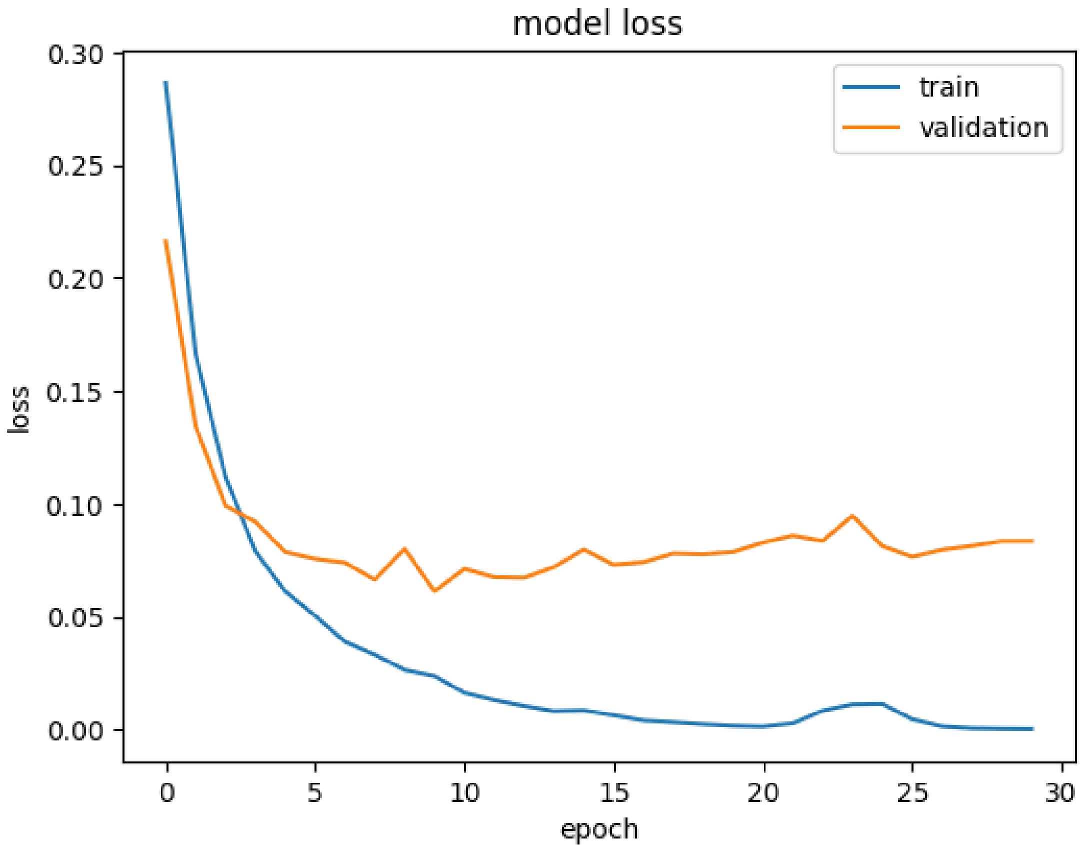
- Using a **validation** dataset is relatively trivial in principle.
- Set aside a representative fraction of the training dataset.
- Train your model on the large training dataset and use the smaller independent dataset to determine how well the model generalizes and/or behaves.
- This is done by comparing the behaviour of the loss function and accuracy of the model on both.
- The **Validation** dataset *may* be used to help “tune” model hyper-parameters.

Training Using Validation Dataset

Good loss function behaviour during training

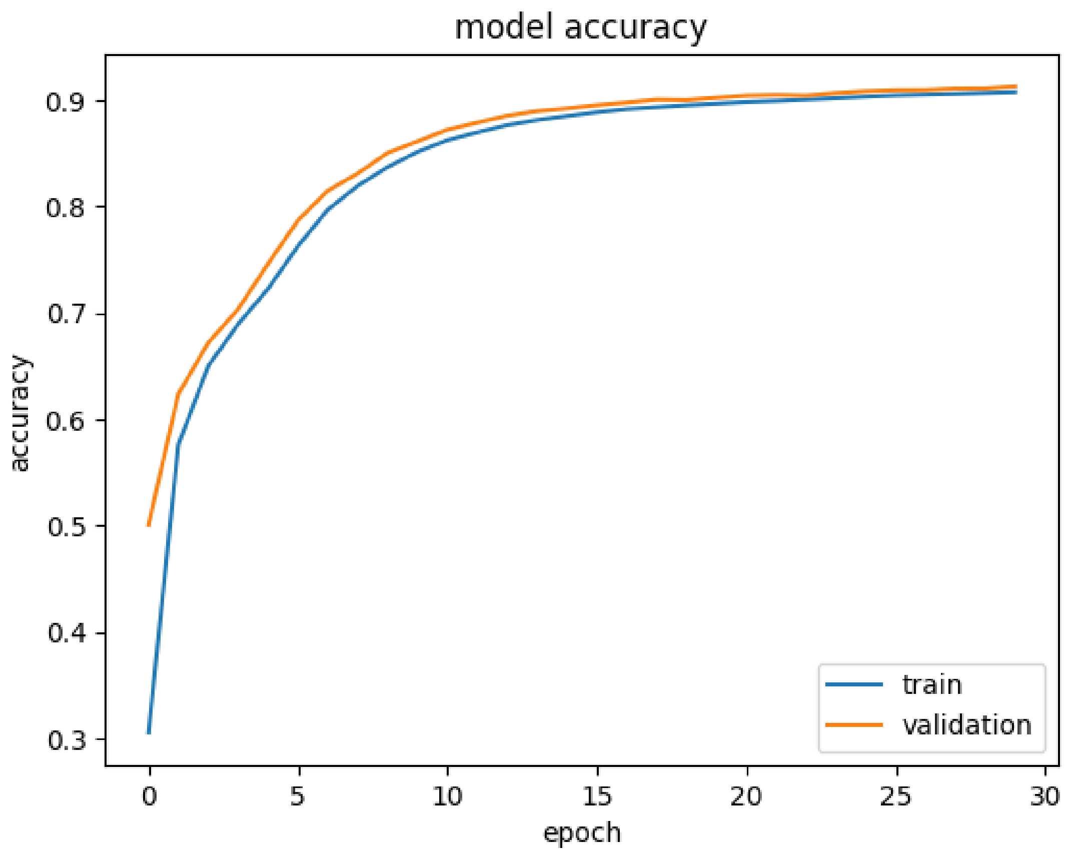


Bad loss function behaviour during training

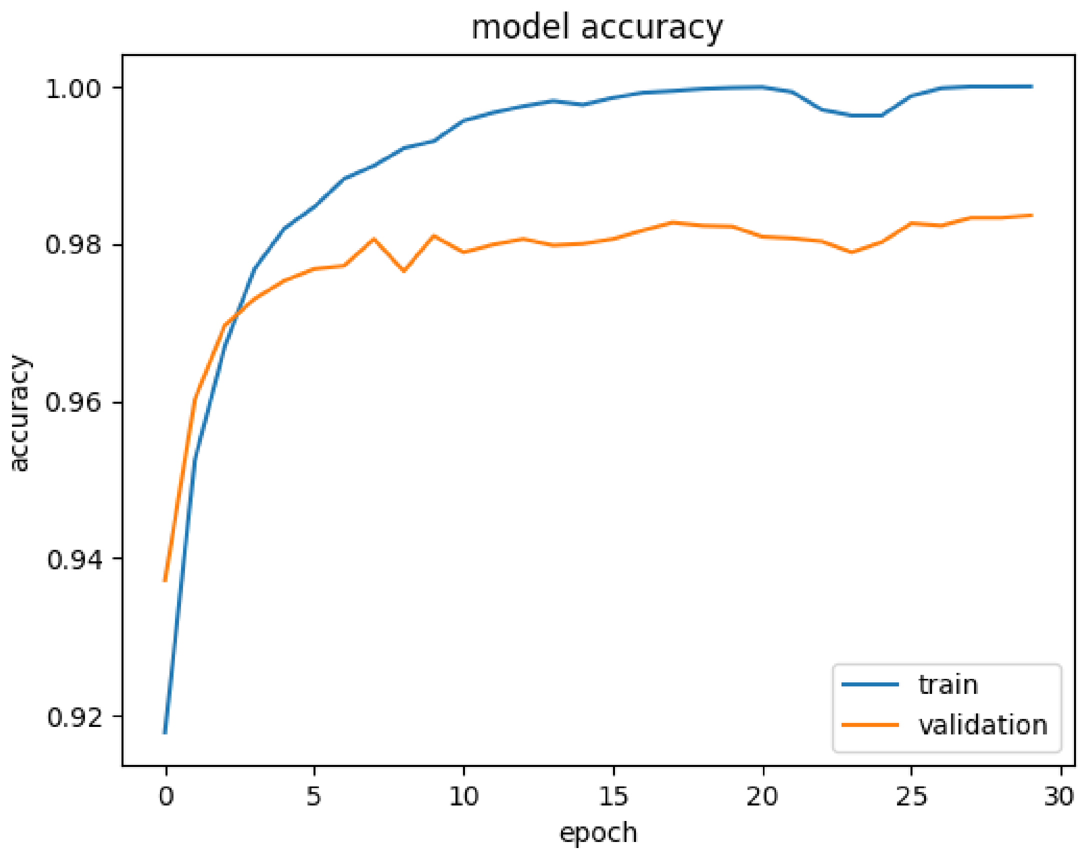


Training Using Validation Dataset

Good model accuracy behaviour during training



Bad model accuracy behaviour during training



Training – Using a Test Dataset

- A **Testing** dataset is typically only ever used once.
- This is extracted in the same way that the same as a **Validation** dataset.
- The main difference is that this dataset is only ever used **once**!.
- It is **not supposed to be used to help tune model hyper-parameters**.
- It is to try and get a “true measure” of the model performance **on unseen data**.

Training – Classification model

This Photo by Unknown Author is licensed under CC BY-SA-NC

- Another way of determining if a model has trained correctly or not is to consider the errors we get from training.

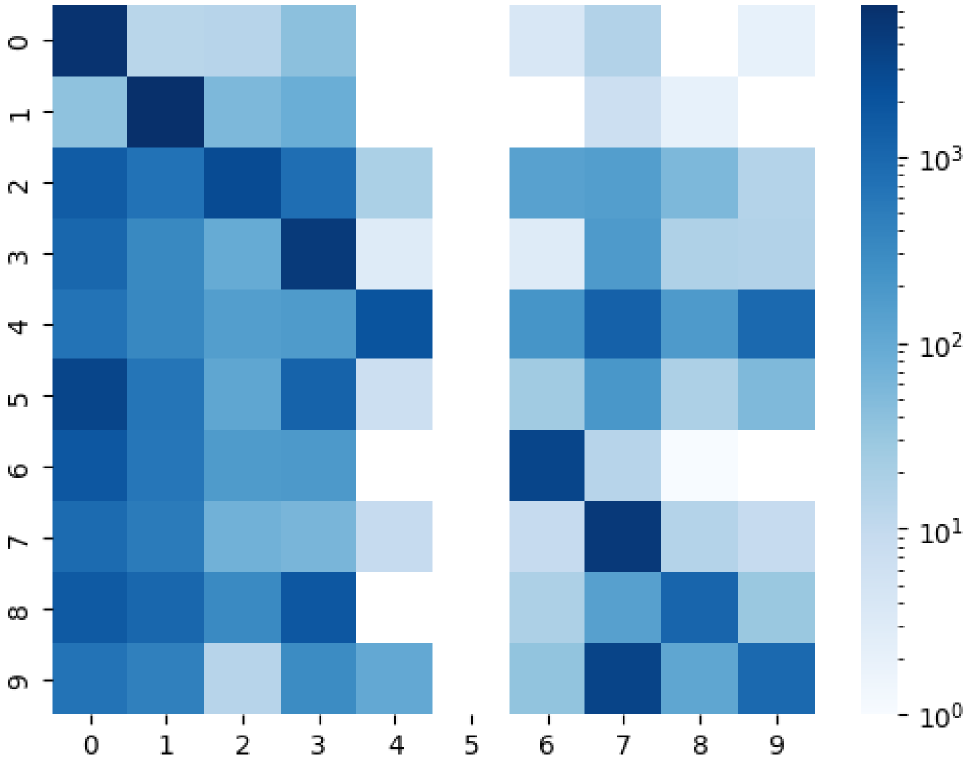
- The ***Error*** or ***Confusion*** matrix.

- This matrix allows us to examine a model and determine how good it is at separating features in each dataset.

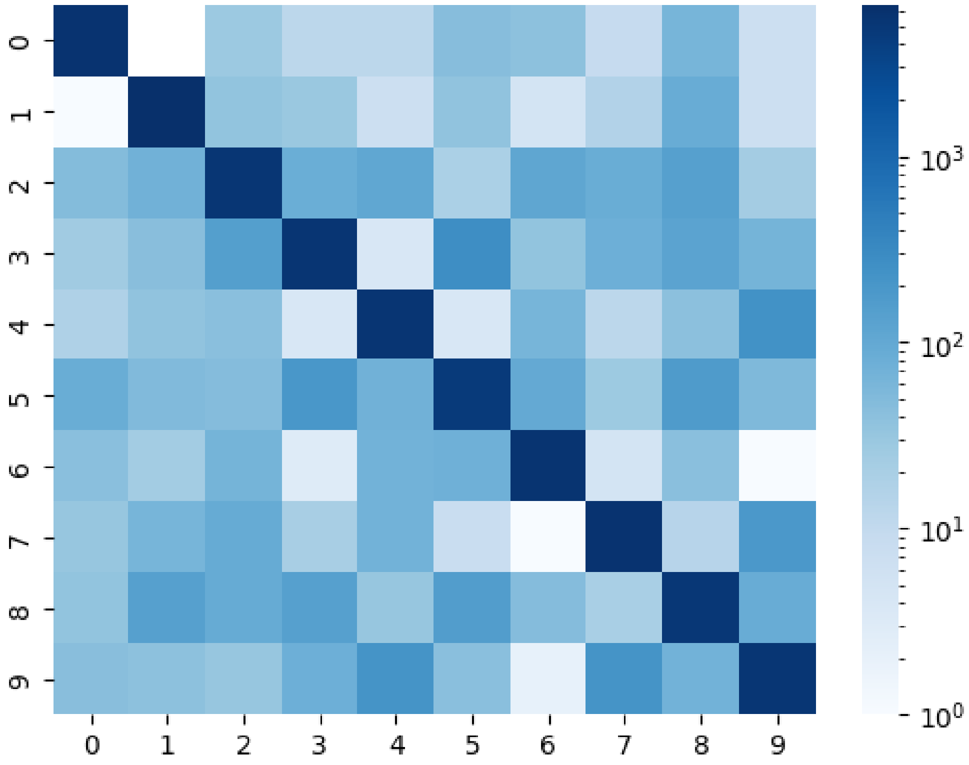
i.e. how many dogs look like cats,
how many background events of type X look like signal,
how many stars of type Y have I mis-classified as Z, …

Training – Confusion Matrix

Not so good



Close to ideal



Training – Model design

This Photo by Unknown Author is licensed under CC BY-SA-NC

- When you’re training a model, you want to avoid problems such as ***vanishing gradients***.

- To try and avoid this happening in training several strategies and layers have been developed to mitigate this.

- In general, most of these strategies try and avoid weights getting too small/large or the gradient getting too large.

Training – Vanishing Gradients (1)

Vanishing gradients is one of the oldest problems when training/fitting to data.
The problem in some ways is quite simple.

For a **DNN** with many layers, as you begin to approach a minima you find features which are detected by earlier neuron layers don't survive to later generations.

Problems such as this can be impacted by many parameters such as:

- **Limited Numerical Accuracy**
- **Choice/Use of Activation Function(s)**
- **Number of nodes in a layer**
- **Value(s) of inputs**
- **Current/Starting values of weights in the DNN**
- **Distance to minima**
- …

Training – Vanishing Gradients (2)

There is no general solution to the problem of vanishing gradients.

Vanishing gradients are also not the only problems which become apparent during fitting.

This can present itself in different ways such as:

- Weights in later layers vary strongly whilst earlier layers adjust slightly
- Training fails to converge in a sensible time period
- Large difference between values of weights in fit
- Large amounts of model weights become supressed to zero

Training – Advanced Techniques/Strategies

There are 2 common approaches which are used to reduce the risk of encountering a vanishing gradient problem on data.

Adding **Biases** to neurons.

Advantage of this is that more features survive to later layers of nodes at the cost of adding a few additional free parameters.

This is allowed because it can't adjust the gradient of the loss function moving backwards.

Regularizing Weights.

This approach introduces some regularization into neurons.

The impact on each weight is expected to be random during training.

Training – Adding Biases

Biases are added to neurons within an ANN.

Adding a bias this way simply increases the total value of the weight in the neuron.

Adding a constant varying term to each neuron doesn’t negatively impact training as this is a constant term compared to the neuron inputs.
This is important as it therefore disappears when using back propagation to train a model.

Training – Regularization

There are 2 approaches to **Regularization** **L1** and **L2**.

L1:

L1 Regularization (Lasso): Adds a penalty to the loss function based on the **absolute value of weights**, encouraging **sparse models** by driving some weights to **exactly zero**.

L2:

L2 Regularization (Ridge): Adds a penalty based on the **squared magnitude of weights**, discouraging **large weights** while allowing small, non-zero values, promoting **weight stability**.

Regularization can bias the possible model weight values.
This can be either desirable or very bad depending on your model.

Training – Normalization

One of the concepts gaining popularity again in modern ML network design is **(Layer)Normalization**.

Typically, this is applied through a normalization layer in the model which scales the output from a previous layer before the values are passed to the next layer.

This is a key part of our **VAE** model we will be building in our workshop today.

Workshop – Today

- Todays workshop we will investigate using and understanding CNNs to build a **Variational Auto-Encoder**.

- Like a **DNN** classifier we will train this over the ***mnsit numerical dataset***.

  Partly for simplicity and partly for speed, you will see how much computational power is needed to get these models to converge. *(>10min on mid-tier consumer GPU!)*

- I am providing a **pre-trained VAE** model which may be more useful when exploring the model outputs rather than training your own large model.