Convolutional Neural Networks

Robert Currie

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 often *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 input(s)

- 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 5x10^4 weights or **~2x10^9 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.

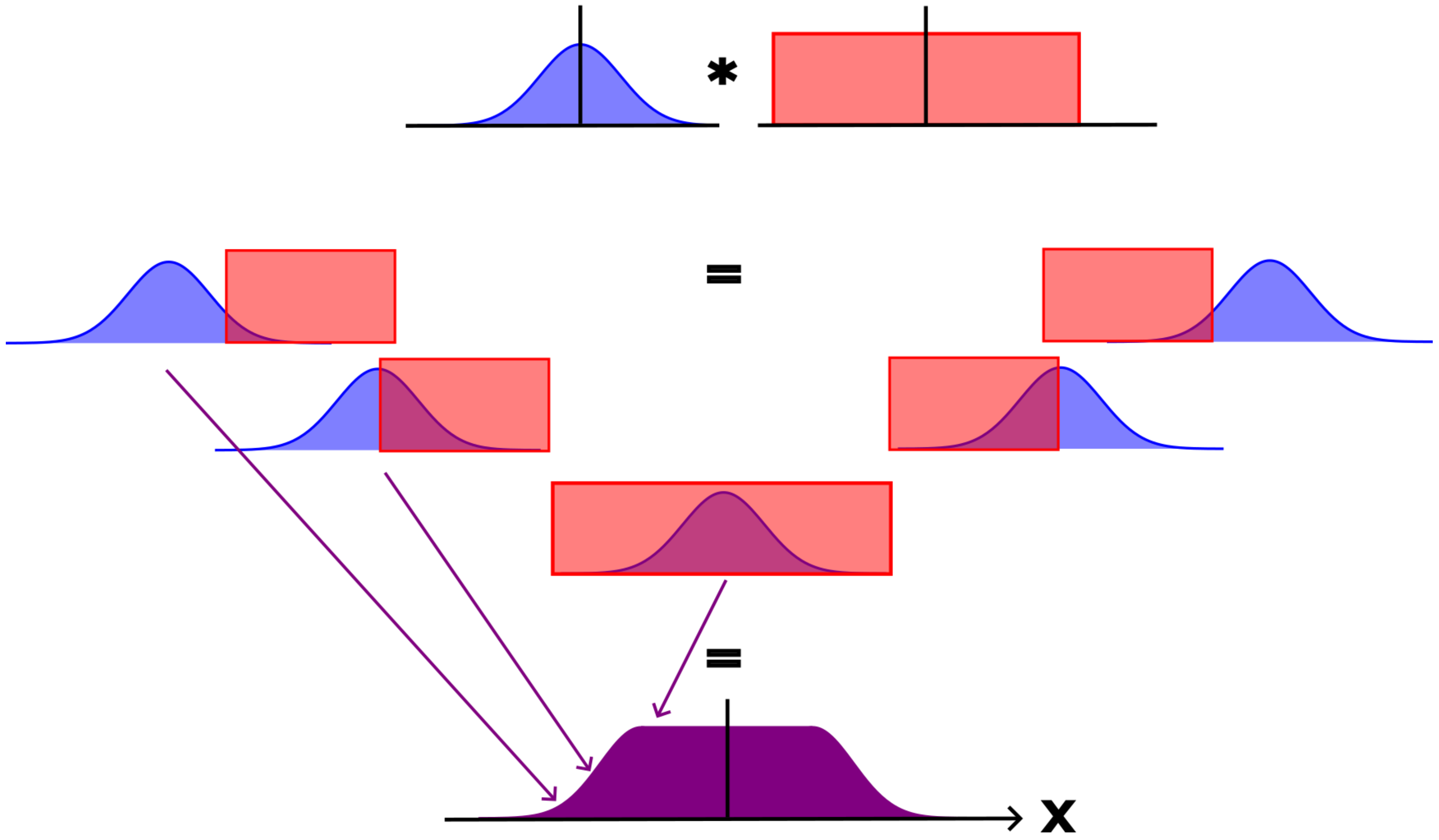
i.e. $Input * \; Operator = Output$

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

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

Convolutions – 1D example

$(f * g)(x)$



Convolutions – 2D

The 2nd dimension!

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

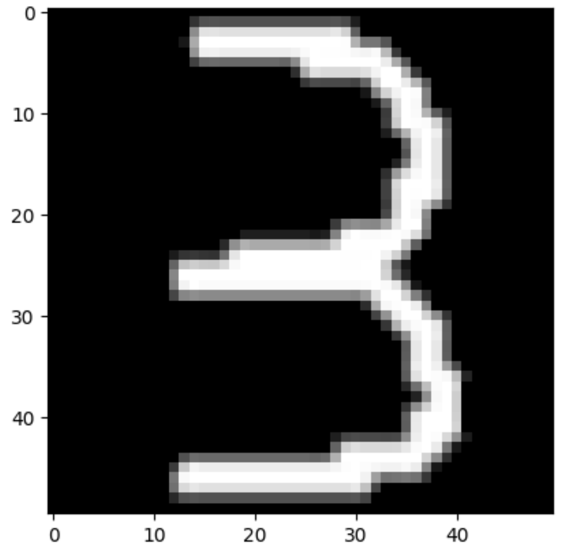
One of the most common 2D functions to demonstrate as I've mentioned is the Sobel Operator*(s)*.

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

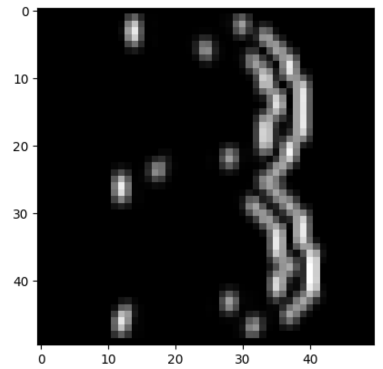
$$\mathbf{G} = \sqrt{\mathrm{G_x^2} + \mathrm{G_y^2}} \qquad \Theta = \mathrm{atan}(G_y, G_x)$$

Convolutions – 2D example

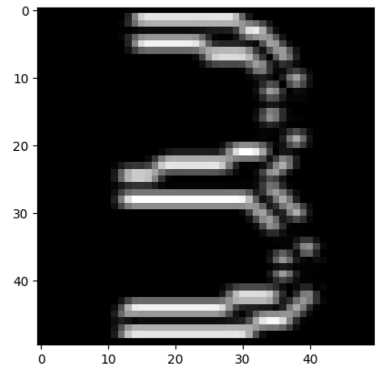
Input=



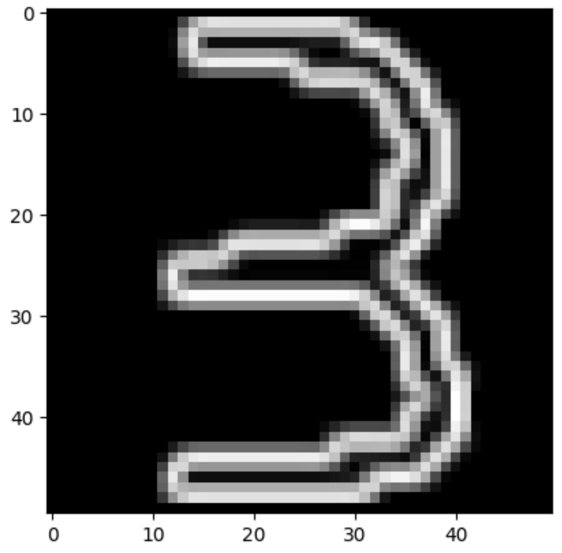
$\mathrm{G_x} * Input =$



$\mathrm{G_y} * Input =$



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



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.

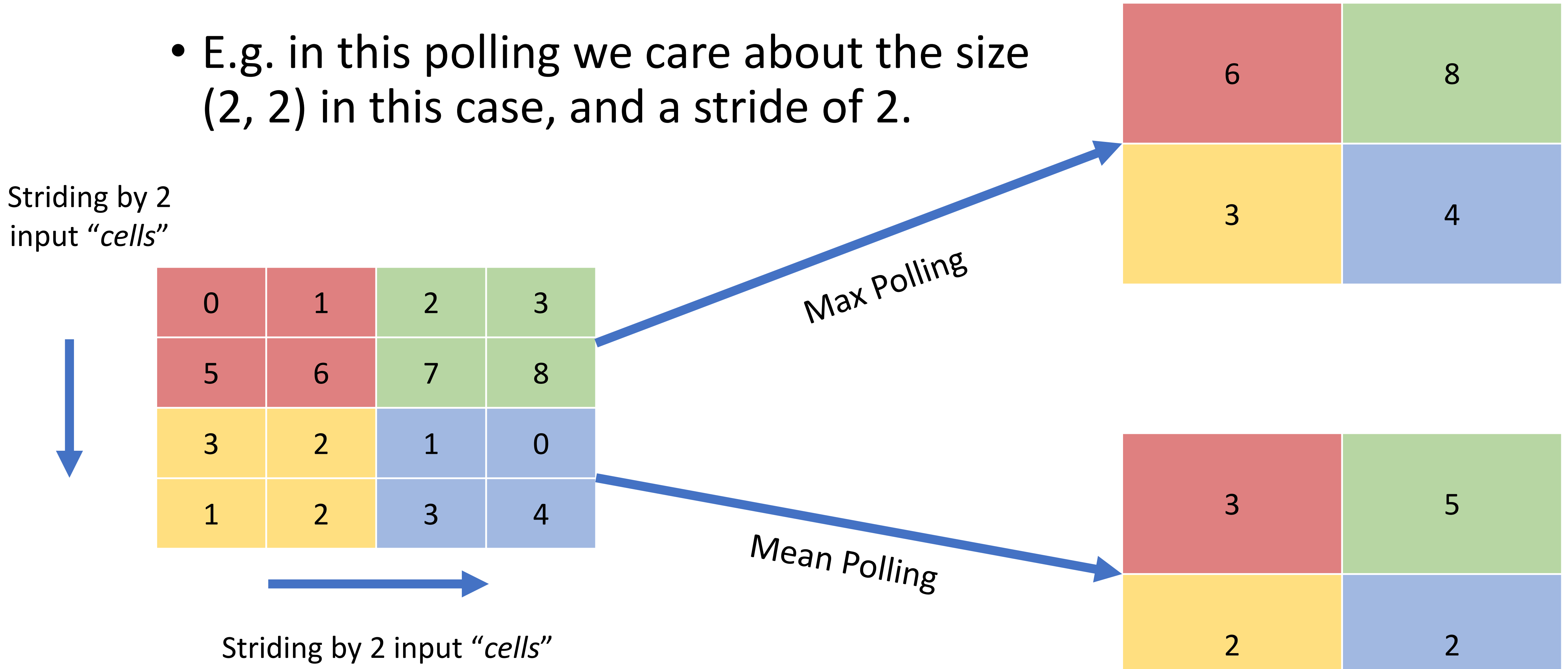


Image Manipulation – Common Nomenclature

- **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

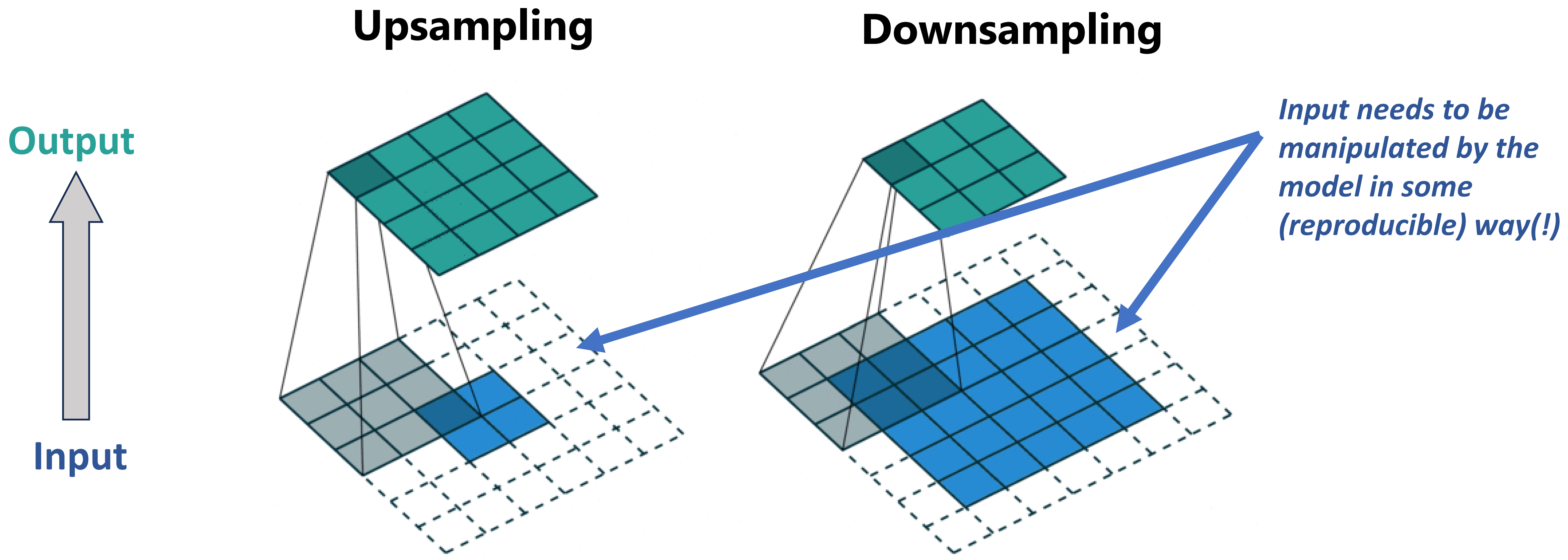
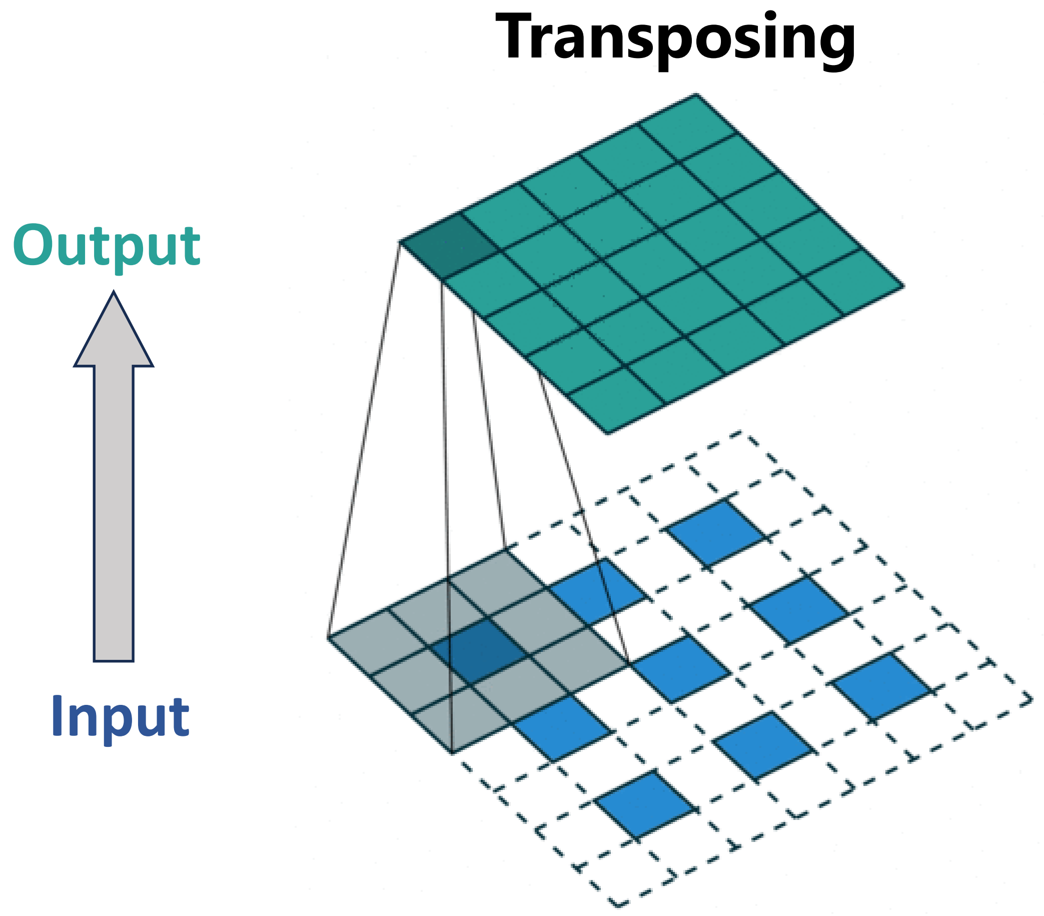
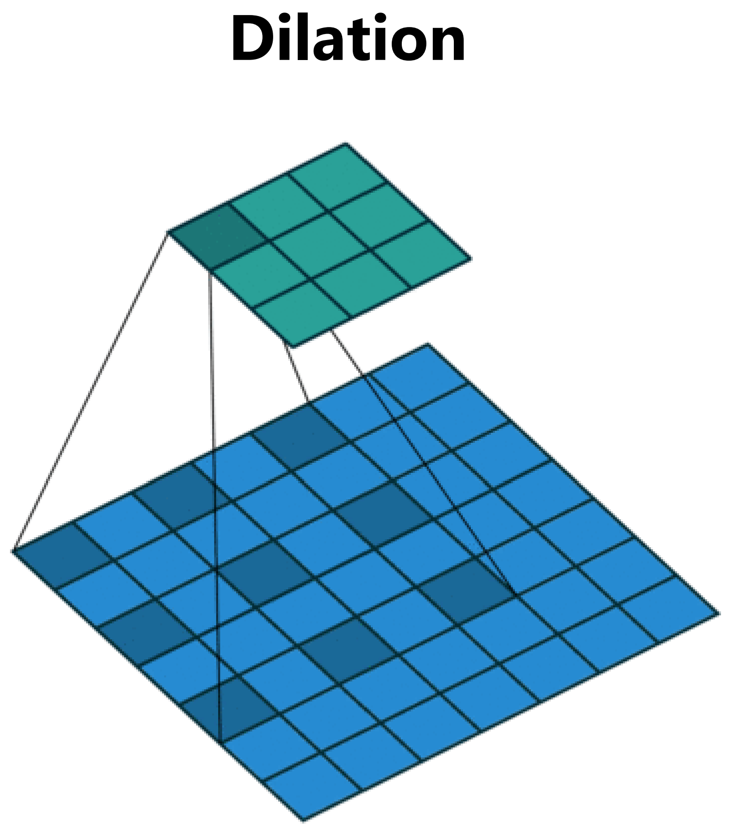

Upsampling
Downsampling
Output
Input
Input needs to be manipulated by the model in some (reproducible) way(!)

Image Manipulation – Common Nomenclature

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

https://github.com/vdumoulin/conv_arithmetic





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 aka 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.

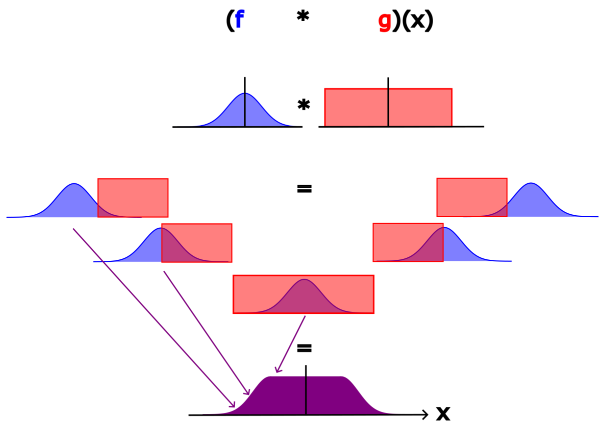
Moving from linear maths to matrix maths.

Matrix of weights

Input Matrix of values

$$\begin{array}{c}\mathit{Neuron}\\ \mathit{Output}\end{array} = \mathit{Activation}\big(\mathit{Conv2D}(\mathit{Weights}, \mathit{Input})\big)$$

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

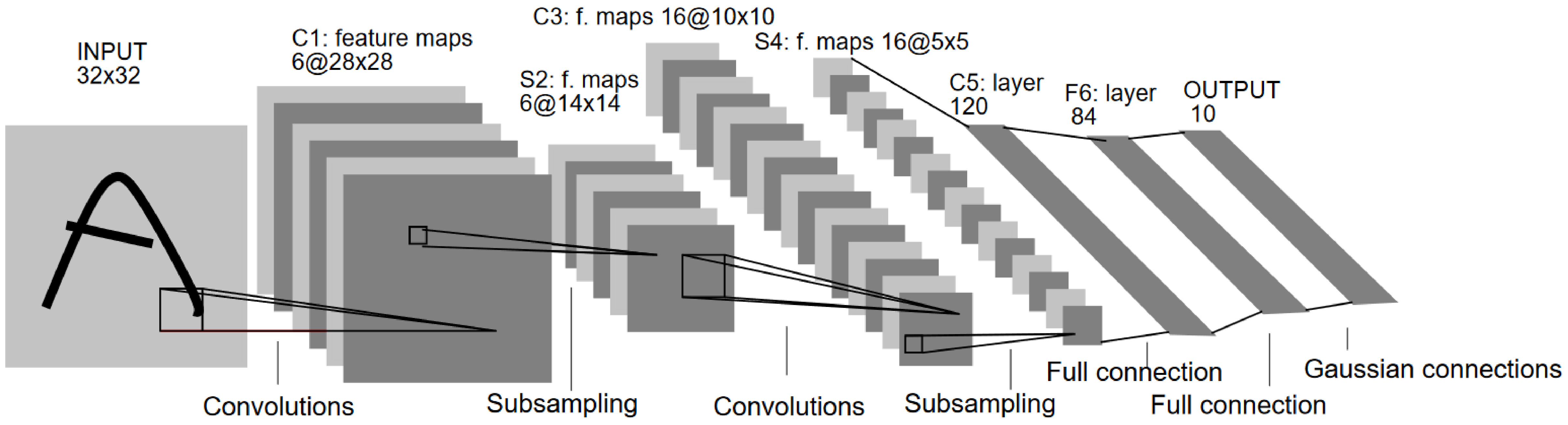
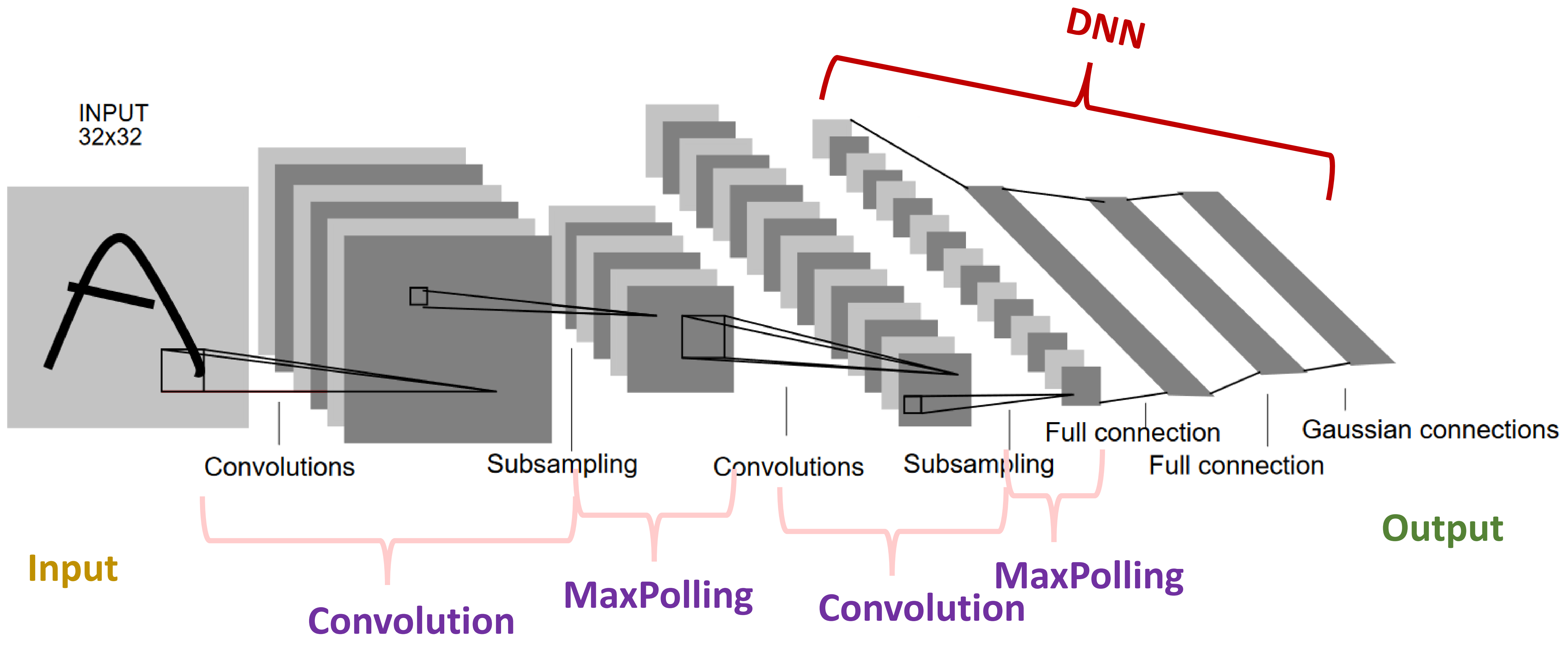


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

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.

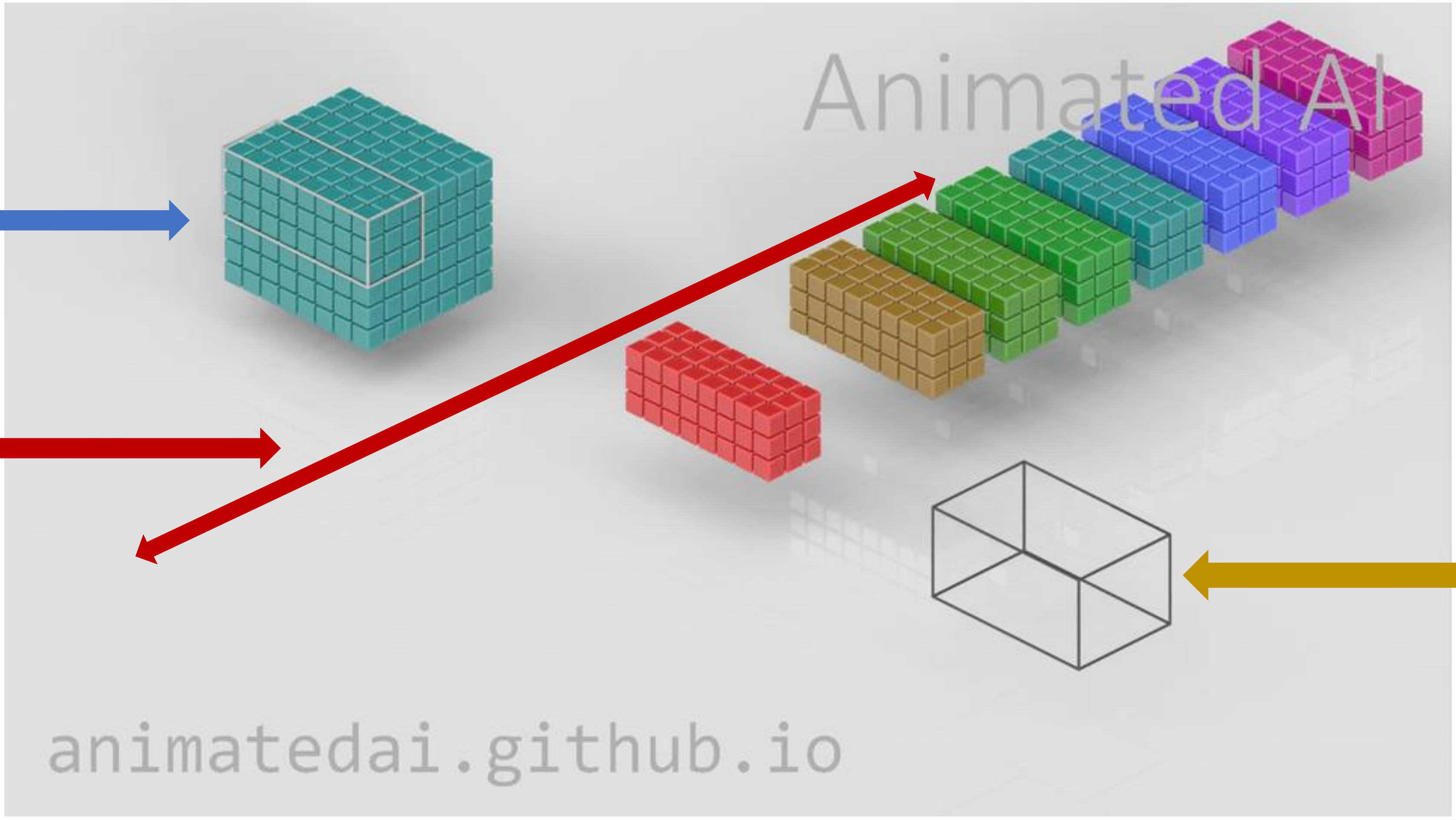
  E.g. *4k image may be reduced to 256x256x1 inputs after extracting features…*

CNN – Network Design

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

https://animatedai.github.io/

Layer INPUT

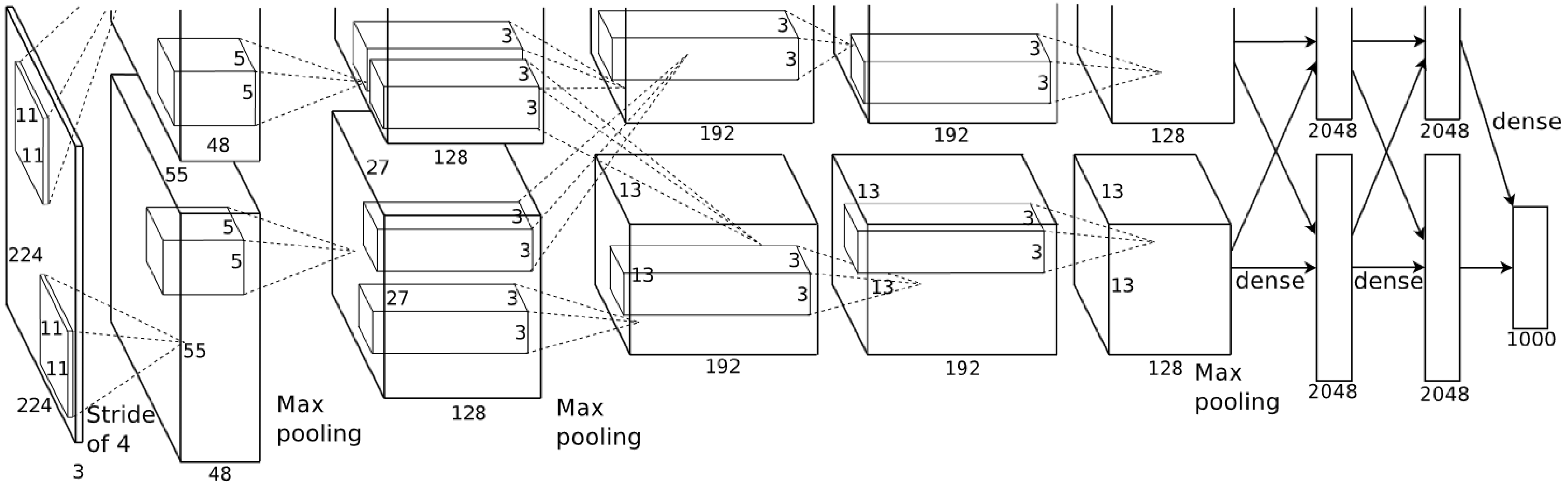


Trainable Model Matrices

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**.

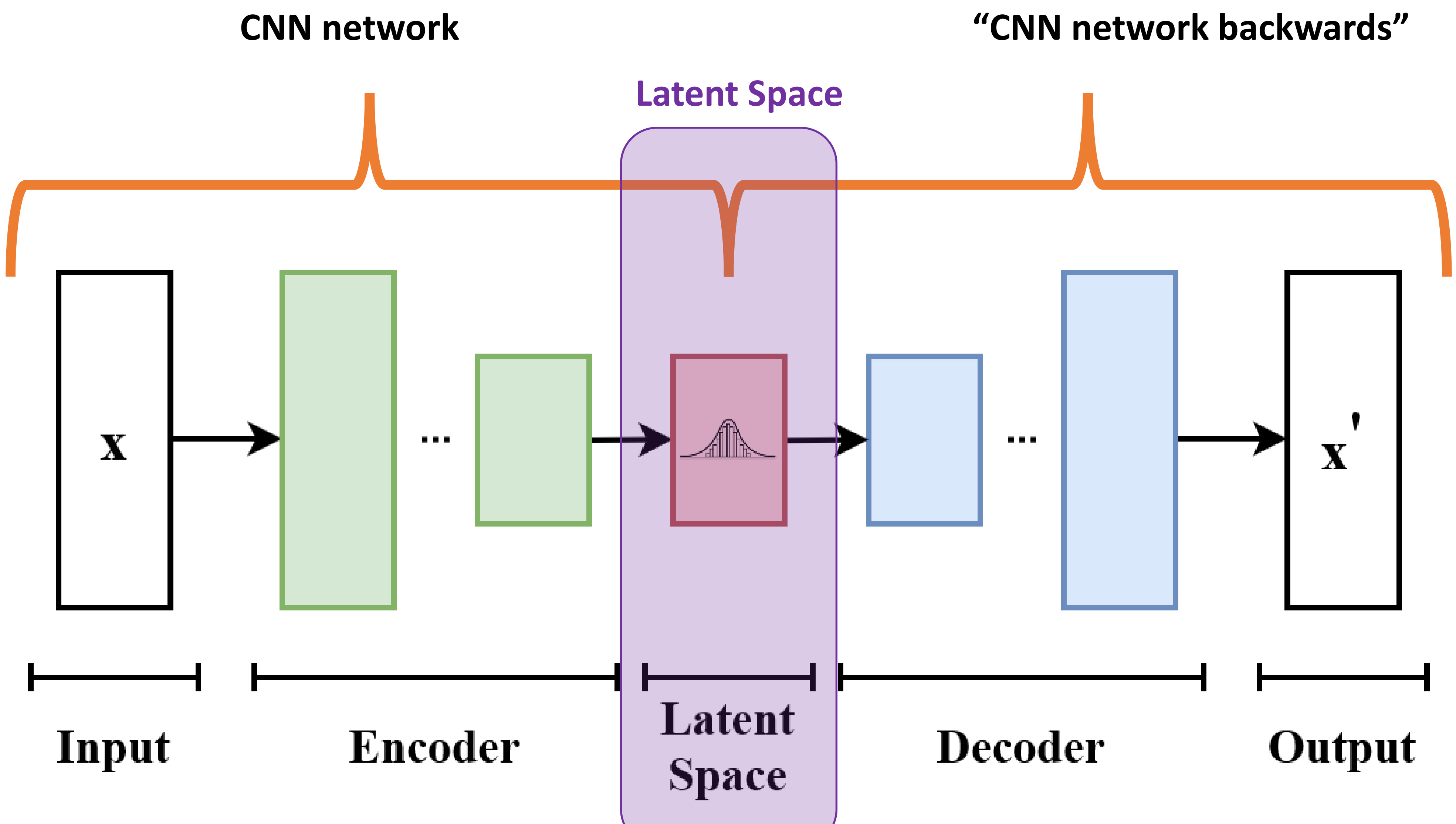
Uses of CNN networks

- 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.

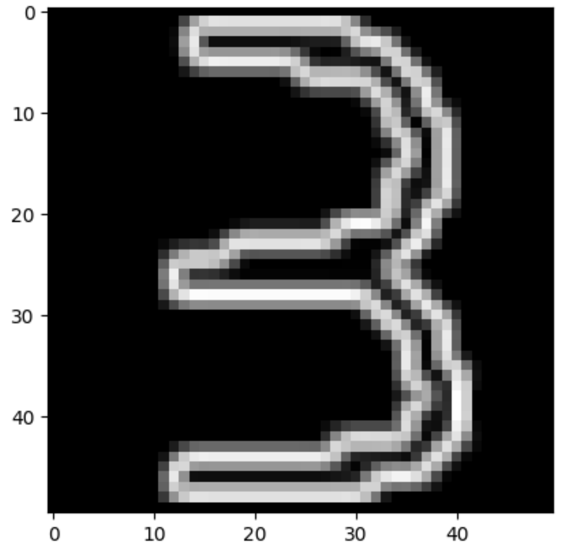
AutoEncoder – Network design



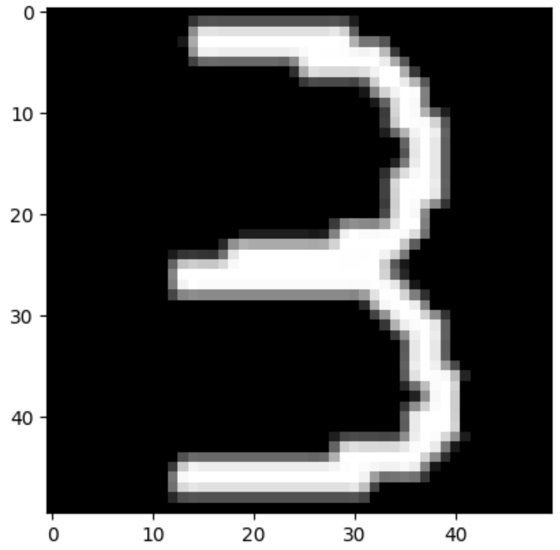
Conv2DTranspose – An Aside

- 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 ☺



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



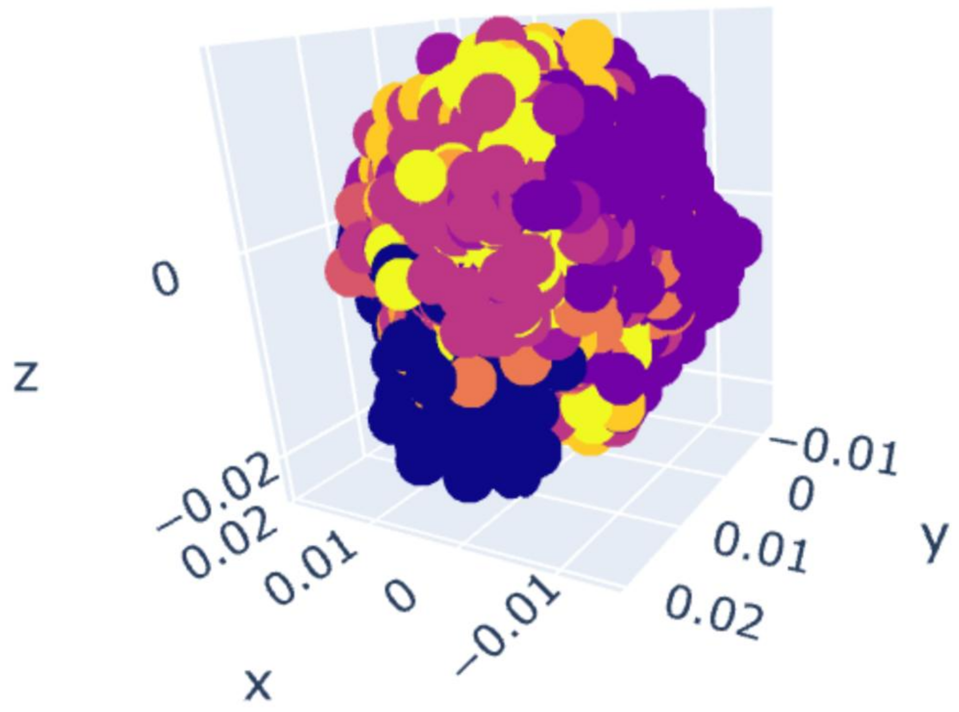
AutoEncoder – Latent Space

- 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.

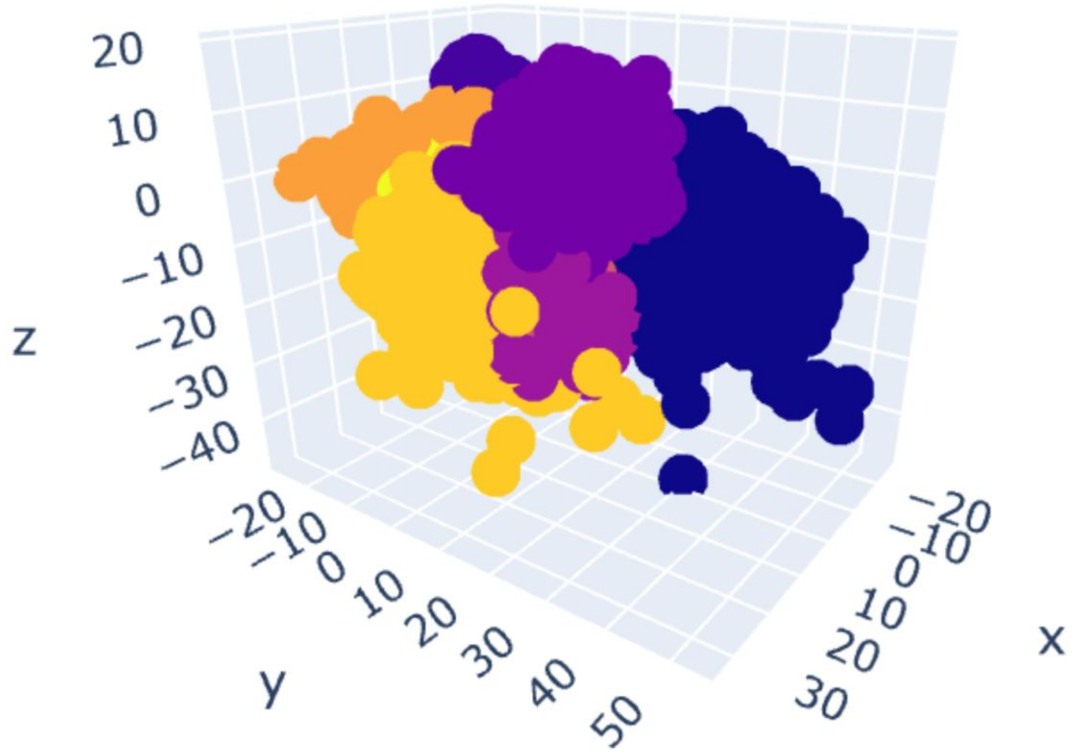
  We also don't know ahead of time what structure our data will take in latent space!

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



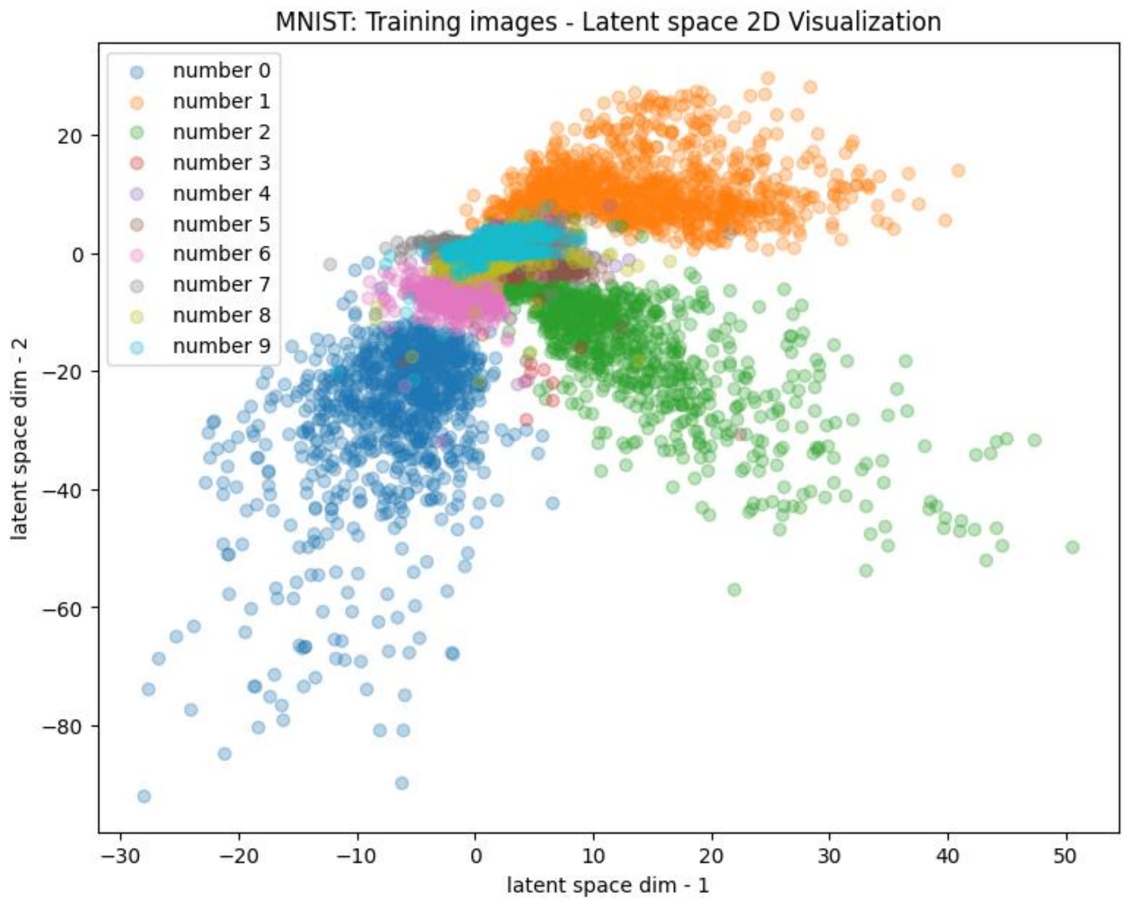
AutoEncoder – Latent Space

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.

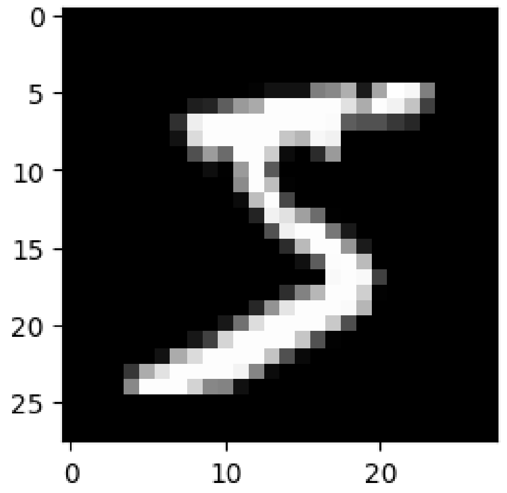
- This model design can be trained to:

- 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***

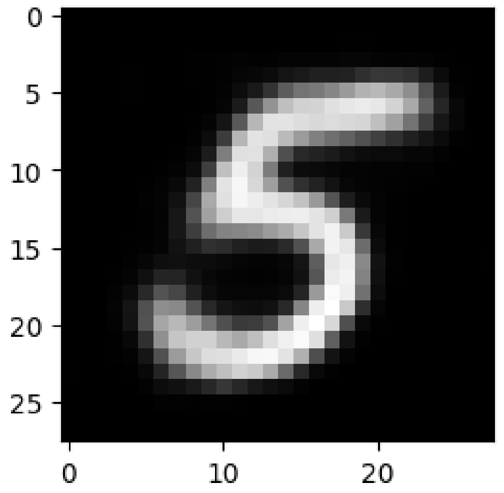
AutoEncoder – Noise Reduction

- 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

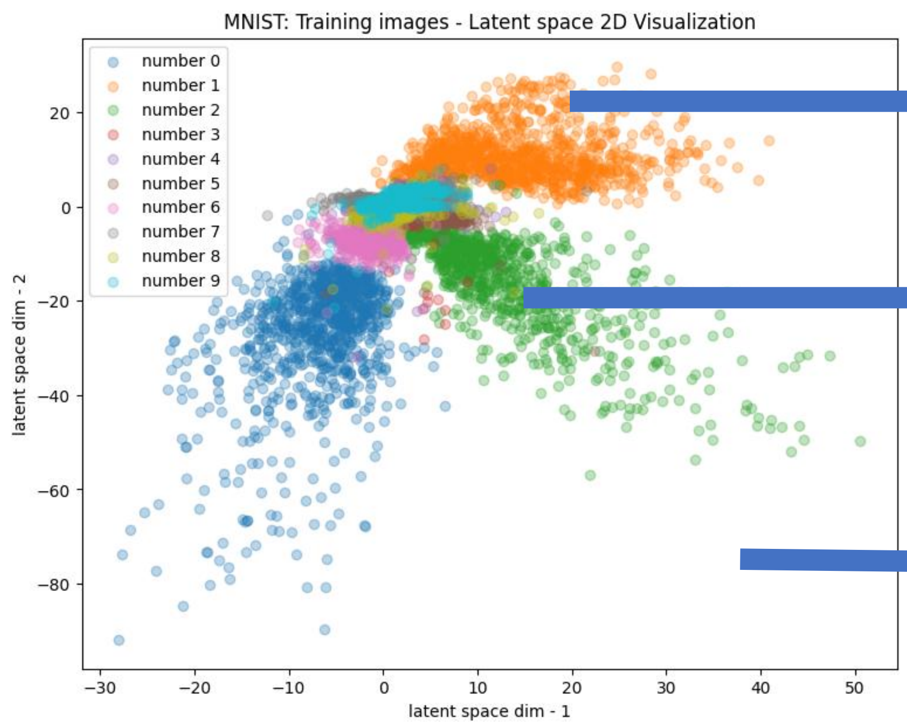


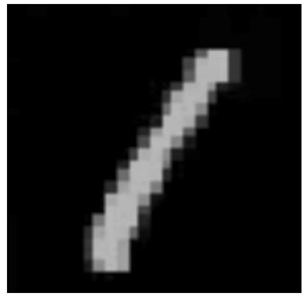
Re-Generated Image using Latent-Space representation



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?





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

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.

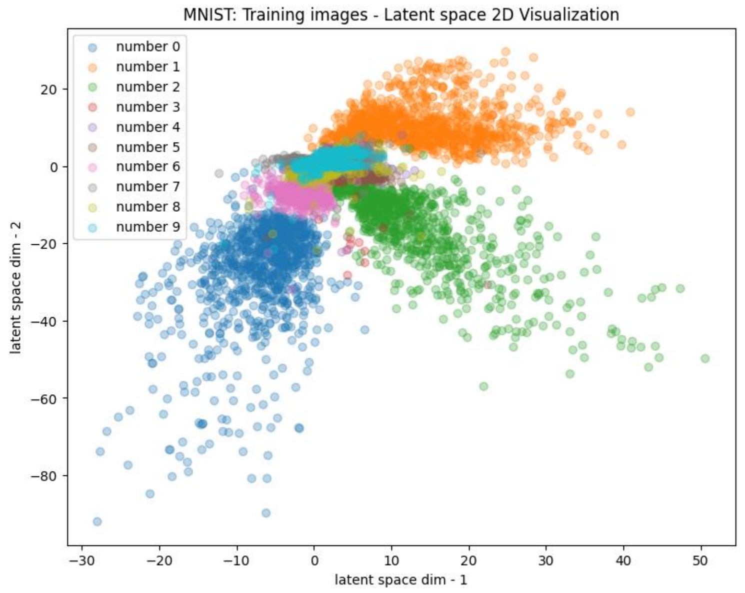
- **AE** model training makes use of a loss function typically comparing the output from the generator to the input using a loss function such as **BinaryCrossEntropy** or equivalent.

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

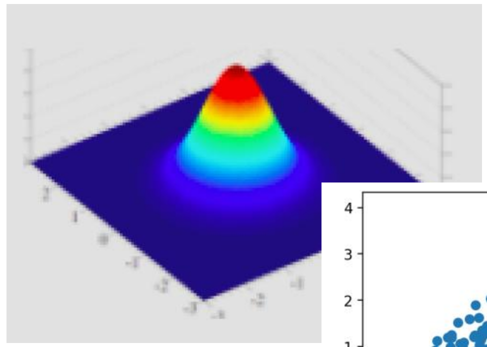
  "***We want to encode latent-space vectors onto a probability space***".

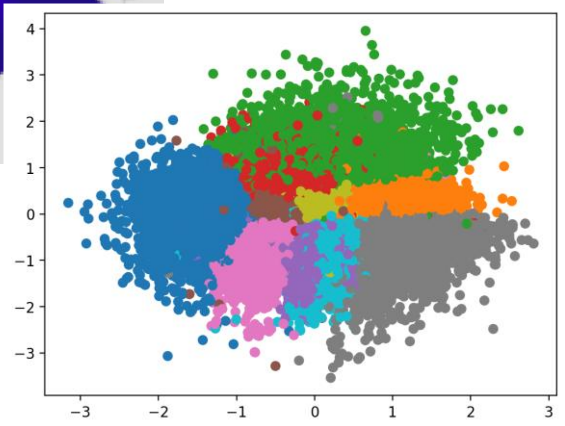
Variational AutoEncoders

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 .

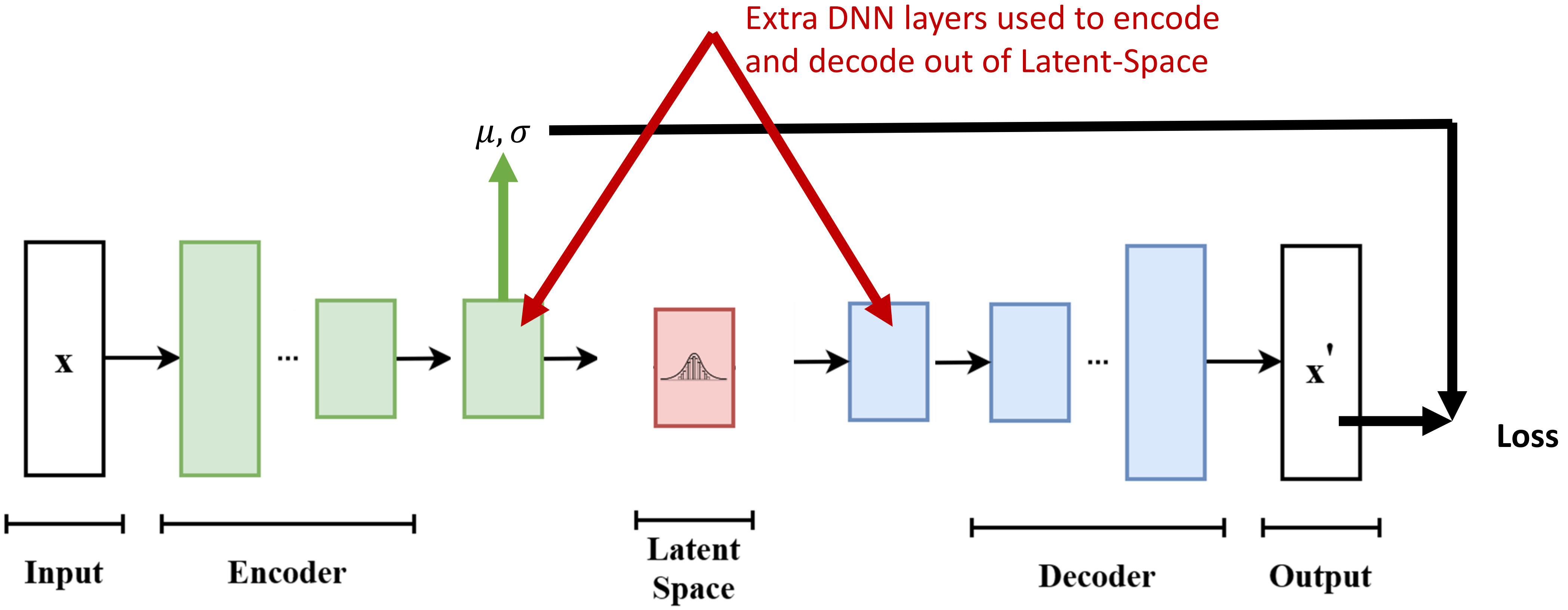
This is a measure of the entropy/information difference between the distributions of $P(x)$ and $Q(x)$.

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.



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:

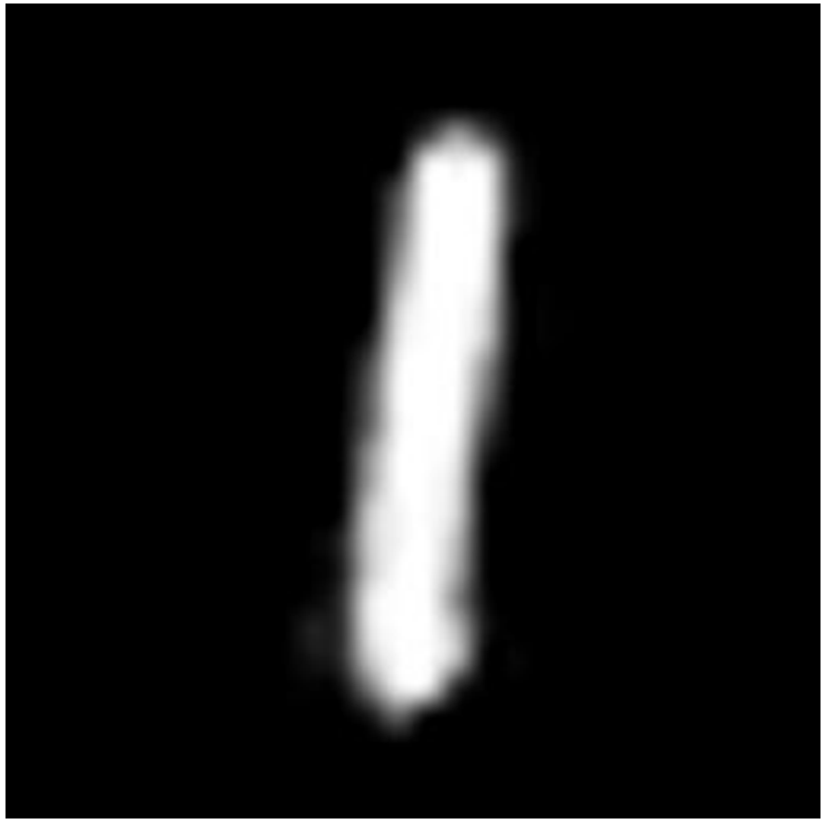
$$Loss = Loss_{BCE}(\boldsymbol{logits}) + Loss_{KL}(\boldsymbol{\mu}, \boldsymbol{\sigma})$$

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

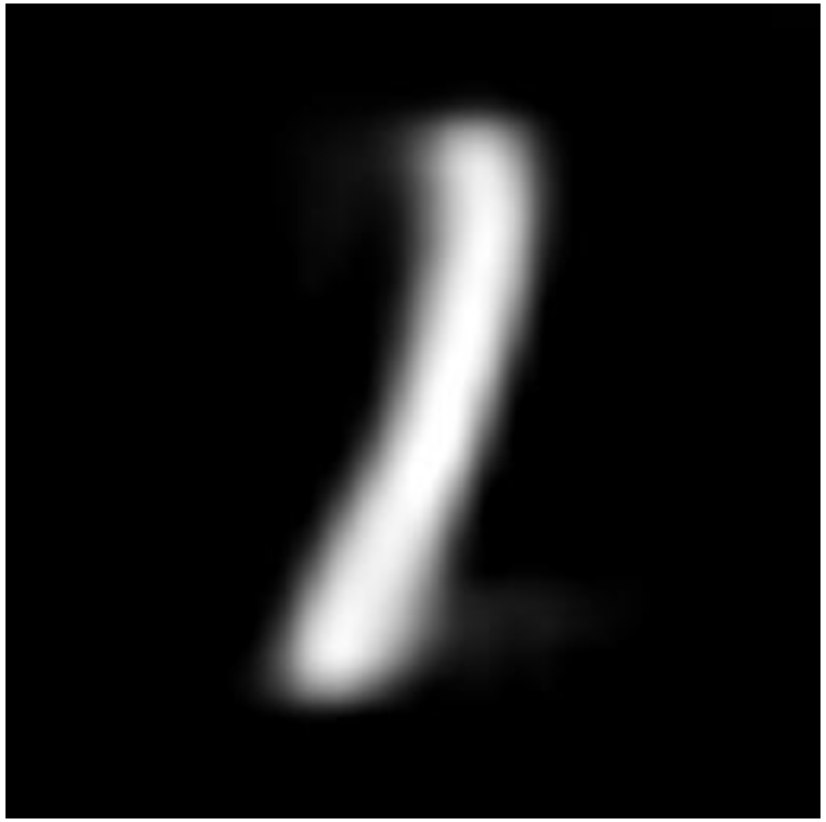
Sampling using an AE vs VAE

Generated using a trained (V)AE network with PyTorch. Took the LS coordinate of a starting image and the end image, then interpolated between the 2 coordinates generating individual frames and stitching them together.
(This took several hours on a GPU!)

AE



VAE



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 ***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?

uNET – An example ML/AI model

- **uNET** architecture from: [arXiv:1505.04597](https://arxiv.org/abs/1505.04597)
- Similar in structure to an **AE**/**VAE**
- Truth when training these models is typically image masks.
- Information is passed through latent-space to "learn" relationships in the training data.
- Information is also passed through the network itself ***skipping*** the **LS** to allow the upscaling components to extract higher-order/precise features.

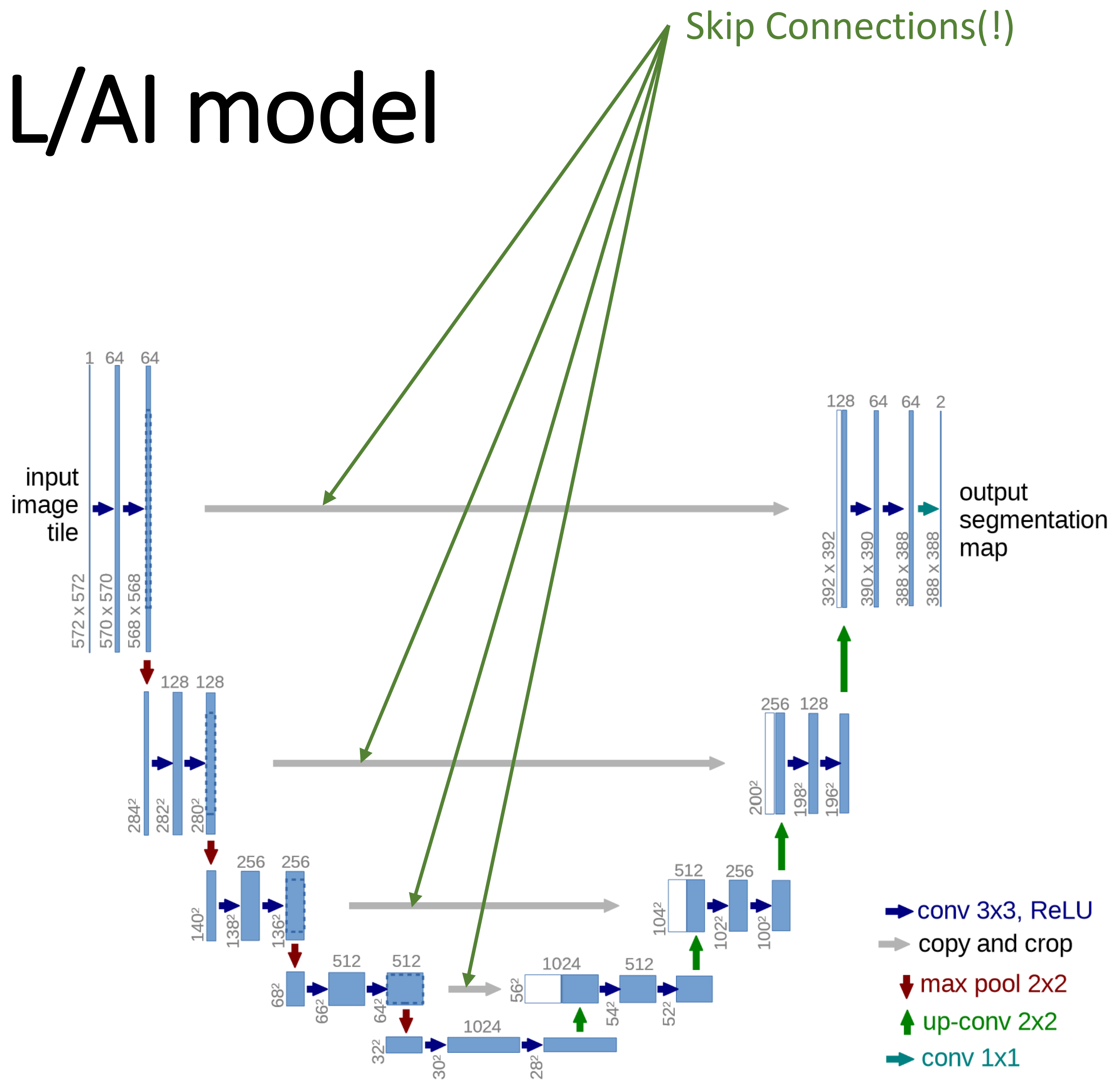


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted

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, or the loss distribution.
- This relies on the fact that a trained model will give an unexpectedly large value in either or both spaces 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.

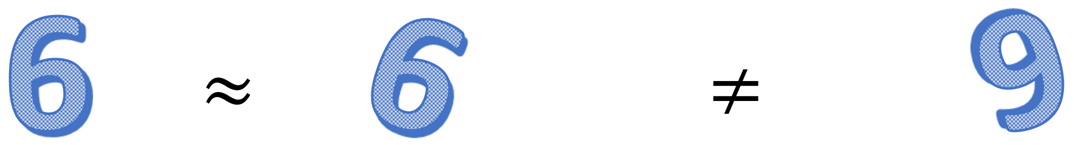
  Model might assume that only right-handed people wave.
  (or that people waving are right-handed…)

- 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)

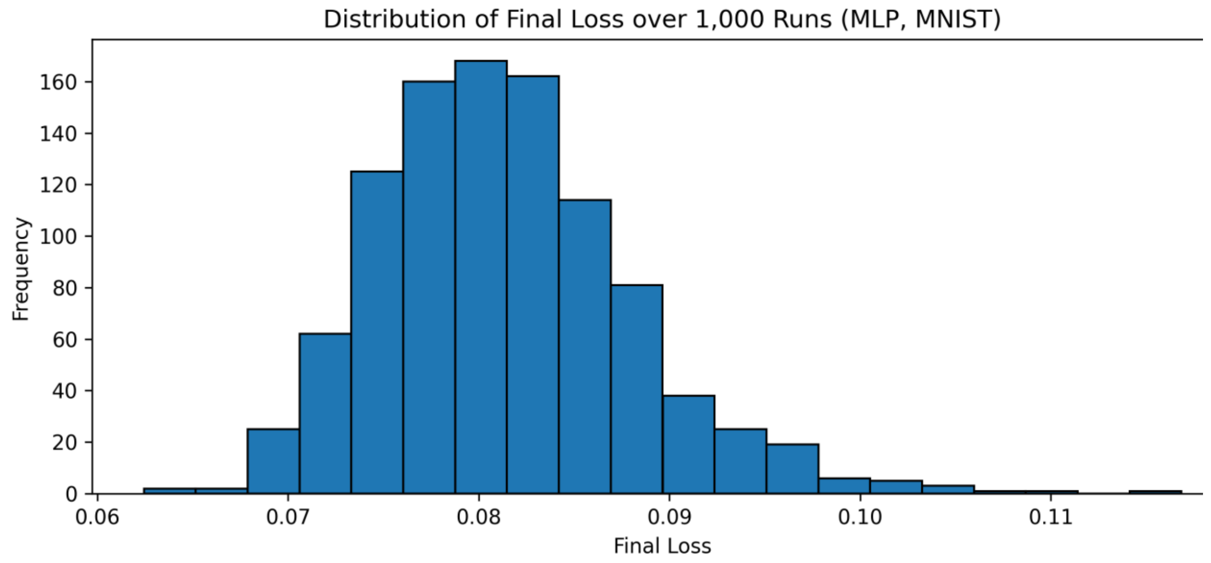
- In the case of certain models and certain datasets invariance is a good and desirable characteristic.
- However, if we’re looking to build automatic number recognition for documents scanned with the *correct* orientation…

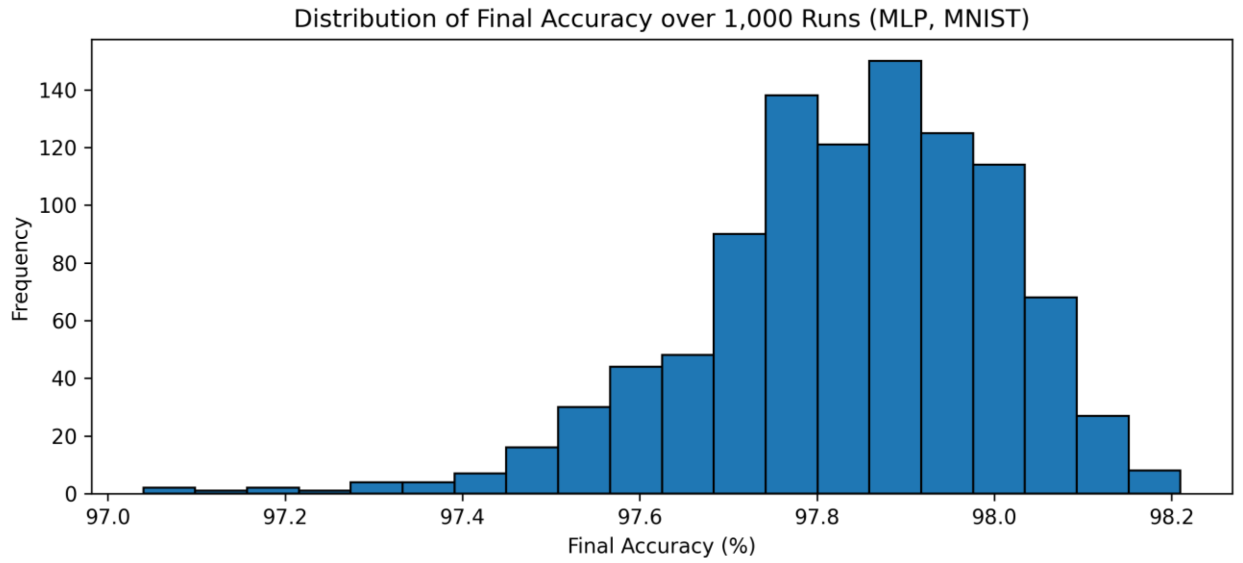


- Translations are good & useful when used correctly, but they can also cause problems when not checked.

Post Training – DNNs

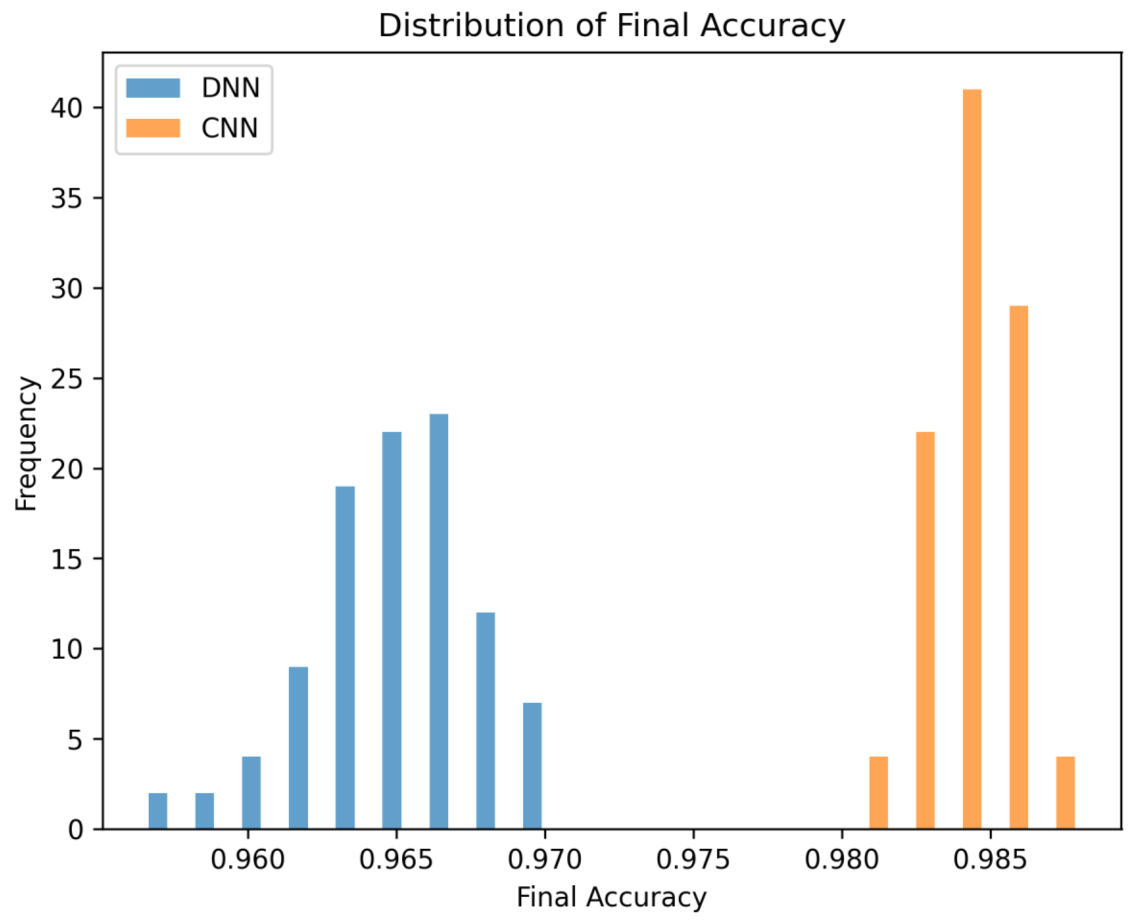
- Final model performance is a function of the input initialization weights.
- Given all we care about is model performance we're free to pick the version which gives us the best outcome each time.

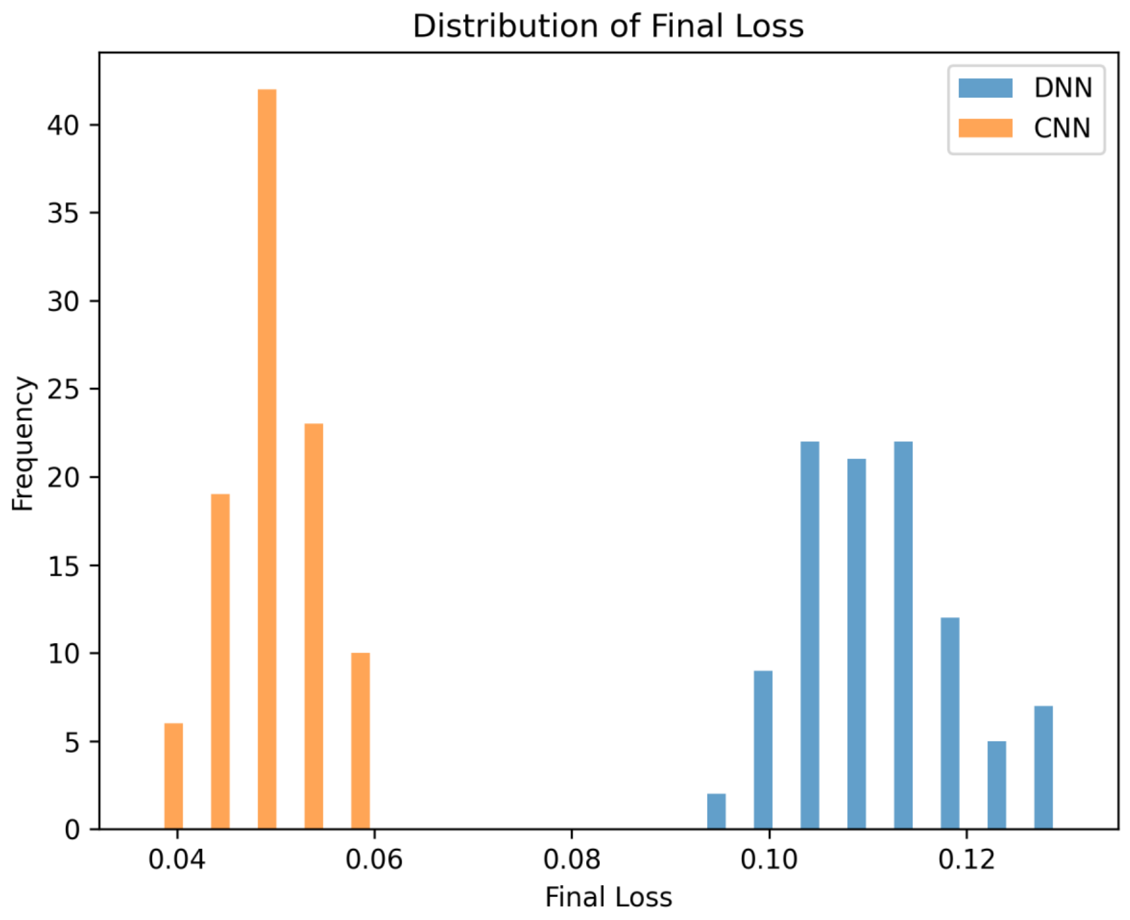




Post Training – CNNs

- Reducing the number of free parameters and being cleverer gives better model performance.
  - *(resulting accuracy/loss from 100 trained mnist classifier models)*





CNN Workshop

Workshop

- Today workshop we will investigate using and understanding **CNN**s to build an **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.
  (still >10min on mid-tier consumer GPU!)

CNNs models

- Matrices can pick up long-range correlations much better than pure DNN models.

- This is because the operations over each neuron operate over more than 1 input (pixel/value) at a time with limited field of focus.
- This means that reconstructed data is more complex with higher fidelity is better with CNN compared to DNN.

- It’s possible to use trained Encoders/Decoders separately or as part of an AE/VAE. It’s up to you which component you are interested in using for what task.

PyTorch matrices

- There are 3 pre-built models available in the Keras/PyTorch implementation.

- Conv1d

https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv1d.html

- Conv2d

https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

- Conv3d

https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv3d.html

VAE in PyTorch

- The implementation in PyTorch or TensorFlow for VAE tends to use a “re-parameterization” trick.

- This corresponds “to pick a random gaussian and compare how much the distribution of LS values from this existing distribution”.

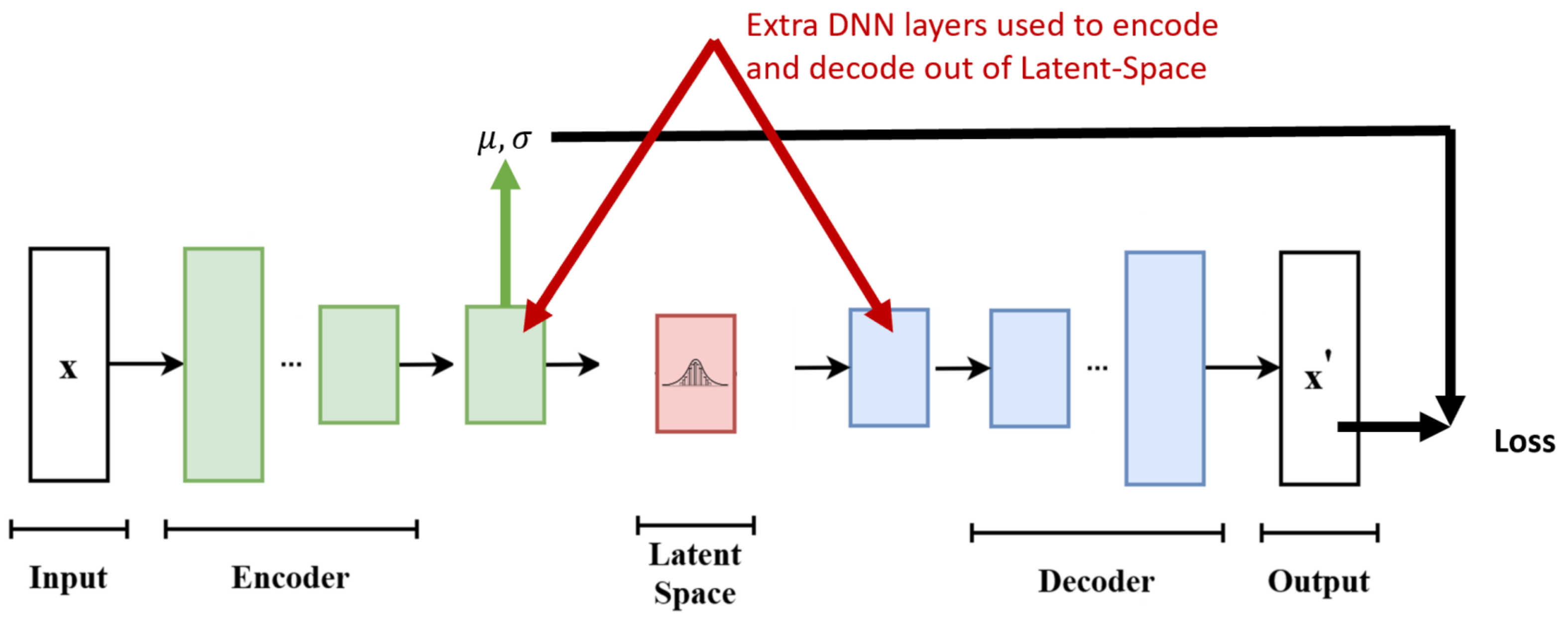
- Instead of evaluating the **L**atent-**S**pace ***separately*** to the ***mean***/***variance*** for each coordinate the LS is ***derived*** from these 2 values.

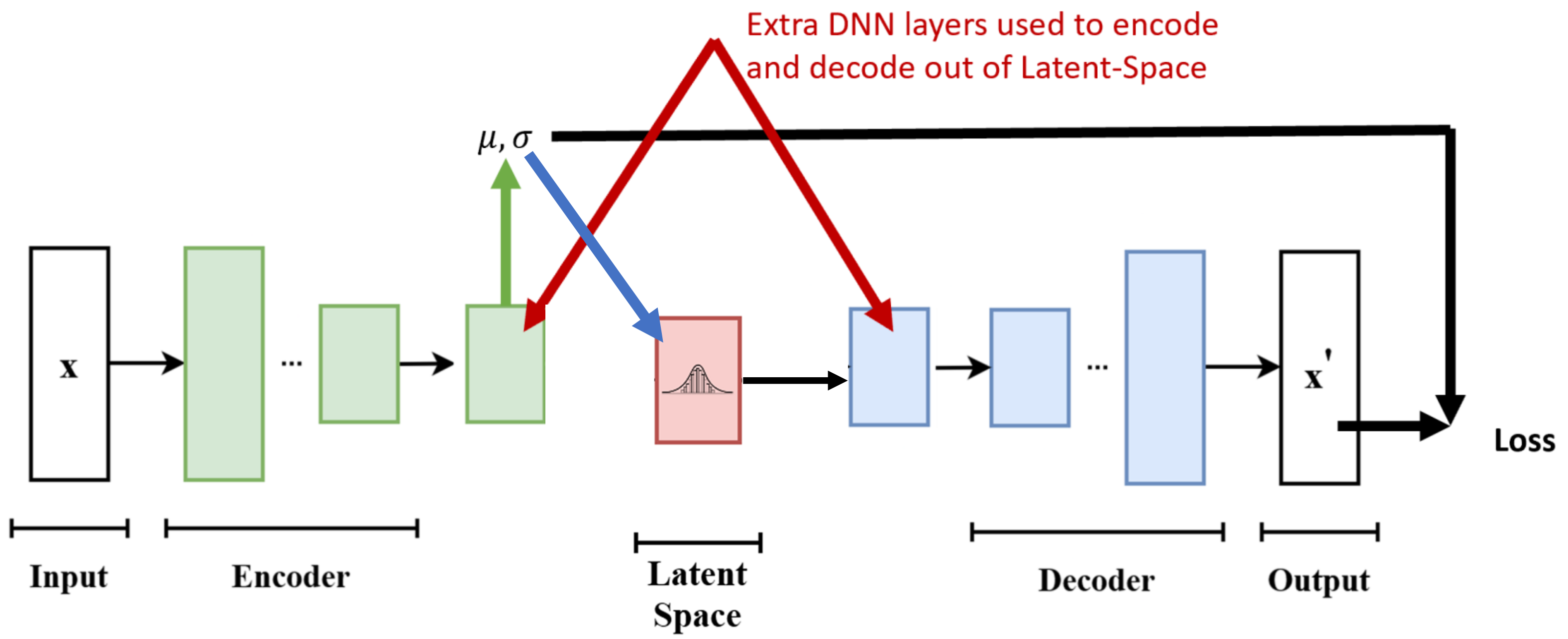
- In practice the output from an Encoder is **μ** and **log(var)** directly.
- The input to the decoder is still the vector in the **LS**.

VAE in PyTorch

- Traditionally presented as "extra" connections

- In practice, the LS is derived from the **μ** and **log(var)**





Today’s Workshop

1. An example of coding up your own Sobol operators and apply them to an input image
2. Now make some random operators and apply them to an input image. Main goal from this is to understand the format of the output image compared to the input.
3. Then we will construct the VAE model and evaluate the anomaly
4. After building the model we want to train it
5. Save&Load the model and generate new images
6. Plot the full distribution of losses per-event and compare it to the anomalous loss graphically
7. Finally build a CNN classifier for the CIFAR10 dataset and see the impact of data augmentation

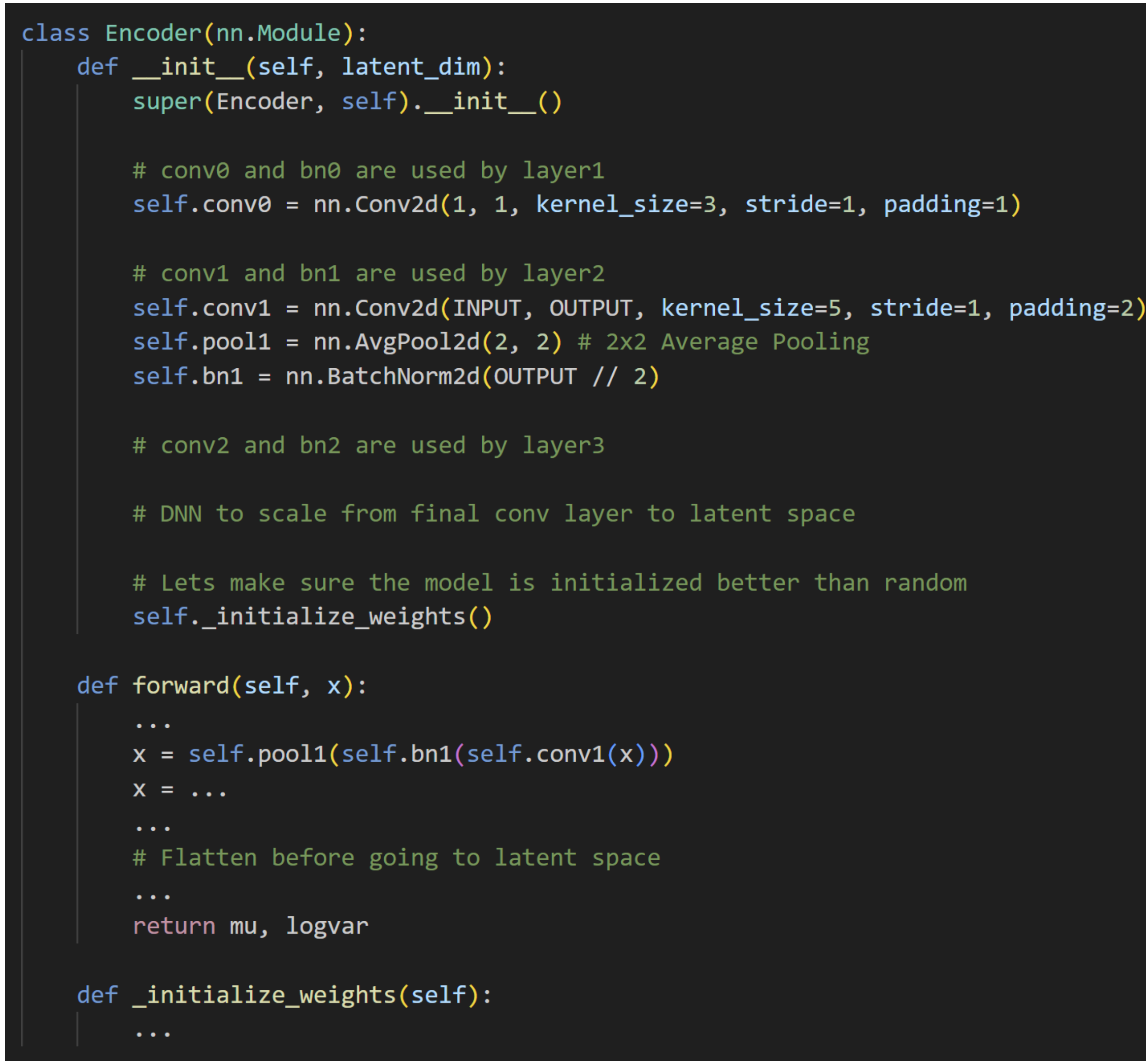
VAE Encoder

Each "layer" consists of 3 parts:

"conv2d" – the filter(s)

"Pooling" – reducing dimensions

"Normalisation" – mainly used because it increases training stability



```python
class Encoder(nn.Module):
    def __init__(self, latent_dim):
        super(Encoder, self).__init__()

        # conv0 and bn0 are used by layer1
        self.conv0 = nn.Conv2d(1, 1, kernel_size=3, stride=1, padding=1)

        # conv1 and bn1 are used by layer2
        self.conv1 = nn.Conv2d(INPUT, OUTPUT, kernel_size=5, stride=1, padding=2)
        self.pool1 = nn.AvgPool2d(2, 2) # 2x2 Average Pooling
        self.bn1 = nn.BatchNorm2d(OUTPUT // 2)

        # conv2 and bn2 are used by layer3

        # DNN to scale from final conv layer to latent space

        # Lets make sure the model is initialized better than random
        self._initialize_weights()

    def forward(self, x):
        ...
        x = self.pool1(self.bn1(self.conv1(x)))
        x = ...
        ...
        # Flatten before going to latent space
        ...
        return mu, logvar

    def _initialize_weights(self):
        ...
```

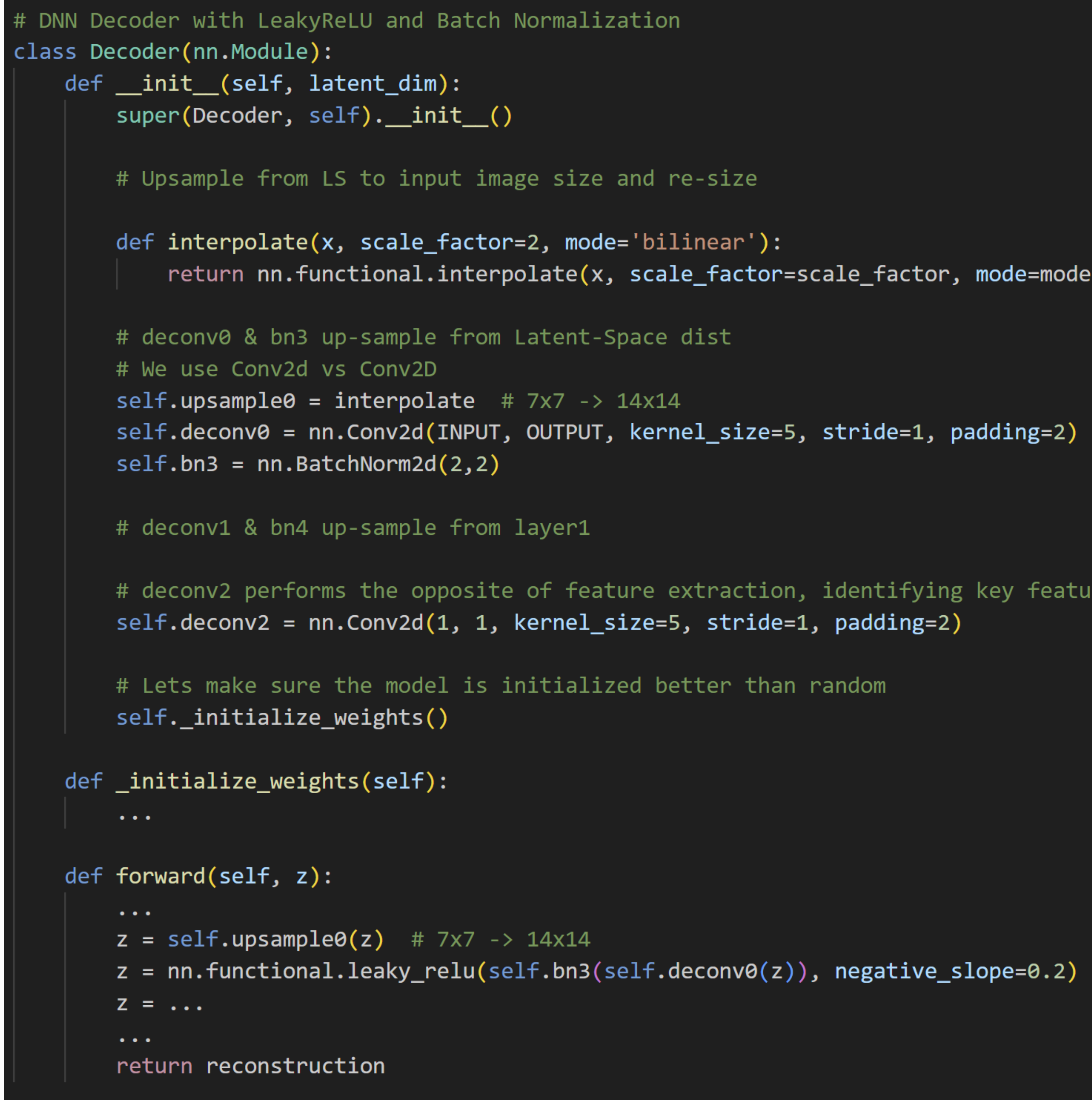
VAE Decoder

Each layer again is 3 parts:

“conv2d” – the filters

“upsample” – increase dimensions

“normalise” – included for training stability



```
# DNN Decoder with LeakyReLU and Batch Normalization
class Decoder(nn.Module):
    def __init__(self, latent_dim):
        super(Decoder, self).__init__()

        # Upsample from LS to input image size and re-size

        def interpolate(x, scale_factor=2, mode='bilinear'):
            return nn.functional.interpolate(x, scale_factor=scale_factor, mode=mode

        # deconv0 & bn3 up-sample from Latent-Space dist
        # We use Conv2d vs Conv2D
        self.upsample0 = interpolate  # 7x7 -> 14x14
        self.deconv0 = nn.Conv2d(INPUT, OUTPUT, kernel_size=5, stride=1, padding=2)
        self.bn3 = nn.BatchNorm2d(2,2)

        # deconv1 & bn4 up-sample from layer1

        # deconv2 performs the opposite of feature extraction, identifying key featu
        self.deconv2 = nn.Conv2d(1, 1, kernel_size=5, stride=1, padding=2)

        # Lets make sure the model is initialized better than random
        self._initialize_weights()

    def _initialize_weights(self):
        ...

    def forward(self, z):
        ...
        z = self.upsample0(z)  # 7x7 -> 14x14
        z = nn.functional.leaky_relu(self.bn3(self.deconv0(z)), negative_slope=0.2)
        z = ...
        ...
        return reconstruction
```