

# Unsupervised methods

Diving deep into autoencoders

Petar Veličković

Artificial Intelligence Group  
Department of Computer Science and Technology, University of Cambridge, UK

# Introduction

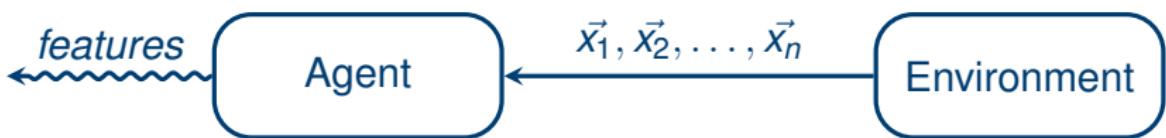
- ▶ In this lecture, I will guide you through the essentials of using deep neural networks for *unsupervised learning*.
- ▶ We will focus primarily on **autoencoders**, which offer a good tradeoff between model capacity and ease of training—and are still widely used both industrially and in research.
- ▶ For completeness, we will also briefly survey two historically used architectures for unsupervised learning (*RBM*s and *DBN*s), and a bleeding-edge *GAN* architecture (more details on Advanced Deep Learning & Reinforcement Learning...)

# The three “flavours” of machine learning

- ▶ Unsupervised learning
- ▶ Supervised learning  
*(more details next week!)*
- ▶ Reinforcement learning  
*(more details on Advanced DL & RL)*

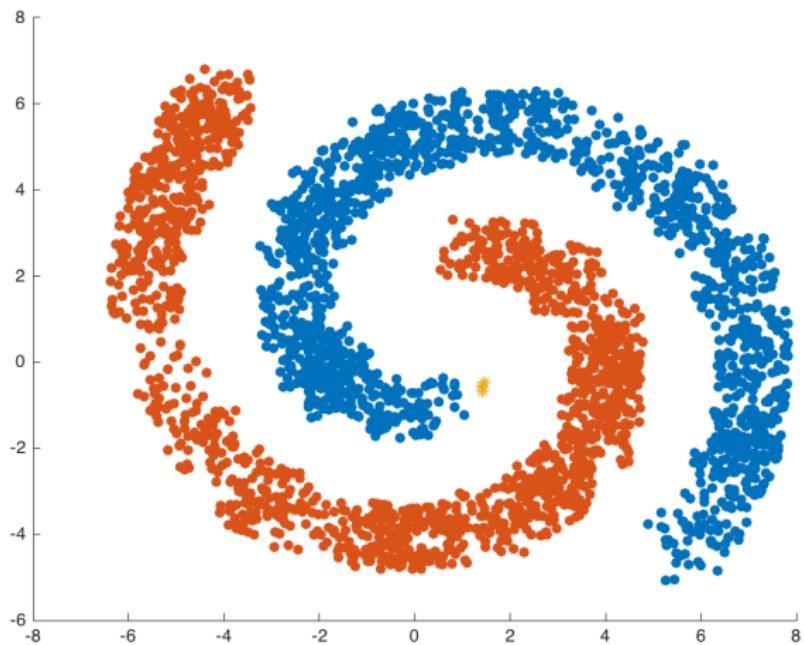
# Unsupervised learning

- ▶ The environment gives you *unlabelled data*—and asks you to assign useful features/structure to it.



- ▶ Example: study data from patients suffering from a disease, in order to discover different (previously unknown) types of it.

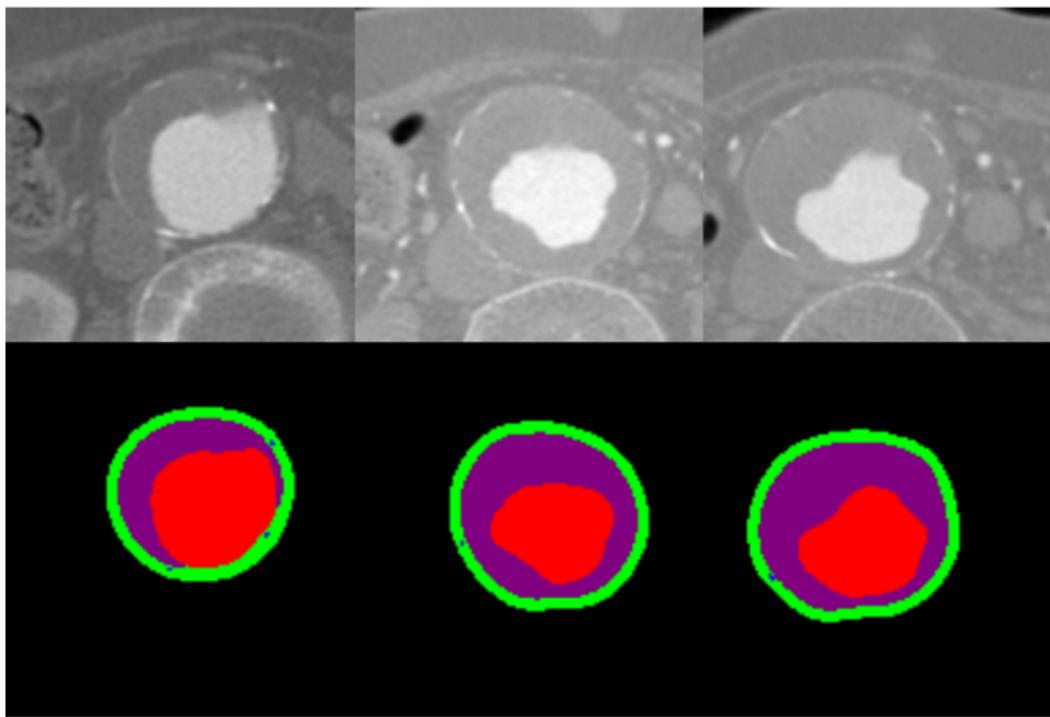
# Example: Clustering



# What's the point if we don't have labels?

- ▶ As in the above example, unsupervised learning can often be a *precursor to supervised learning*, if we don't even know what the labels should be (e.g. disease subtypes)!
- ▶ Often vastly increases the amount of data available! Obtaining labelled data is not always:
  - ▶ Appropriate (e.g. if, as above, *we don't even know the labels*);
  - ▶ Cheap (e.g. *segmenting medical images*);
  - ▶ Feasible (e.g. clinical studies for *extremely rare diseases*).
- ▶ Can aid better *dimensionality reduction*, simplifying the work of other algorithms, allow for *synthesising new training data*... and much more.
- ▶ Humans are essentially learning (mostly) unsupervised!

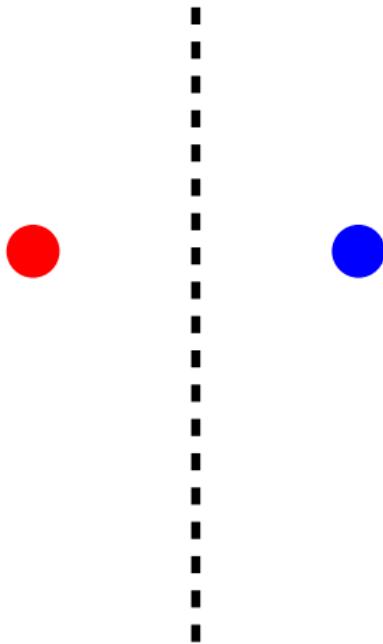
## *Example: Medical image segmentation*



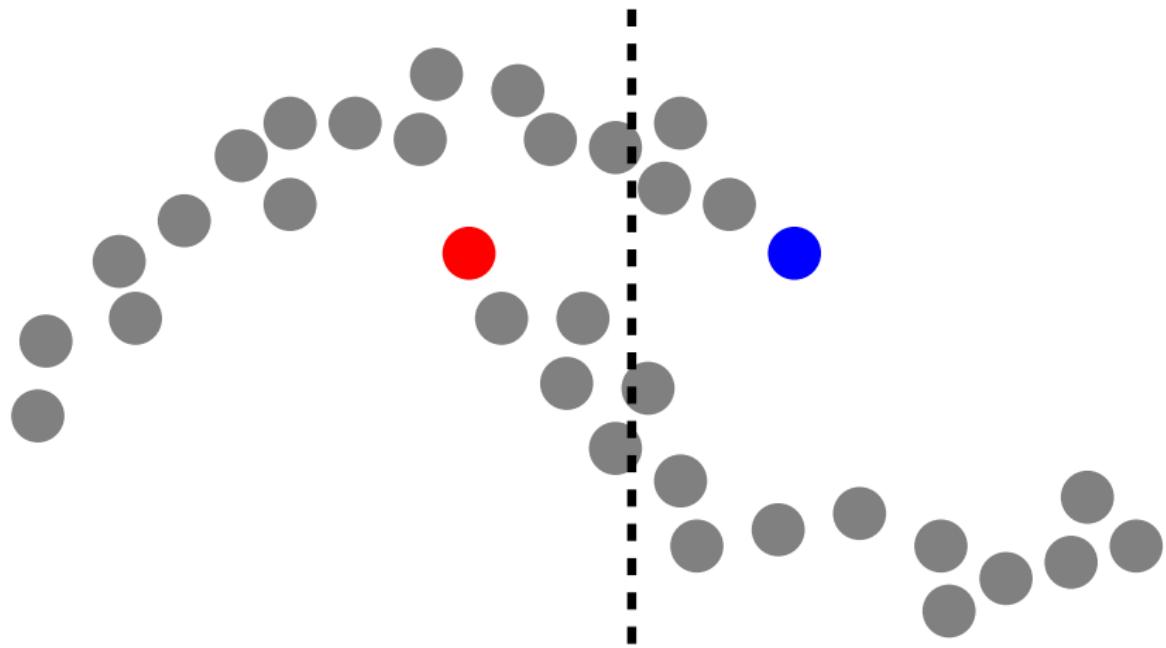
# How can unlabelled data help?



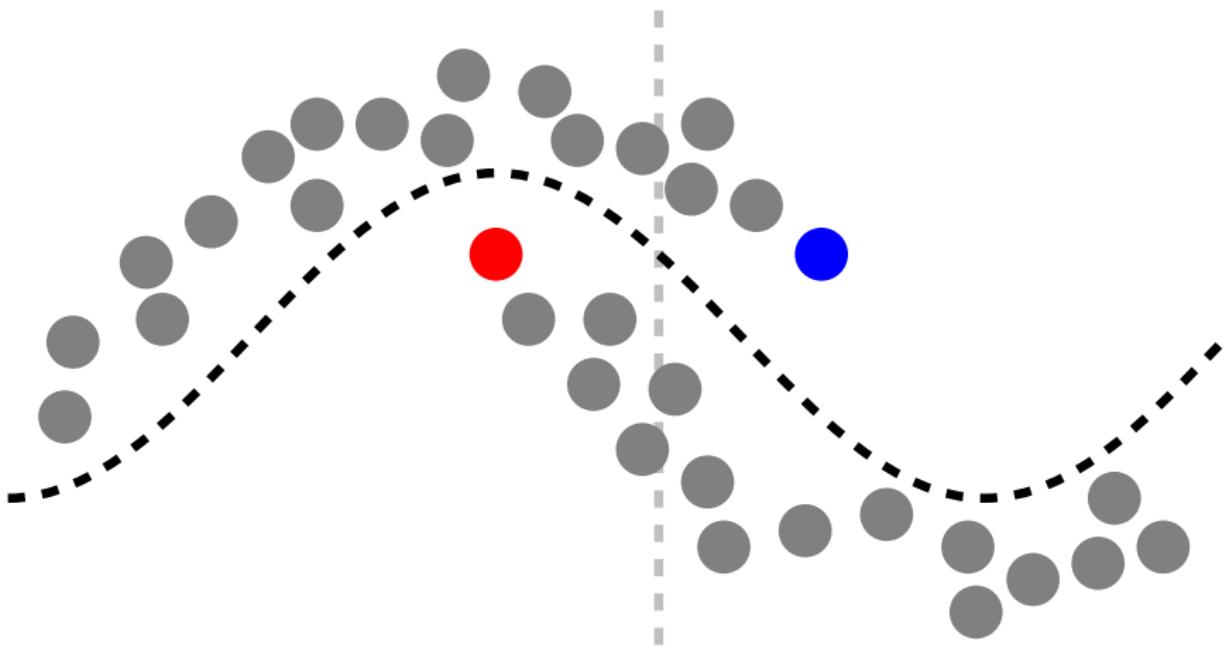
# How can unlabelled data help?



# How can unlabelled data help?



# How can unlabelled data help?



# Unsupervised learning is the future! (LeCun, 2017)



## ■ "Pure" Reinforcement Learning (cherry)

- ▶ The machine predicts a scalar reward given once in a while.
- ▶ **A few bits for some samples**

## ■ Supervised Learning (icing)

- ▶ The machine predicts a category or a few numbers for each input
- ▶ Predicting human-supplied data
- ▶ **10→10,000 bits per sample**

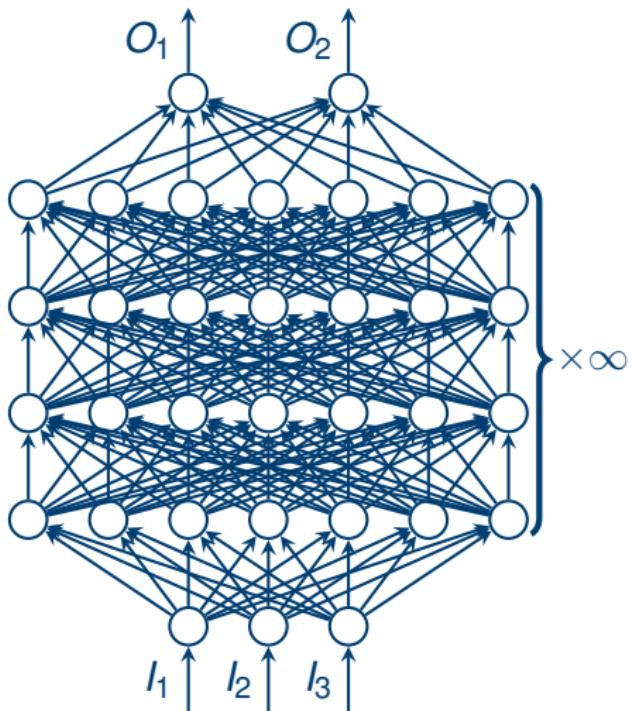
## ■ Unsupervised/Predictive Learning (cake)

- ▶ The machine predicts any part of its input for any observed part.
- ▶ Predicts future frames in videos
- ▶ **Millions of bits per sample**

■ (Yes, I know, this picture is slightly offensive to RL folks. But I'll make it up)



# (Re)introducing neural networks and deep learning

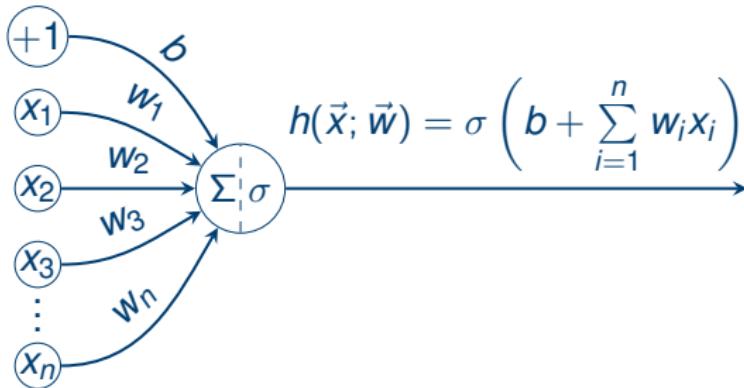


# Neural networks

- ▶ To make life simpler (esp. notationally!), let's start with a slightly more thorough introduction to simple neural networks.
- ▶ This might restate some of the material you've already seen, with the aim of making notation more consistent!
- ▶ **Neural networks** are structures of interconnected processing units (*neurons*).
- ▶ Each neuron computes a linear combination of its *inputs*, afterwards potentially applying an *activation function*, to produce its *output*.
- ▶ Occasionally, I will illustrate how to specify neural networks of interest using *Keras* (`keras.io`). (**highly recommended!**)

# A single neuron

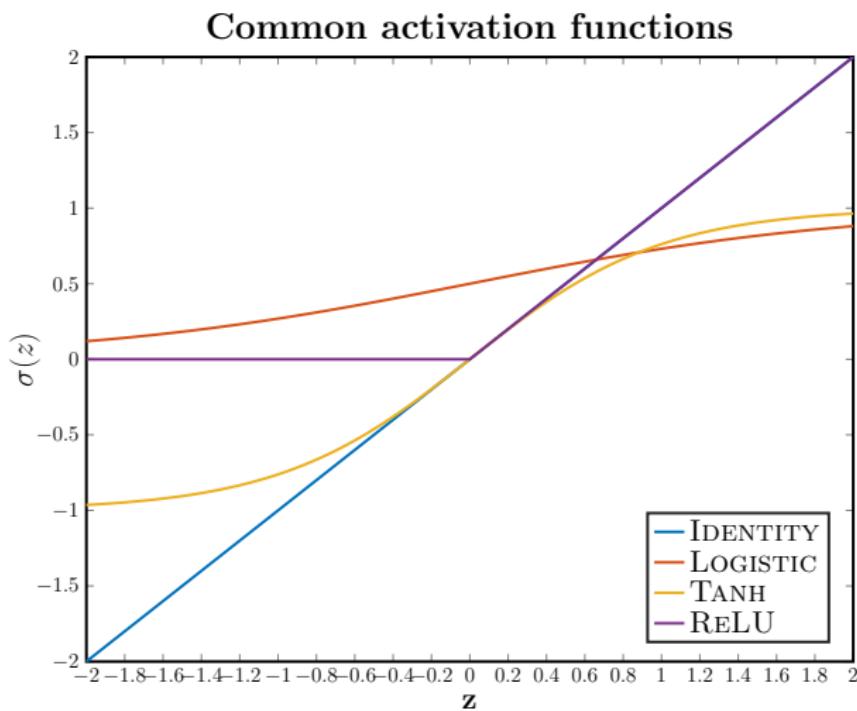
Within this context sometimes also called a *perceptron* (...)



Popular choices for the activation function  $\sigma$ :

- ▶ *Identity:*  $\sigma(x) = x$ ;
- ▶ *Rectified linear unit (ReLU):*  $\sigma(x) = \max(0, x)$ ;
- ▶ *Sigmoid functions:*  $\sigma(x) = \frac{1}{1+\exp(-x)}$  (*logistic*);  $\sigma(x) = \tanh x$ .

# Activation functions



# Neural networks and deep learning

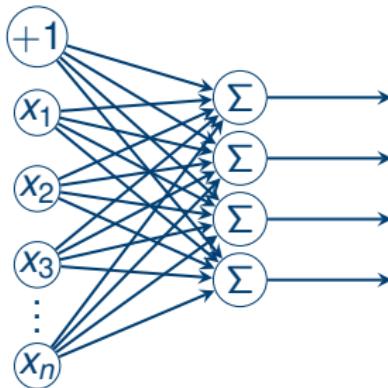
- ▶ It is easy to extend a single neuron to a *neural network*—simply connect outputs of neurons to inputs of other neurons.
- ▶ We may do this in two ways:
  - ▶ **Feedforward**: the computation graph does not have cycles;
  - ▶ **Recurrent**: the computation graph has cycles.
- ▶ Typically we organise neural networks in a sequence of *layers*, such that a single layer only processes output from the previous layer. Everything with  $> 1$  hidden layer is “deep”!

## A few details on training

- ▶ Neural networks are trained from known (input, output) samples. The training algorithm adapts the neurons' weights to maximise *predictive power* on the training examples.
- ▶ This is done, for a single training example  $(\vec{x}, y)$ , by:
  - ▶ Computing the output of the network  $y' = h(\vec{x}; \vec{w})$ ;
  - ▶ Determining the *loss* of this output  $\mathcal{L}(y, y')$ ;
  - ▶ Computing partial derivatives of the loss with respect to each weight,  $\frac{\partial \mathcal{L}}{\partial w_i}$ , and using these to update weights.
  - ▶ Key words: *backpropagation, stochastic gradient descent*.
- ▶ More details next week!

# A simple classifier

Let's ignore the activation functions and “deep learning” for now... here is a simple, shallow, 4-class classifier.

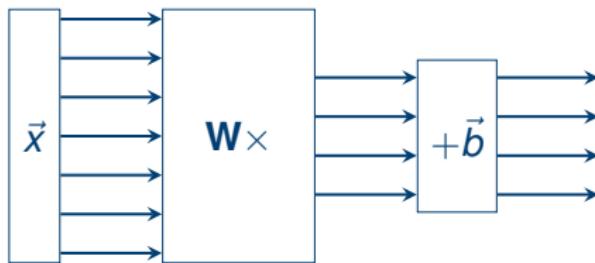


Choose the class which has the maximal *output*:

$$C = \operatorname{argmax}_j \{ b_j + \sum_{i=1}^n w_{ij} x_i \}$$

# Block notation

Note that this layer is essentially doing a matrix multiplication...



$$C = \operatorname{argmax}_j \left( \mathbf{W} \vec{x} + \vec{b} \right)_j$$

**N.B.**  $\mathbf{W}$  of size  $4 \times n$ ,  $\vec{b}$  of size 4!

# Softmax

- ▶ **Problem:** what should the targets be?
- ▶ Outputs are *unbounded*! For an example of the second class, the targets should be  $\vec{y} = [-\infty \quad +\infty \quad -\infty \quad -\infty] \dots$
- ▶ **Solution:** transform the outputs monotonically to the  $[0, 1]$  range, using the *softmax* function:

$$\text{softmax}(\vec{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

# Probabilistic classification

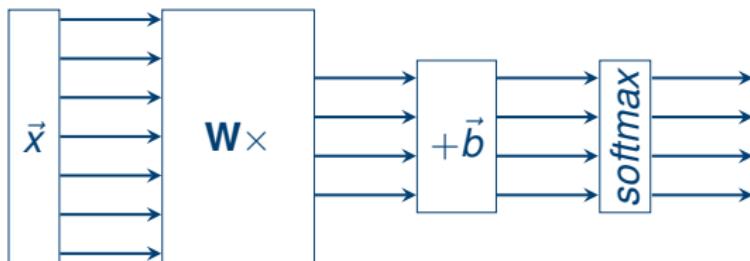
- ▶ This conveniently also makes the outputs add up to 1, so we can interpret  $y'_i = \text{softmax}(h(\vec{x}))_i = \mathbb{P}(\vec{x} \text{ in class } i)$ .
- ▶ Now the target for an example of the second class should be  $\vec{y} = [0 \ 1 \ 0 \ 0]$  ( $\sim$  one-hot encoding).
- ▶ Typically express the loss function as the *cross-entropy*:

$$\mathcal{L}(\vec{y}, \vec{y}') = - \sum_{i=1}^K y_i \log y'_i$$

where  $K$  is the number of classes.

# Back in business

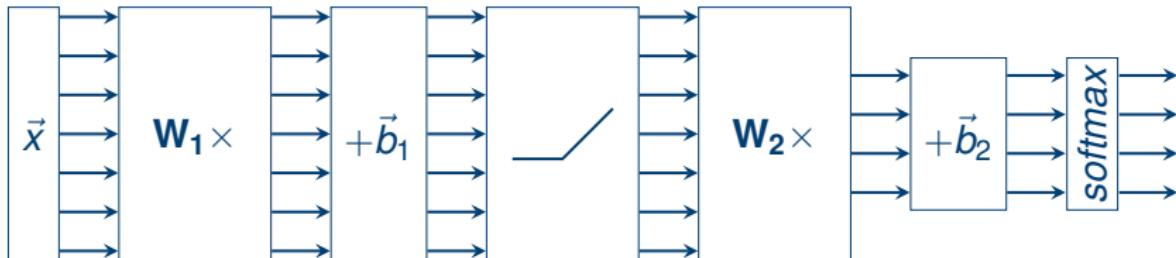
Integrating into our simple classifier:



$$C = \operatorname{argmax}_j \left\{ softmax \left( \mathbf{W} \vec{x} + \vec{b} \right)_j \right\}$$

# Going deeper with LEGO™

Making things *deep* is now easy...

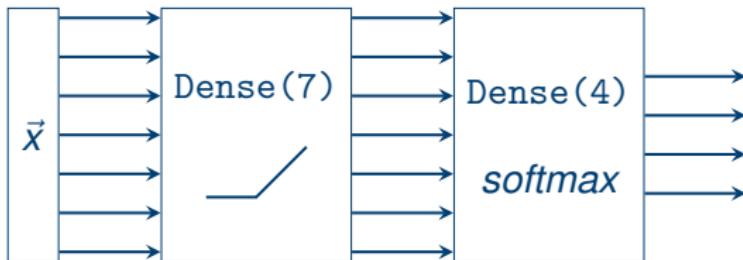


$$C = \operatorname{argmax}_j \left\{ \text{softmax} \left( \mathbf{W}_2 \text{ReLU} \left( \mathbf{W}_1 \vec{x} + \vec{b}_1 \right) + \vec{b}_2 \right)_j \right\}$$

**N.B.** the ReLU is *important*! A composition of linear functions is itself a linear function... (at least in theory—thank you, OpenAI :))

# Fully connected layers

The “matrix-multiply–bias–activation” (sometimes also called *fully connected* or Dense) layer is a common building block of neural networks.



Keras code:

```
x = Input(shape=(7,))
h = Dense(7, activation='relu')(x)
y = Dense(4, activation='softmax')(h)
```

# Working with images

- ▶ Simple fully-connected neural networks (as described already) typically fail on high-dimensional datasets (e.g. images).
  - ▶ Treating each pixel as an independent input...
  - ▶ ... results in  $h \times w \times d$  new parameters per neuron in the first hidden layer...
  - ▶ ... quickly deteriorating as images become larger—requiring exponentially more data to properly fit those parameters!
- ▶ **Key idea:** downsample the image until it is small enough to be tackled by such a network!
  - ▶ Would ideally want to extract some useful features first...
- ▶  $\implies$  exploit spatial structure!

# The *convolution* operator



## Enter the *convolution* operator

- ▶ Define a small (e.g.  $3 \times 3$ ) matrix (the *kernel*,  $\mathbf{K}$ ).
- ▶ Overlay it in all possible ways over the *input image*,  $\mathbf{I}$ .
- ▶ Record *sums of elementwise products* in a new image.

$$(\mathbf{I} * \mathbf{K})_{xy} = \sum_{i=1}^h \sum_{j=1}^w \mathbf{K}_{ij} \cdot \mathbf{I}_{x+i-1, y+j-1}$$

- ▶ This operator exploits *structure*—neighbouring pixels influence one another stronger than ones on opposite corners!
- ▶ Start with *random kernels*—and let the network find the optimal ones on its own!

# Convolution example

The diagram illustrates a convolution operation. It shows three matrices: the input matrix  $I$ , the kernel matrix  $K$ , and the resulting output matrix  $I * K$ .

**Input Matrix  $I$ :**

0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	1	1	0	0	0	0
1	1	0	0	0	0	0

**Kernel Matrix  $K$ :**

1	0	1
0	1	0
1	0	1

**Output Matrix  $I * K$ :**

1	4	3	4	1
1	2	4	3	3
1	2	3	4	1
1	3	3	1	1
3	3	1	1	0

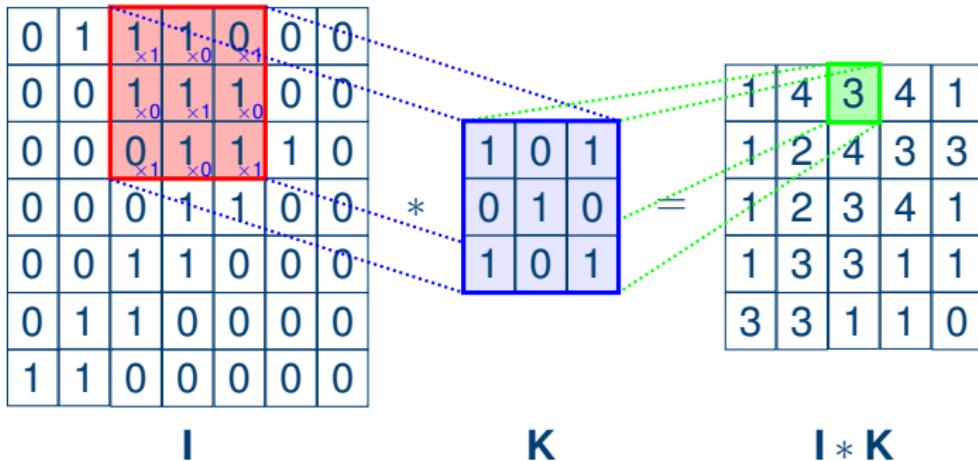
Dotted arrows indicate the receptive fields of each element in the output matrix, showing how elements in the input  $I$  contribute to the final result. The first element of the output matrix is highlighted in green.

# Convolution example

$$\begin{matrix} \begin{matrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{matrix} & * & \begin{matrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{matrix} & = & \begin{matrix} 1 & 4 & 3 & 4 & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{matrix} \end{matrix}$$

The diagram illustrates a convolution operation between two matrices, I and K. Matrix I is a 7x7 input matrix with values ranging from 0 to 1. Matrix K is a 3x3 kernel matrix with values 1, 0, 1. The result of the convolution is matrix I \* K, which is a 5x5 output matrix. The result of the convolution is highlighted in green, showing a value of 4 at position (2,2). Dotted lines indicate the receptive field of each output unit in matrix I.

# Convolution example



# Convolution example

The diagram illustrates a convolution operation. It shows three matrices: the input matrix  $I$ , the kernel matrix  $K$ , and the resulting output matrix  $I * K$ .

**Input Matrix  $I$ :**

0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	1	1	0	0	0	0
1	1	0	0	0	0	0

**Kernel Matrix  $K$ :**

1	0	1
0	1	0
1	0	1

**Output Matrix  $I * K$ :**

1	4	3	4	1
1	2	4	3	3
1	2	3	4	1
1	3	3	1	1
3	3	1	1	0

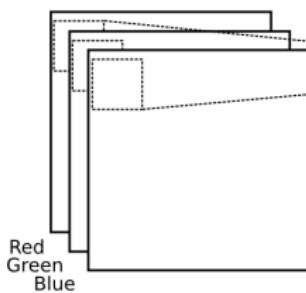
The diagram shows the convolution process. The input matrix  $I$  is shown with a red 3x3 window highlighting the center element. The kernel matrix  $K$  is shown below it. The result of the convolution is the output matrix  $I * K$ . The output matrix is shown with green dashed lines indicating the receptive field of each output unit. The value 4 in the top-left cell of the output matrix is highlighted in green, indicating it is the result of the current convolution step.

# Stacking convolutions

First layer weights:

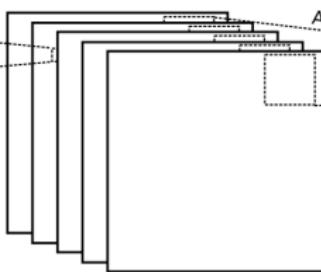


Second layer weights:

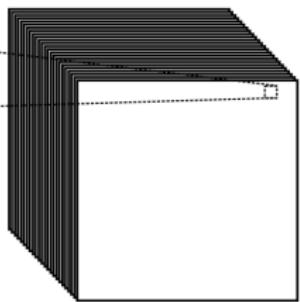


Input Image (Colour)

Apply filter 2

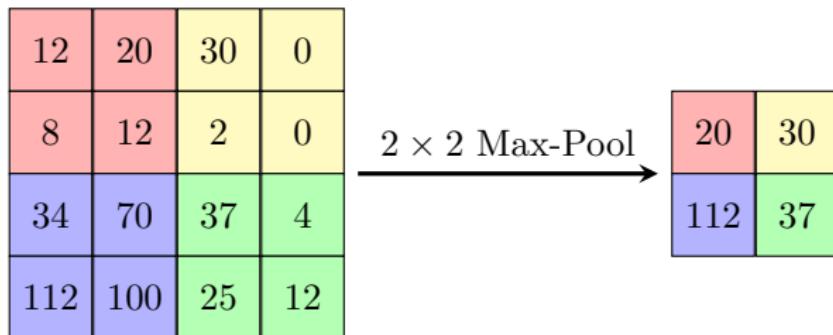


Apply filter 24

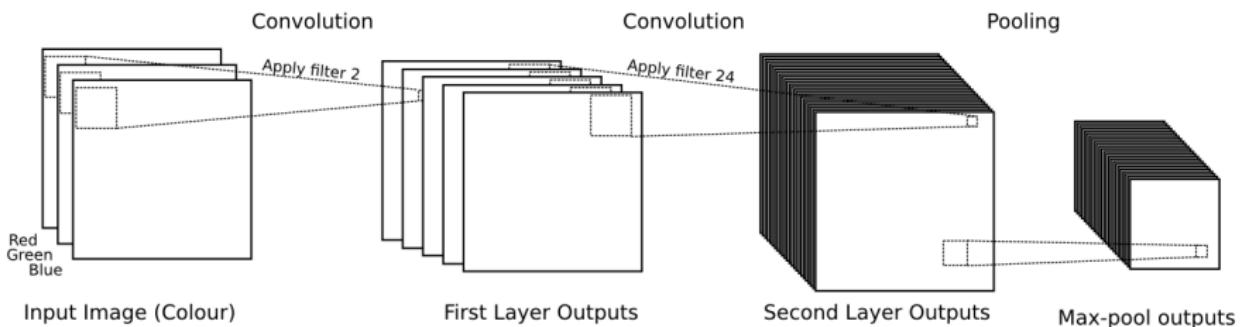


## Downsampling ( $\sim$ max-pooling)

Convolutions *light up* when they detect a particular feature in a region of the image. Therefore, when downsampling, it is a good idea to preserve maximally activated parts. This is the inspiration behind the *max-pooling* operation.

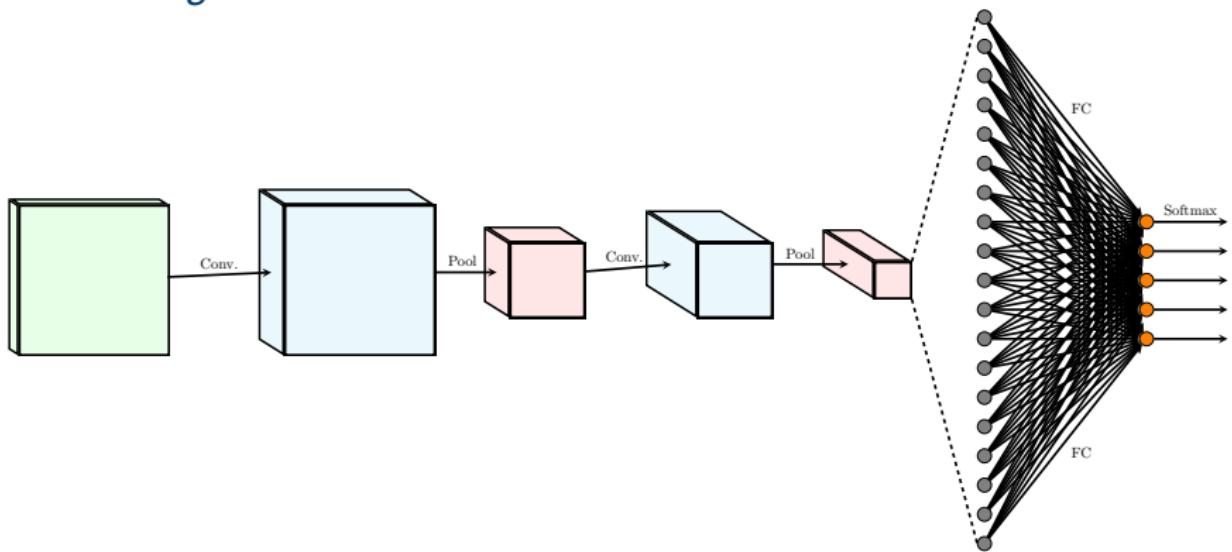


# Stacking convolutions and poolings



# Stacking convolutions and poolings

*Rough rule of thumb:* increase the *depth* (number of convolutions) as the *height* and *width* decrease.



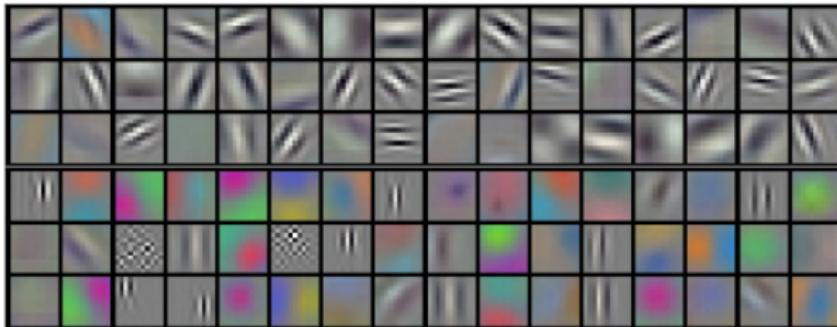
# CNN representations

Three ways to examine the CNN's internal representations:

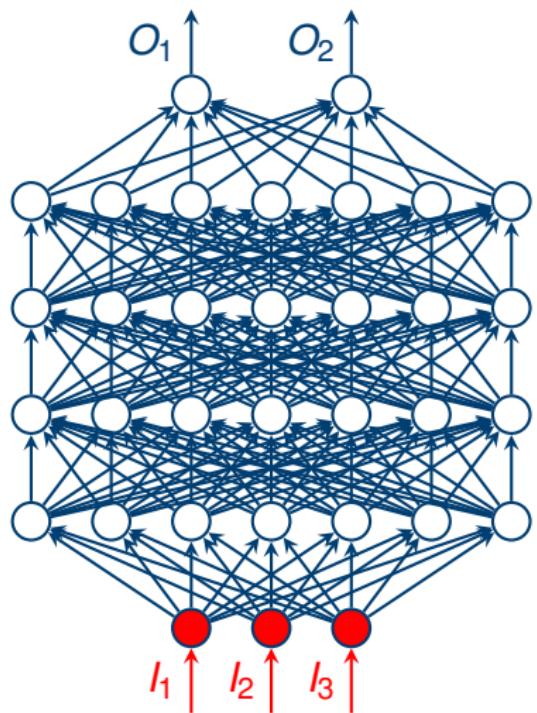
1. Observe the learnt *kernels*;
2. Pass an input through the network, observe the *activations*;
3. Coming later in this lecture...

# Observing kernels

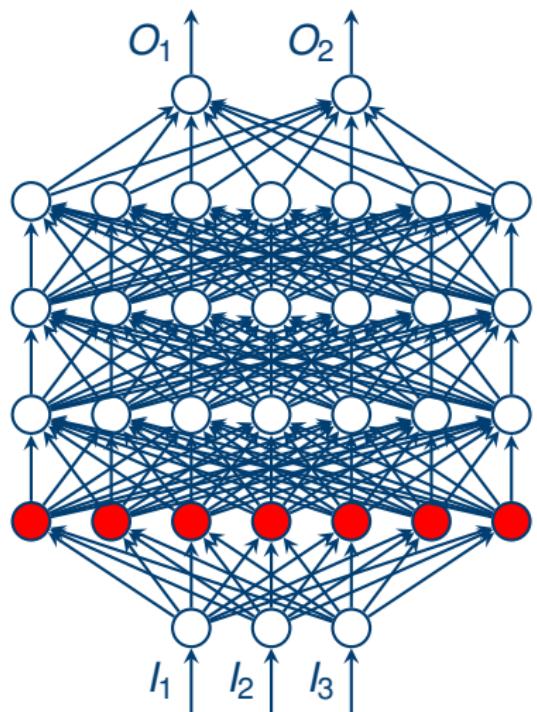
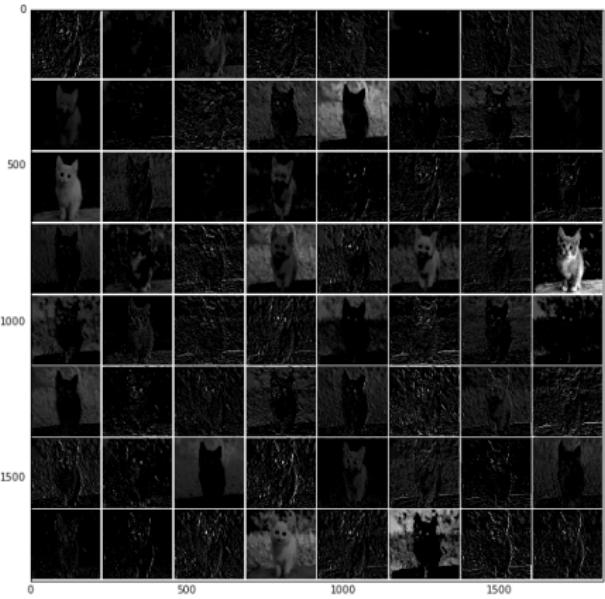
- ▶ Typically, as the kernels are small, gaining useful information from them becomes difficult already *past the first layer*.
- ▶ However, the first layer of kernels reveals something *magical*... In almost all cases, these kernels will learn to become *edge detectors*!



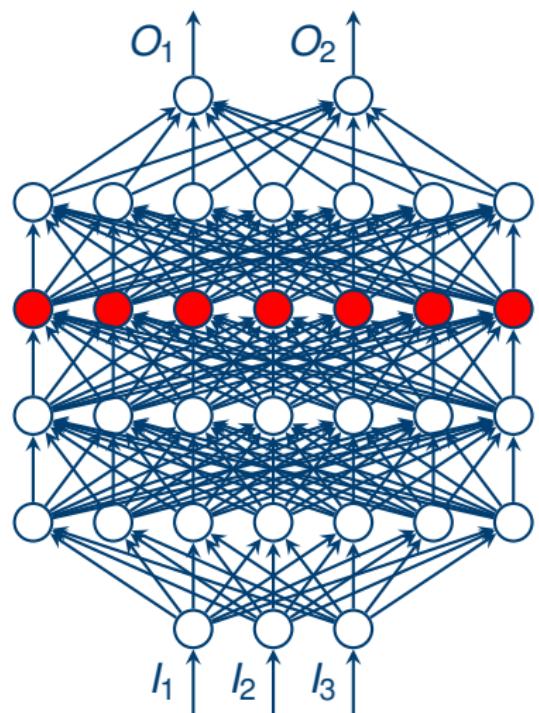
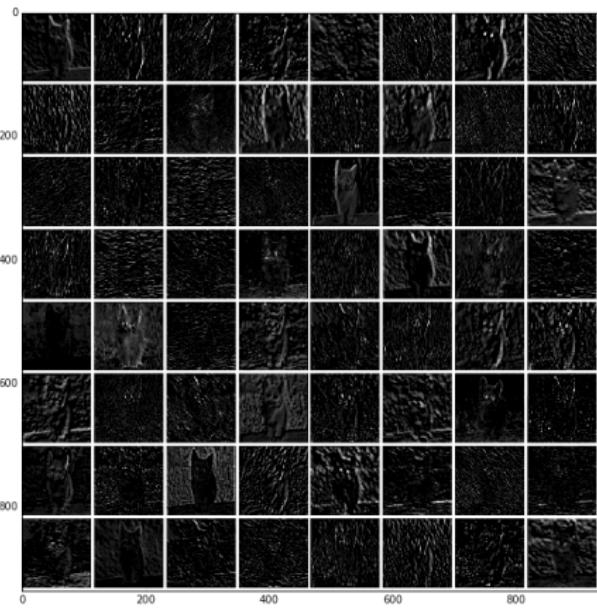
# Passing data through the network: *Input*



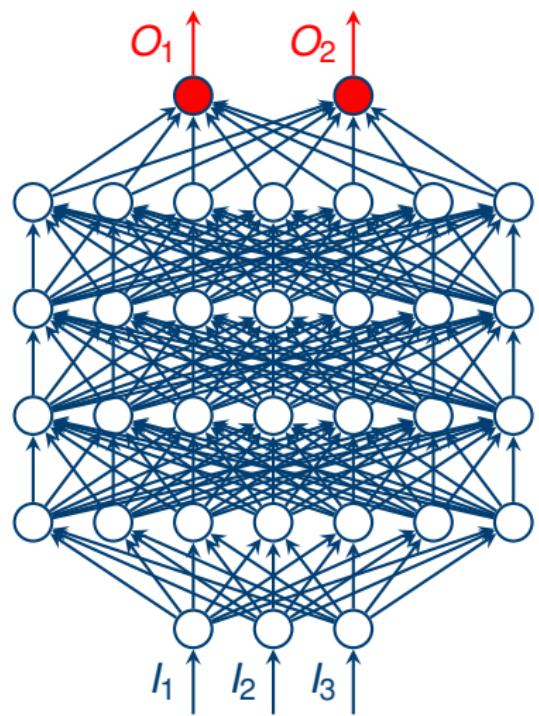
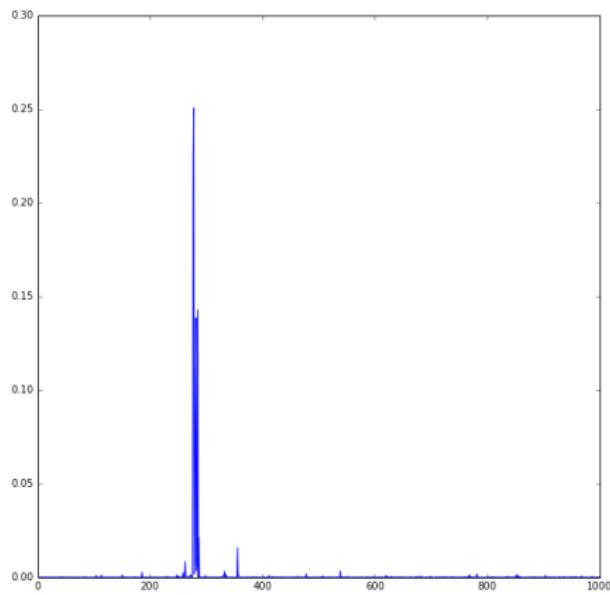
# Passing data through the network: *Shallow layer*



# Passing data through the network: *Deep layer*



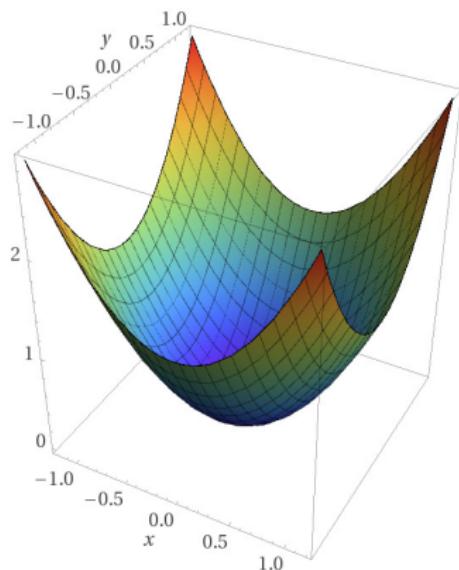
# Passing data through the network: *Output*



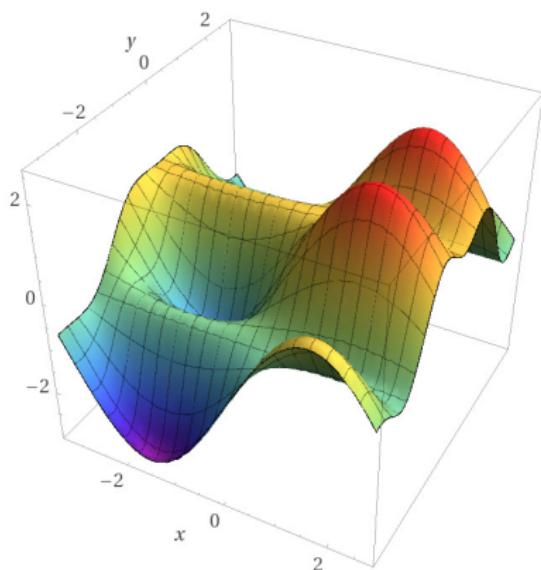
# Issues with learning through backpropagation

- ▶ We will start by analysing a direct way in which unsupervised techniques can aid the kind of supervised learning more common for deep neural networks.
- ▶ “Deep learning” was around for decades, but took a long time to become practically usable.
- ▶ We’d start with some randomly initialised weights, present our training examples to the network, and... the network *wouldn’t really learn that well.*

# Loss function surfaces

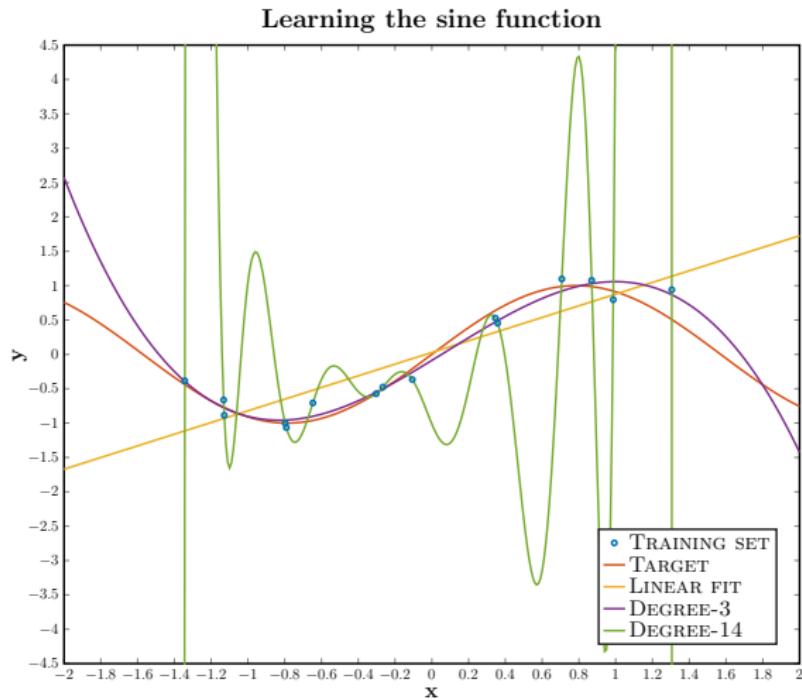


Computed by Wolfram|Alpha



Computed by Wolfram|Alpha

# Beware of *overfitting*!



## (In)appropriate initialisers

- ▶ It has since been determined that *initialisation* plays a **critical** role in neural network training stability.
- ▶ Extremely, what happens to error gradients if you initialise the network with *zero weights*? Or randomly sample them with a *huge variance*?
- ▶ *Be good to your signals...*
- ▶ Using an appropriate initialiser can mean the difference between getting *great results* and *not converging at all!*

# What can we do?

- ▶ Reuse a network that performs great on a much larger dataset, and *fine-tune* (some of) its weights.
  - ▶ This is the concept of *transfer learning* (week 7), and is fundamental to the successes of many deep learning startups. :)
  - ▶ Not always possible to do!
- ▶ I will now show how we can employ *unsupervised techniques* to determine weights that are “good” for working with our input data (regardless of what the outputs are!).
  - ▶ One of the first “success stories” of deep learning!
- ▶ In 2010, *Glorot and Bengio* discovered very appropriate parameters for randomly initialising weights (week 5). Unsupervised pre-training is very scarcely used nowadays.

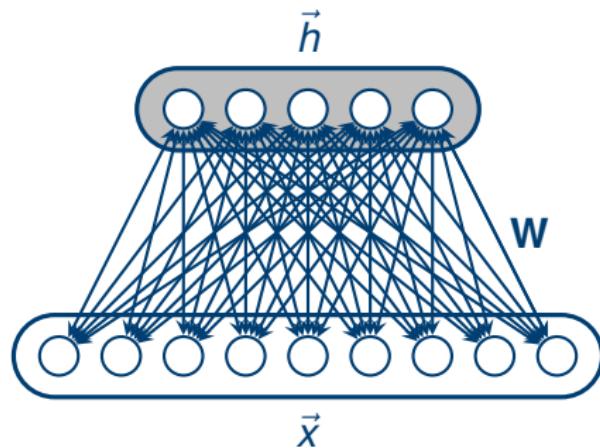
## Reconstruction objective

- ▶ Assume that we want to initialise a *single-layer fully-connected* neural network, to be trained by backpropagation.
- ▶ We will further assume, for simplicity, that the inputs in our training dataset are *binarised* (0/1).
- ▶ Without a clear target, we can assume that *a good choice of weights will cause the output to retain most of the information about the input.*
- ▶ Therefore, **the weights should be chosen such that we can also use them to reconstruct the input given the output!**

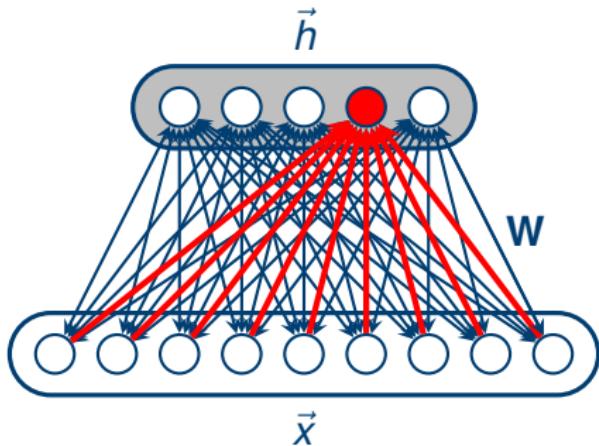
# Restricted Boltzmann Machine (RBM)

- ▶ A simple unsupervised *stochastic* extractor of binary features,  $\vec{h}$ , from binary data,  $\vec{x}$ .
- ▶ Parametrised by a weight matrix  $\mathbf{W}$  and bias vectors  $\vec{a}$  and  $\vec{b}$  to transform the data to the feature space, but *also to go back!*
- ▶  $\mathbb{P}(h_j = 1 | \vec{x}) = \sigma((\mathbf{W}\vec{x} + \vec{b})_j) = \frac{1}{1 + \exp(-(W\vec{x} + \vec{b})_j)}$
- ▶  $\mathbb{P}(x_i = 1 | \vec{h}) = \sigma((\mathbf{W}^T \vec{h} + \vec{a})_i) = \frac{1}{1 + \exp(-(W^T \vec{h} + \vec{a})_i)}$
- ▶ Trained efficiently using *contrastive divergence* (Hinton, 2010). Once trained, can use  $\mathbf{W}$  and  $\vec{b}$  as initial values for a neural net!

# Restricted Boltzmann Machine (RBM)

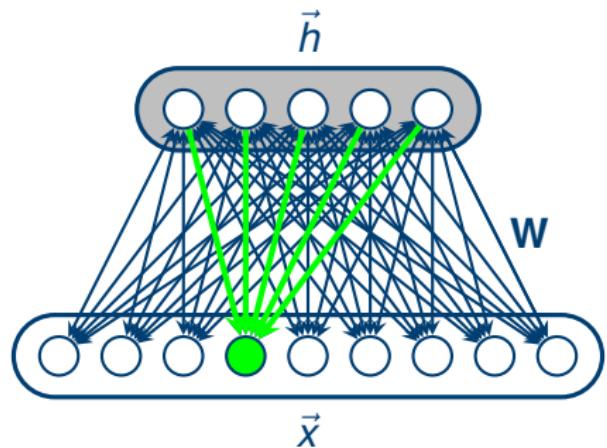


# Restricted Boltzmann Machine (RBM)



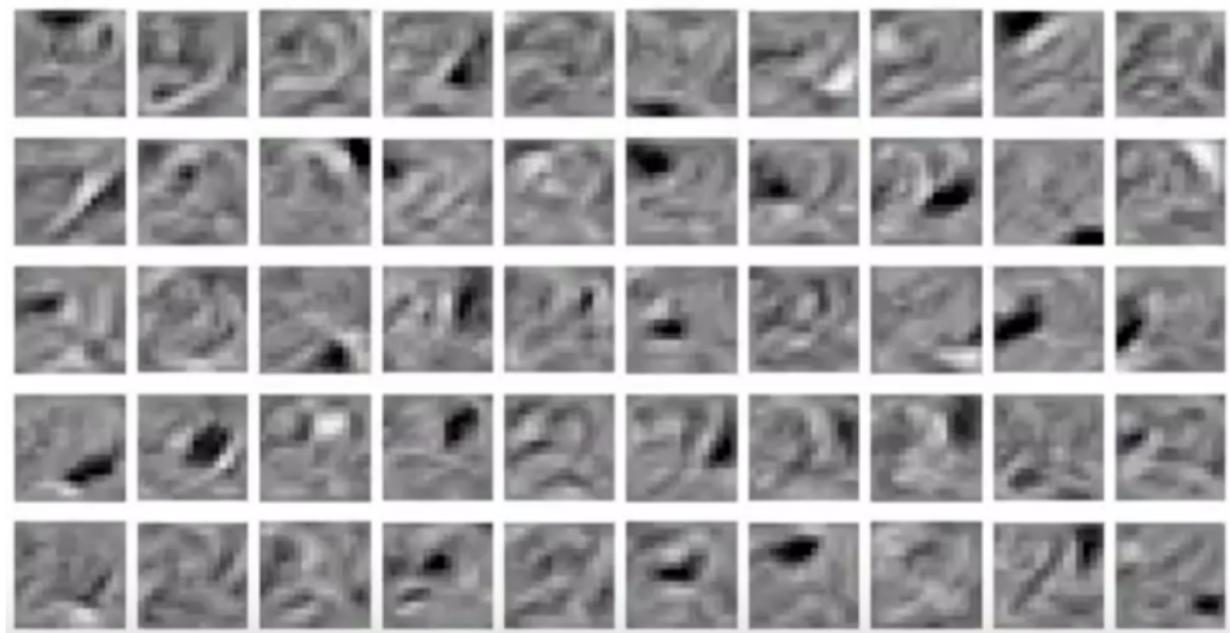
$$\mathbb{P}(h_j = 1 | \vec{x}) = \sigma \left( (\mathbf{W} \vec{x} + \vec{b})_j \right) = \frac{1}{1 + \exp \left( -(\mathbf{W} \vec{x} + \vec{b})_j \right)}$$

# Restricted Boltzmann Machine (RBM)



$$\mathbb{P}(x_i = 1 | \vec{h}) = \sigma \left( (\mathbf{W}^T \vec{h} + \vec{a})_i \right) = \frac{1}{1 + \exp \left( -(\mathbf{W}^T \vec{h} + \vec{a})_i \right)}$$

# RBM feature extraction (MNIST '2' digits)



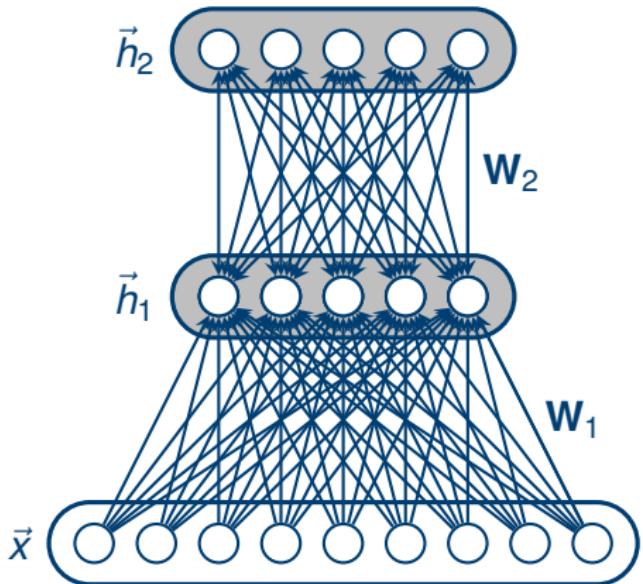
# RBM reconstruction



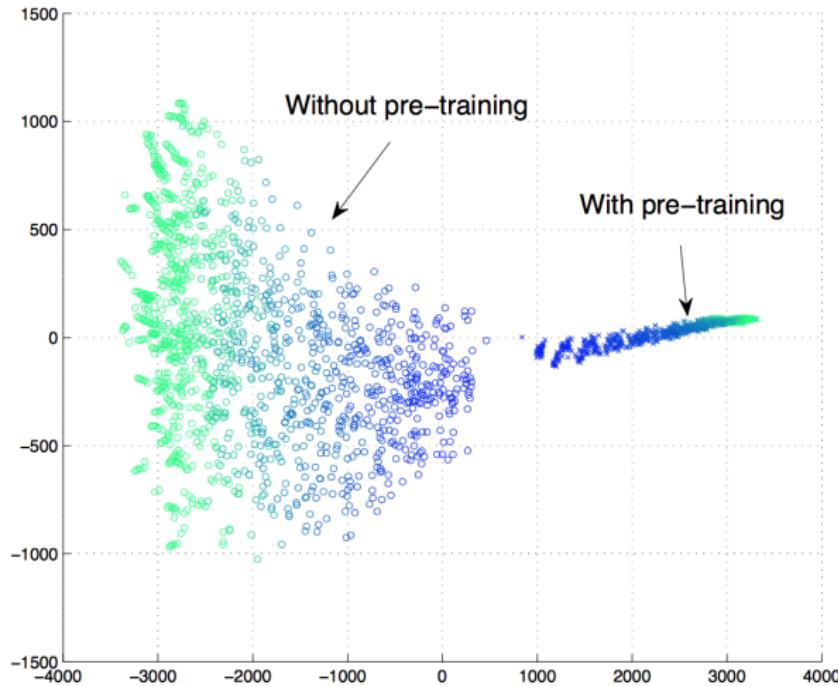
# Deep Belief Network (DBN)

- ▶ If we want to go deeper, we can *stack additional RBMs*.
- ▶ Use the layers trained so far to generate outputs (by sampling or averaging), and use those as inputs for a new RBM. This construction is known as a **deep belief network (DBN)**.
- ▶ Can iteratively stack as many layers as we like!

# Deep Belief Network (DBN)



# Pre-training can help! (*Erhan et al., 2010*)



# Dimensionality reduction

- ▶ We now focus on the general unsupervised problem of **dimensionality reduction**—finding a way to appropriately *compress* our input into a useful “bottleneck” vector of smaller dimensionality (we often call this algorithm an **encoder**).
- ▶ Obvious application to supervised learning: *feeding the output of the bottleneck into a simple classifier* (e.g. kNN, SVM, logistic regression...), perhaps fine-tuning the encoder as well.
- ▶ Fundamentally, dimensionality reduction (along with appropriate *interpretability*) is the **essence** of unsupervised learning—to compress data well, one must first *understand* it!

# Reconstruction strikes again

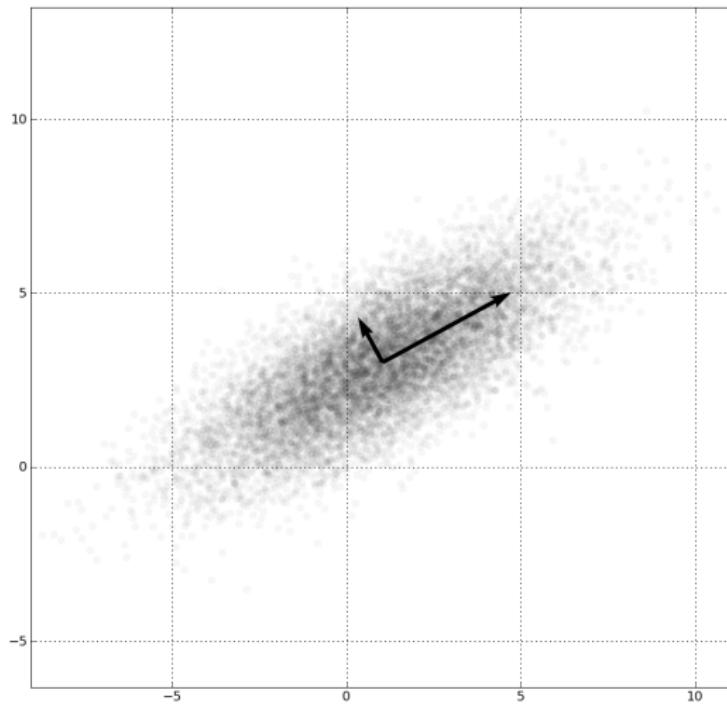
- ▶ Once again—in absence of any other information (that would be contained in labels), the best notion of “usefulness” for the bottleneck is **our ability to reconstruct the input from it.**
- ▶ Broadly speaking, we aim to specify two transformations:
  - ▶ The **encoder** –  $enc : \mathcal{X} \rightarrow \mathcal{Z}$
  - ▶ The **decoder** –  $dec : \mathcal{Z} \rightarrow \mathcal{X}$where  $\mathcal{X}$  and  $\mathcal{Z}$  are the input and code spaces, respectively (these are often simply  $\mathbb{R}^n$  and  $\mathbb{R}^m$  with  $n > m$ ).
- ▶ Then we seek to find parameters of the encoder/decoder that minimise the *reconstruction loss*:

$$\mathcal{L}(\vec{x}) = \|dec(enc(\vec{x})) - \vec{x}\|^2$$

# Principal Component Analysis

- ▶ Perhaps the simplest instance of this framework is the **principal component analysis** (PCA) algorithm.
- ▶ Encode by *projecting* the  $n$ -dimensional data onto a set of  $m$  *orthogonal* axes ( $n \geq m$ ).
- ▶ To preserve the most information, always choose one of the axes to be the direction in which the dataset *has the highest variance!* Then constrain subsequent ones to be orthogonal...
- ▶ Preserve  $m$  axes with highest variance.

# PCA in action



## PCA details

- ▶ Since projection onto orthogonal axes is a linear operation, the PCA encoder can be seen as simple matrix multiplication:

$$enc(\vec{x}) = \mathbf{W}\vec{x}$$

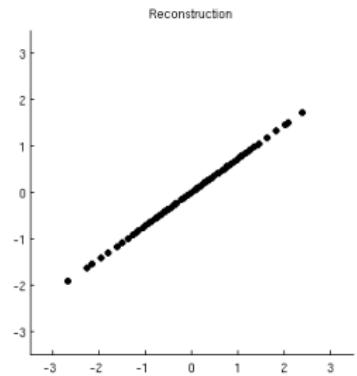
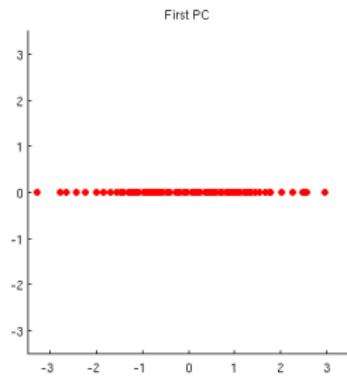
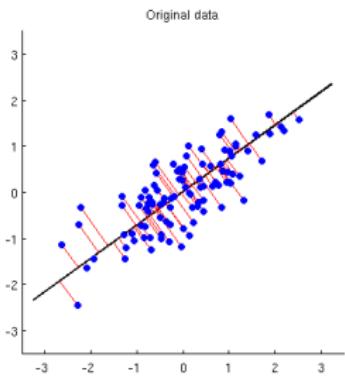
where  $\mathbf{W}$  is of size  $m \times n$ .

- ▶ As this is an *orthogonal* transformation, its inverse (*along retained axes only*) is its matrix's *transpose*:

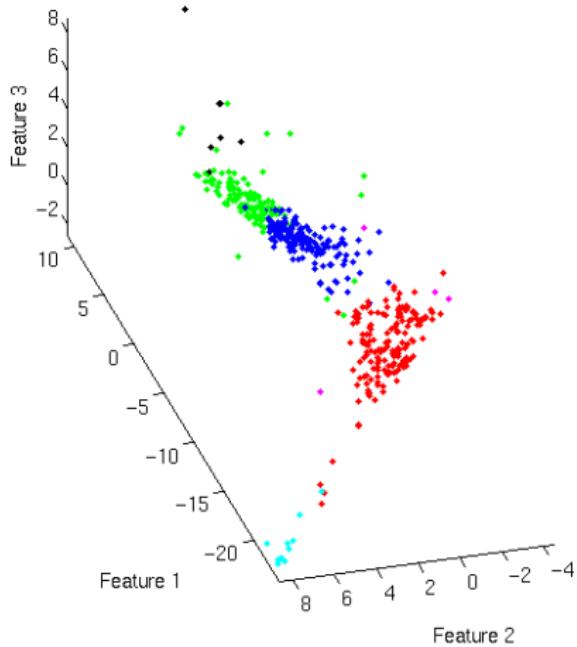
$$dec(\vec{z}) = \mathbf{W}^T\vec{z}$$

- ▶ We therefore seek to choose  $\mathbf{W}$  to minimise  $\|\vec{x} - \mathbf{W}^T\mathbf{W}\vec{x}\|^2$ . Can solve this *explicitly* (using eigenvalue analysis)!

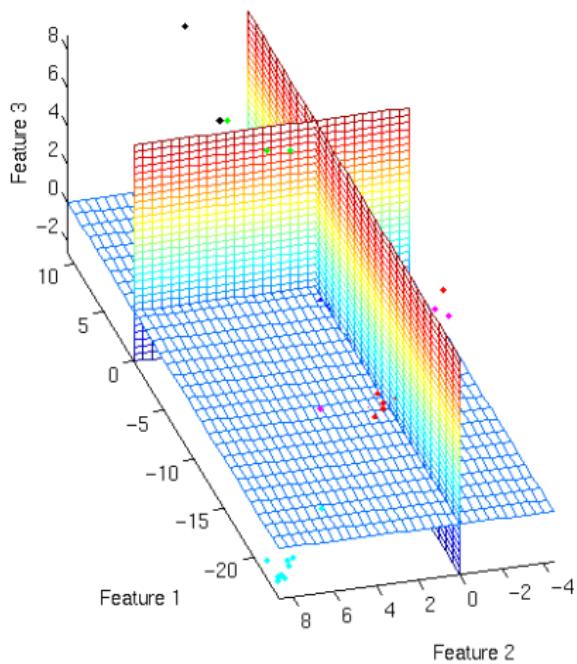
# PCA reconstruction



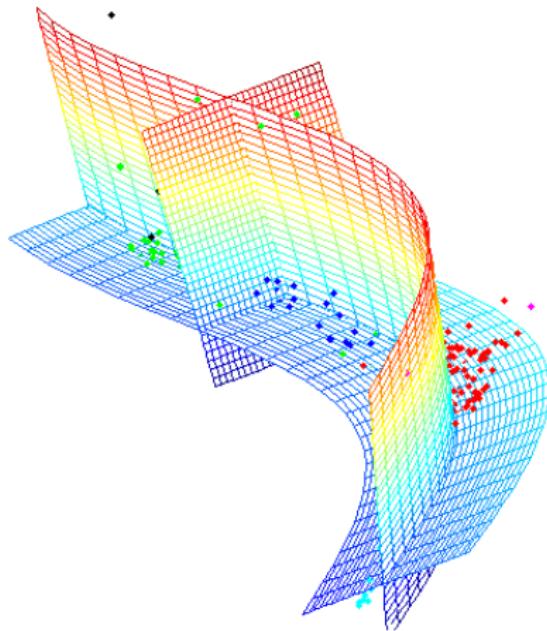
# Limitations of PCA



# Limitations of PCA



# Limitations of PCA



Linear model  $\implies$  Incapable of capturing *nonlinear manifolds!*

## Alternative perspective

- ▶ It should be simple to relate the operations of PCA to those of a *two-layer fully-connected neural network without activations!*
- ▶ This would allow us to work in exactly the same scenario, but train using *backpropagation!*

$$\vec{z} = \text{enc}(\vec{x}) = \mathbf{W}_1 \vec{x} + \vec{b}_1$$

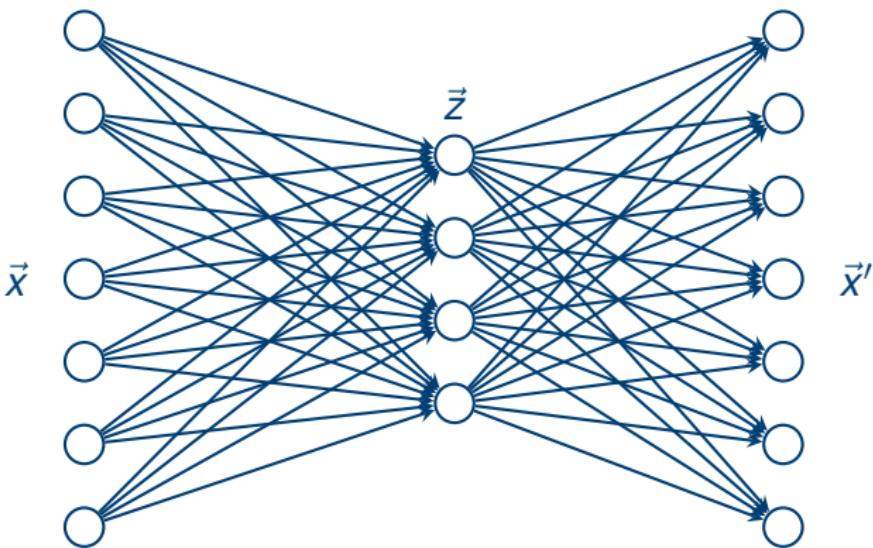
$$\vec{x}' = \text{dec}(\vec{z}) = \mathbf{W}_2 \vec{z} + \vec{b}_2$$

- ▶ Once again, we optimise the *reconstruction loss*

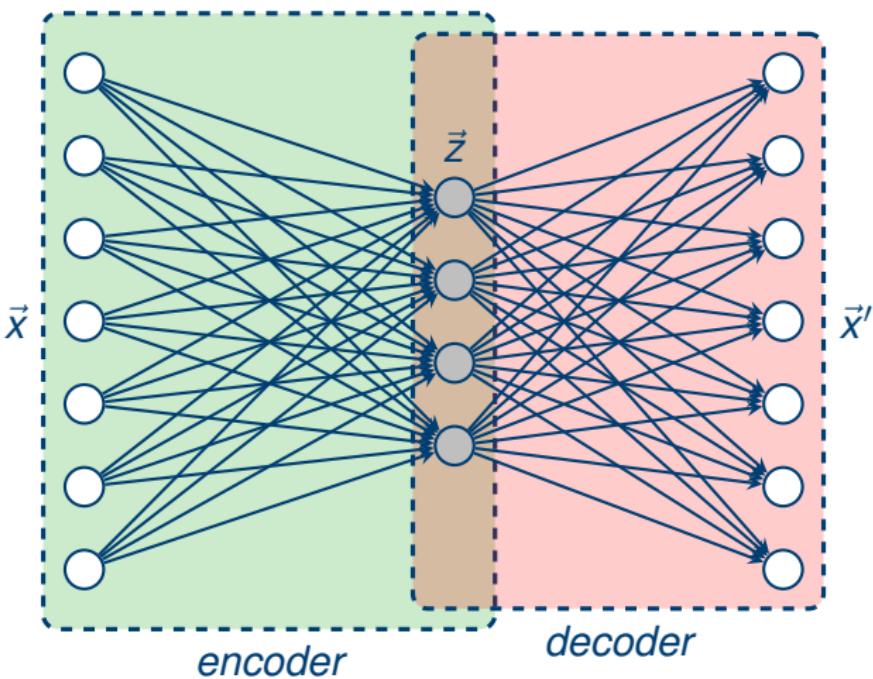
$$\mathcal{L}(\vec{x}') = \|\vec{x}' - \vec{x}\|^2$$

- ▶ We have just built our first **autoencoder!**

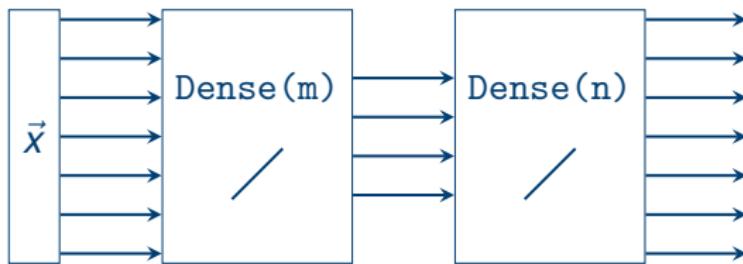
# Autoencoder



# Autoencoder



# Autoencoder in Keras



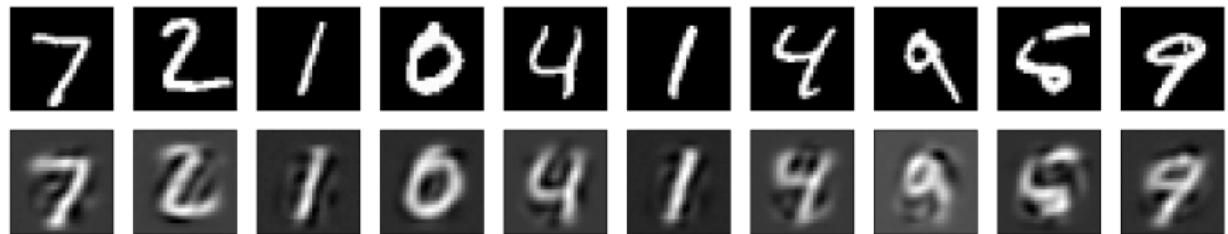
Keras code (for MNIST):

```
x = Input(shape=(784,))
z = Dense(32)(x)
y = Dense(784)(z)
```

# Autoencoder performance on MNIST



# Autoencoder performance on MNIST



## Going deeper...

- ▶ Thus far, our autoencoder was only capable of the same kind of expressivity as PCA.
  - ▶ More so, it was doing so *inefficiently* compared to PCA!
- ▶ However, enabling training by backpropagation means that we can now introduce *depth* and *nonlinearity* into the model!
- ▶ This should allow us to capture complex nonlinear manifolds more accurately...

# Deep autoencoders

- We will introduce an additional layer of depth, ReLU activations, and use *logistic sigmoid* units to reconstruct the image (closer to 1 ~ whiter).

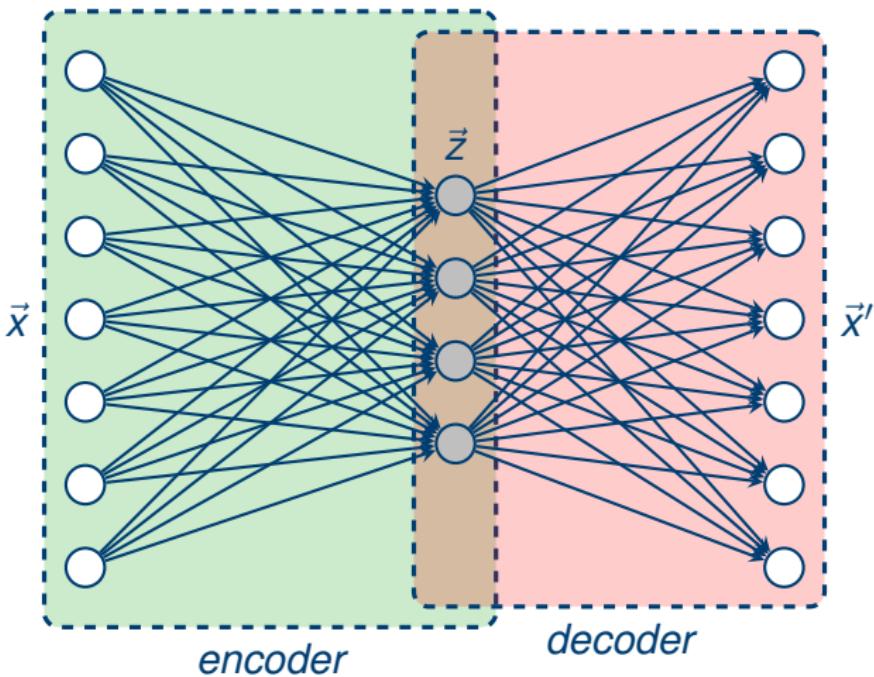
$$\vec{z} = \text{enc}(\vec{x}) = \text{ReLU} \left( \mathbf{W}_2 \text{ReLU} \left( \mathbf{W}_1 \vec{x} + \vec{b}_1 \right) + \vec{b}_2 \right)$$
$$\vec{x}' = \text{dec}(\vec{z}) = \sigma \left( \mathbf{W}_4 \text{ReLU} \left( \mathbf{W}_3 \vec{z} + \vec{b}_3 \right) + \vec{b}_4 \right)$$

- Now we can interpret the output as the probability of each pixel being white—can use *cross-entropy* as the reconstruction loss!

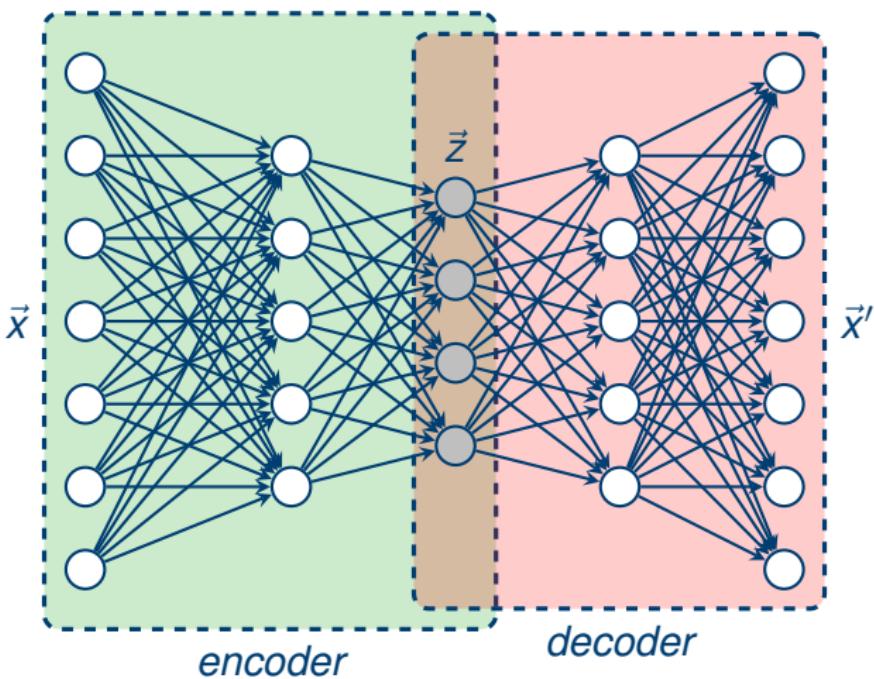
$$\mathcal{L}(\vec{x}') = - \sum_{i=1}^n x_i \log x'_i + (1 - x_i) \log(1 - x'_i)$$

- We now have a **deep autoencoder!**

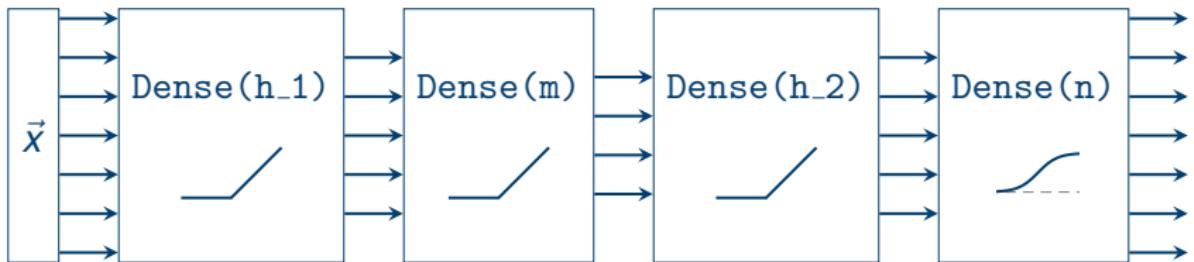
# From autoencoders...



...to deep autoencoders



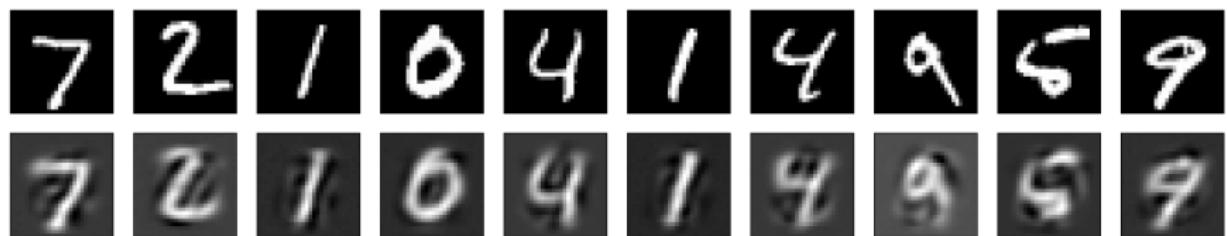
# Deep autoencoder in Keras



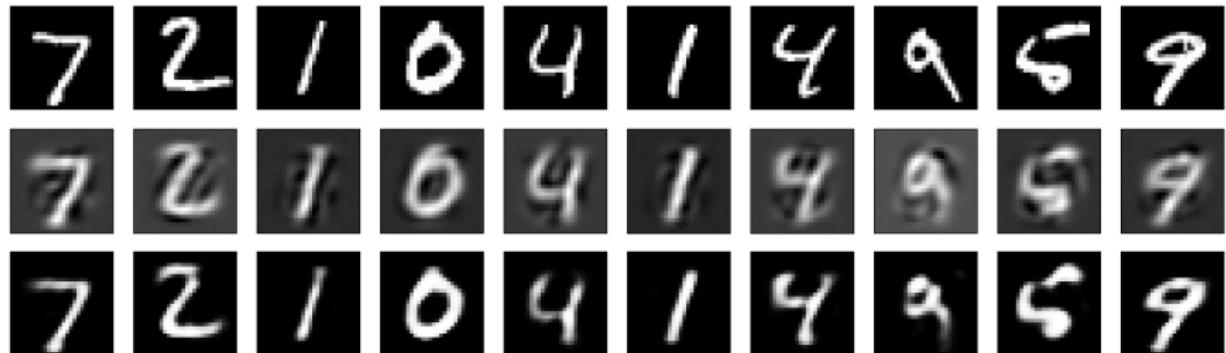
Keras code (for MNIST):

```
x = Input(shape=(784,))  
h_1 = Dense(128, activation='relu')(x)  
z = Dense(32, activation='relu')(h_1)  
h_2 = Dense(128, activation='relu')(z)  
y = Dense(784, activation='sigmoid')(h_2)
```

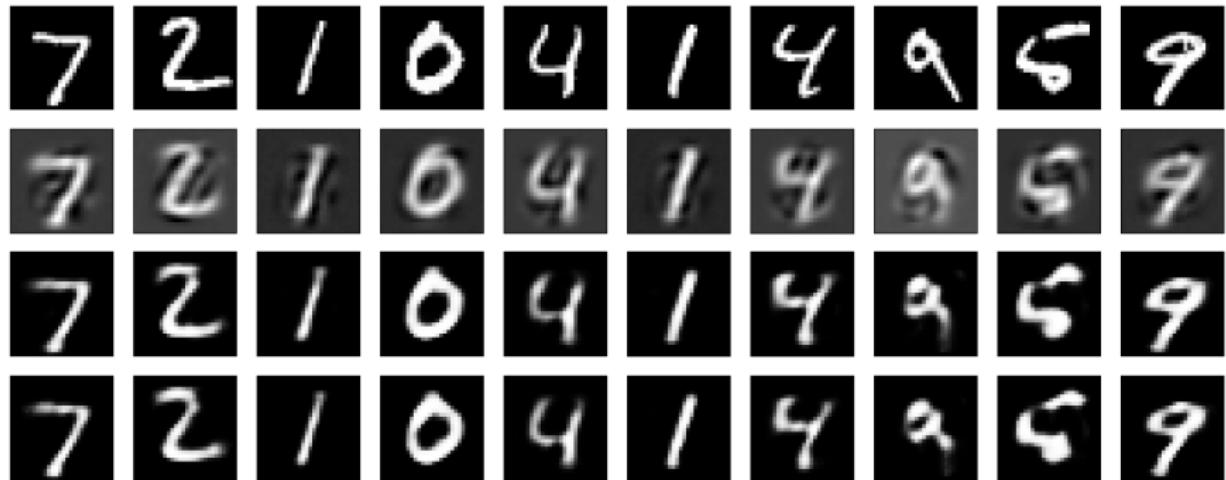
# Deep autoencoder performance on MNIST



# Deep autoencoder performance on MNIST



# Deep(er) autoencoder performance on MNIST



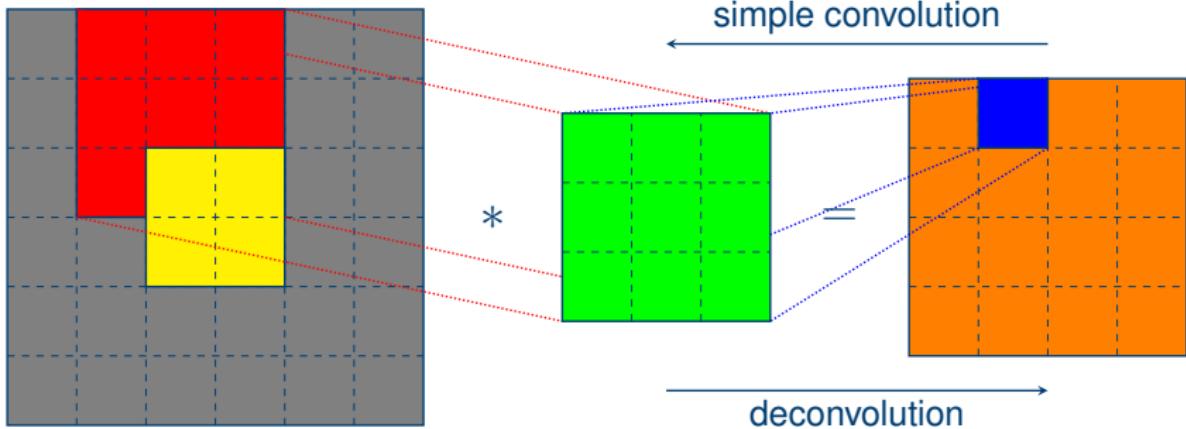
# Reintroducing convolutional layers

- ▶ But... we've been working with **images** all this time! Didn't I say that fully-connected layers are bad for images?
- ▶ They are indeed—MNIST is small ( $28 \times 28 \times 1$ ). We can't go much further with only fully-connected layers.
- ▶ Luckily, inserting convolutional and pooling layers into autoencoders is not a major issue.
- ▶ **How do we *decode*** (especially from a *downsampled* image)?

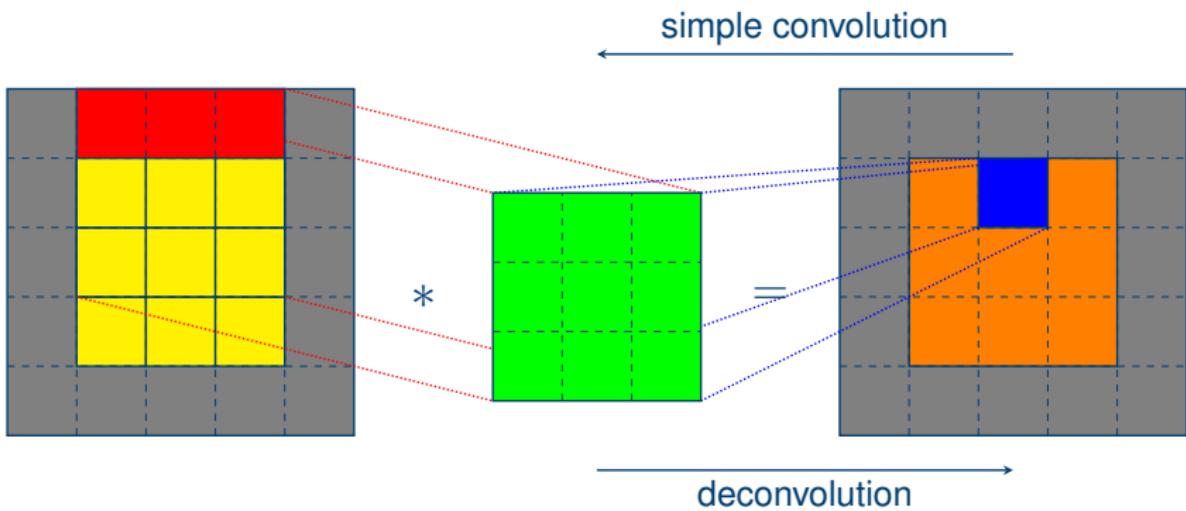
## Towards the *deconvolutional* layer

- ▶ Essentially, we want to transform the *output* of a convolution into something of the shape of its *input*, while maintaining the desired connectivity patterns.
- ▶ Luckily (and perhaps surprisingly), *the backpropagation update of a convolutional layer is **itself a convolution!***
  - ▶ Details omitted, to follow in week 3 (but illustrations incoming)!
- ▶ This means that we can carefully craft a (potentially *strided*) convolutional layer that will behave like a “target” convolutional layer in the backwards direction!
- ▶ This forms the basis of the **deconvolutional** layer, which is extremely useful across a variety of applications of deep learning to computer vision!

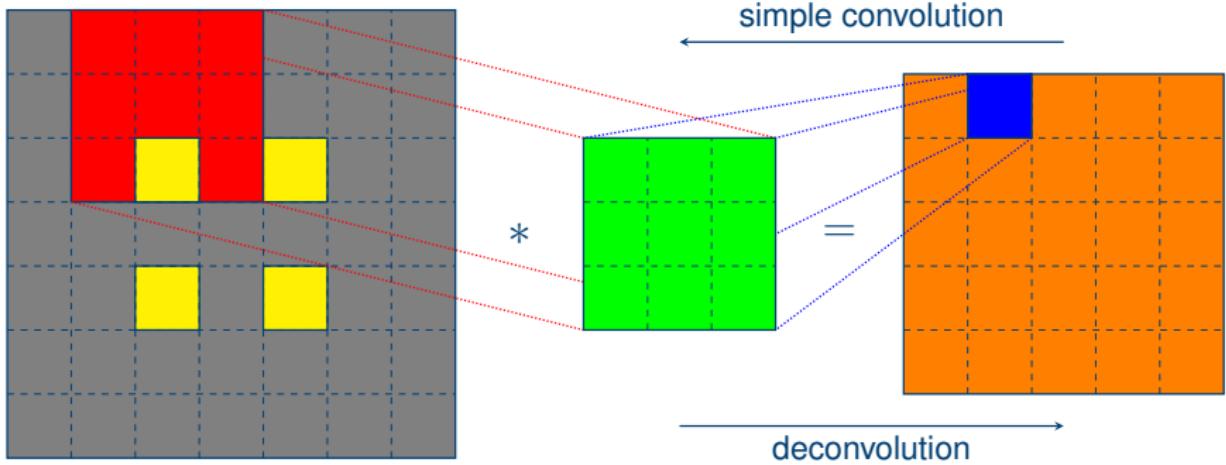
# Deconvolution



# Deconvolution



# Deconvolution



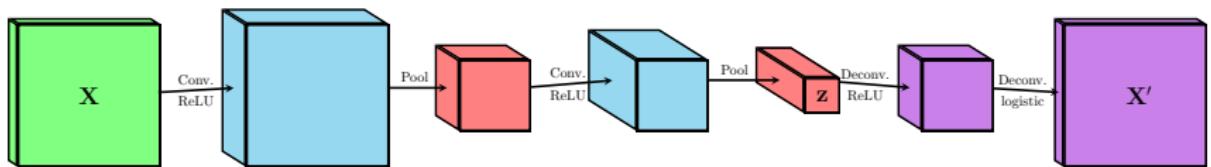
# Convolutional autoencoder

- Now we stack convolutional and pooling layers in the encoder, and deconvolutional layers in the decoder. All other details remain unchanged.

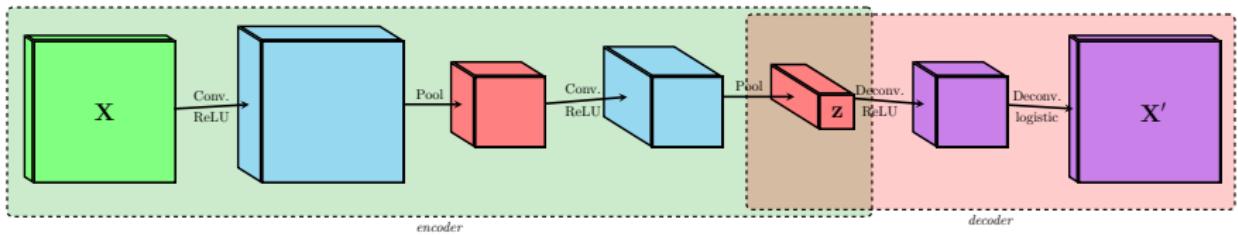
$$\begin{aligned}\mathbf{z} &= \text{enc}(\mathbf{X}) \\ &= \text{Pool}(\text{ReLU}(\text{Conv}(\text{Pool}(\text{ReLU}(\text{Conv}(\mathbf{X}, \mathbf{K}_1))), \mathbf{K}_2)))\end{aligned}$$

$$\mathbf{X}' = \text{dec}(\mathbf{z}) = \sigma(\text{Deconv}(\text{ReLU}(\text{Deconv}(\mathbf{z}, \mathbf{K}_3)), \mathbf{K}_4))$$

# Convolutional autoencoder (CAE)



# Convolutional autoencoder (CAE)



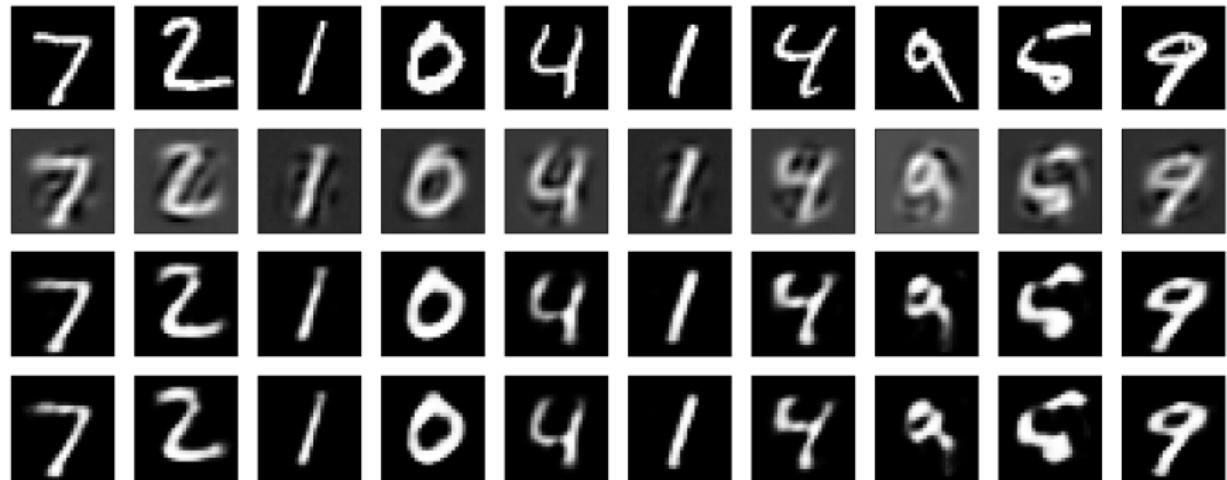
# Convolutional autoencoder in Keras: Encoder

```
x = Input(shape=(28, 28, 1))
x_up = ZeroPadding2D((2, 2))(x)
h_1 = Conv2D(16, (3, 3), padding='same',
activation='relu')(x_up)
p_1 = MaxPooling2D((2, 2))(h_1)
h_2 = Conv2D(8, (3, 3), padding='same',
activation='relu')(p_1)
p_2 = MaxPooling2D((2, 2))(h_2)
h_3 = Conv2D(8, (3, 3), padding='same',
activation='relu')(p_2)
z = MaxPooling2D((2, 2))(h_3)
```

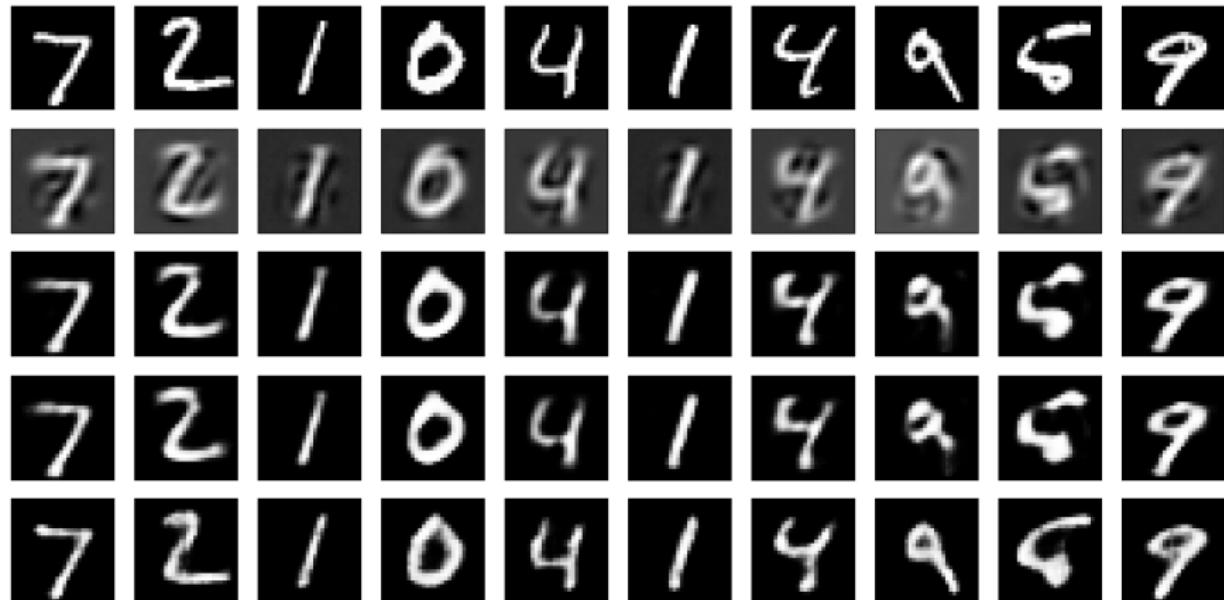
# Convolutional autoencoder in Keras: Decoder

```
h_4 = Conv2DTranspose(8, (3, 3), padding='same',
                     strides=(2, 2), activation='relu')(z)
h_5 = Conv2DTranspose(16, (3, 3), padding='same',
                     strides=(2, 2), activation='relu')(h_4)
y_up = Conv2DTranspose(1, (3, 3), padding='same',
                      strides=(2, 2), activation='sigmoid')(h_5)
y = Cropping2D((2, 2))(y_up)
```

# Convolutional autoencoder performance on MNIST

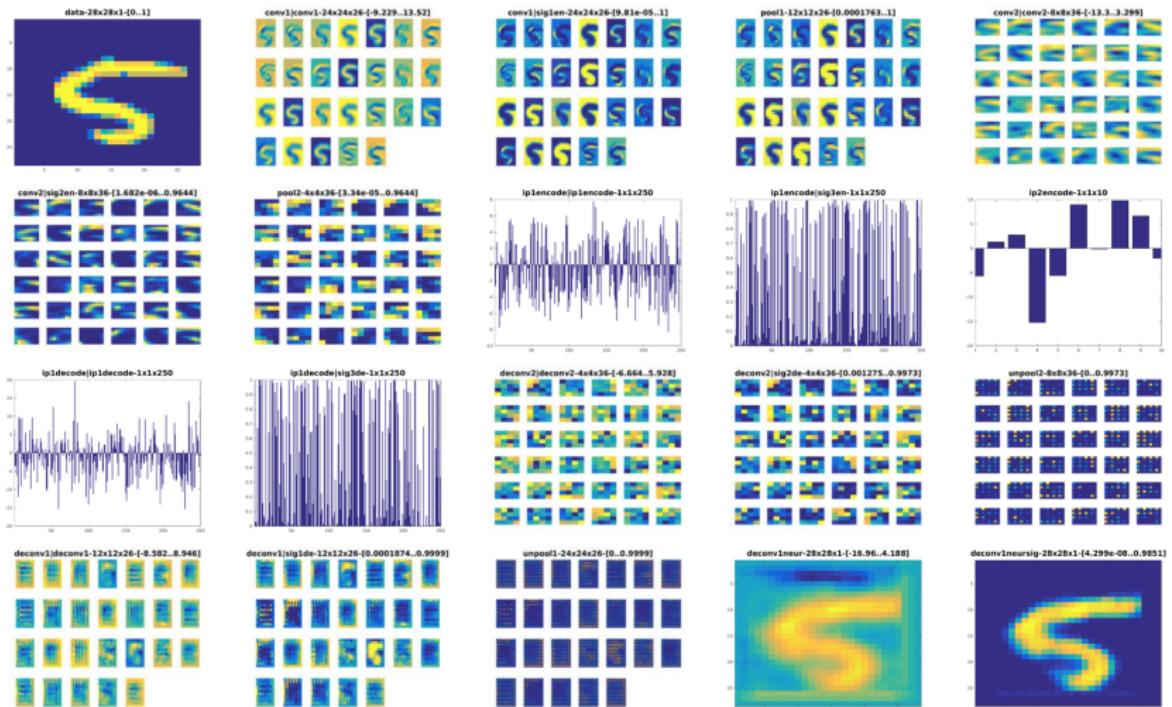


# Convolutional autoencoder performance on MNIST

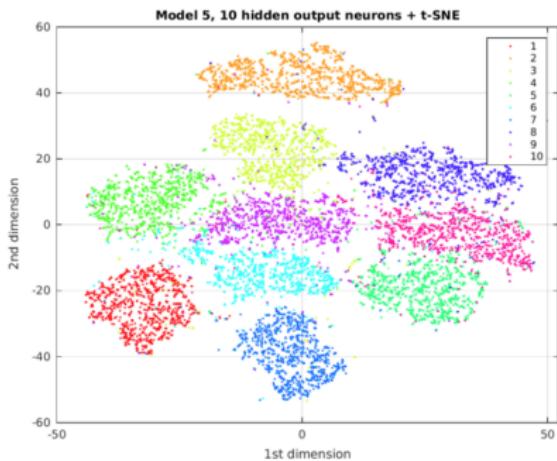
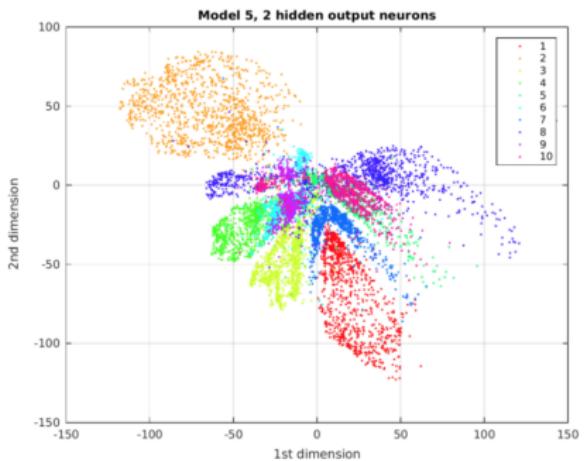


with ***significantly*** fewer parameters ( $\sim 3K$  vs.  $\sim 220K$ )!

# Intermediate layers (*Turchenko et al., 2017*)



# The code is *discriminative* (*Turchenko et al., 2017*)



# Applying autoencoders to eliminating noise

- ▶ One popular application of autoencoders is **denoising**.
- ▶ Namely, we may wish to be able to reconstruct our input while simultaneously *eliminating any noise present within it!*
- ▶ This comes from several motivations:
  - ▶ Real-world data *is* often noisy;
  - ▶ Combatting overfitting the training data;
  - ▶ *Learning more robust representations!*

# Denoising autoencoder

- ▶ Constructing a **denoising autoencoder** (of any kind!) requires modifying two aspects of the framework.
- ▶ Firstly, the input data  $\vec{x}$  is *corrupted* by a corruption process (for images, this could be e.g. injecting Gaussian noise):

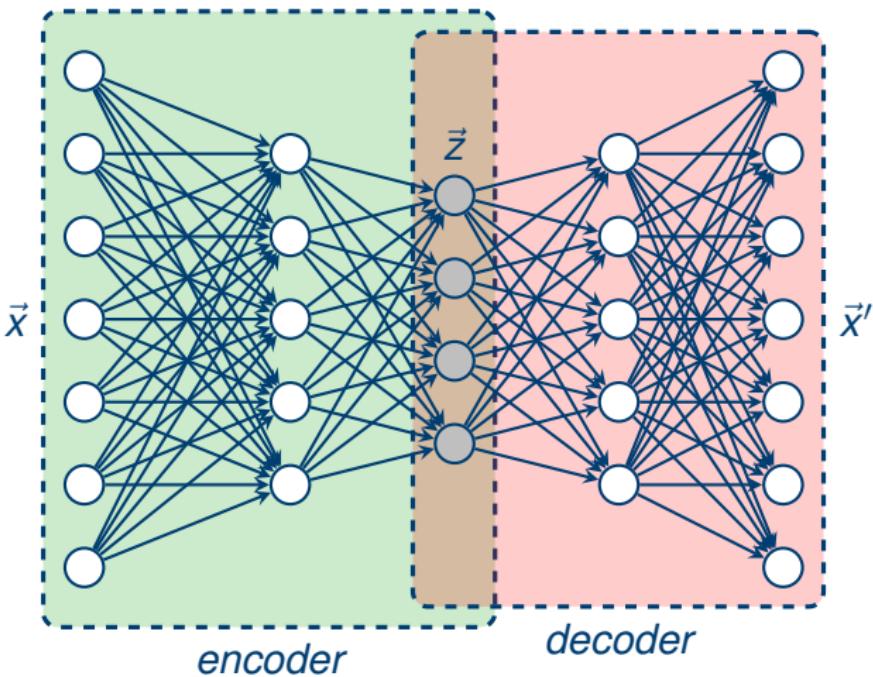
$$\vec{\tilde{x}} = \text{corrupt}(\vec{x})$$

- ▶ Then, the network is judged on how well it reconstructs the *original input* from the *corrupted input*:

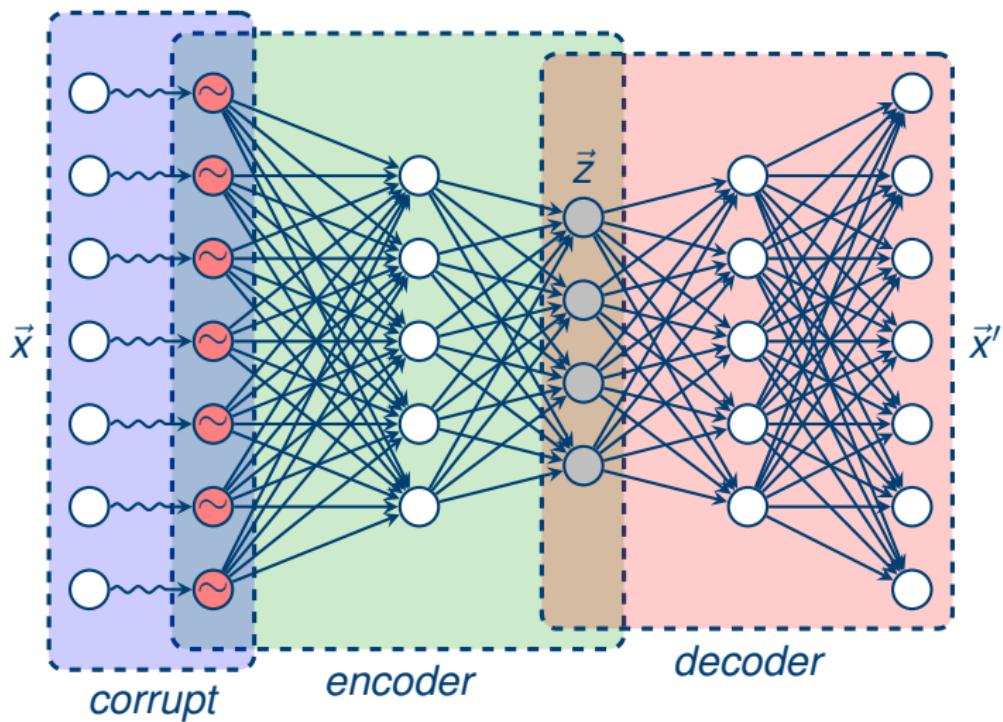
$$\mathcal{L}(\vec{x}) = \| \text{dec}(\text{enc}(\text{corrupt}(\vec{x}))) - \vec{x} \|^2$$

(similarly if cross-entropy is used)

# Denoising autoencoder (DAE)



# Denoising autoencoder (DAE)



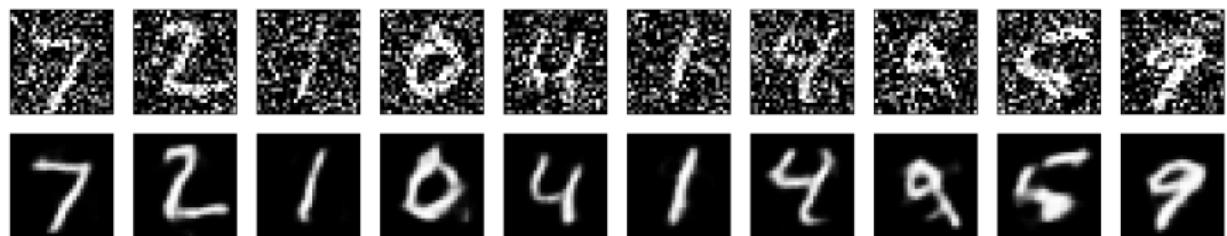
# Denoising convolutional autoencoder in Keras

```
x = Input(shape=(28, 28, 1))
x_c = GaussianNoise(0.5)(x)
h_1 = Conv2D(32, (3, 3), padding='same',
activation='relu')(x_c)
p_1 = MaxPooling2D((2, 2))(h_1)
h_2 = Conv2D(32, (3, 3), padding='same',
activation='relu')(p_1)
z = MaxPooling2D((2, 2))(h_2)
h_3 = Conv2DTranspose(32, (3, 3), padding='same',
strides=(2, 2), activation='relu')(z)
y = Conv2DTranspose(1, (3, 3), padding='same',
strides=(2, 2), activation='sigmoid')(h_3)
```

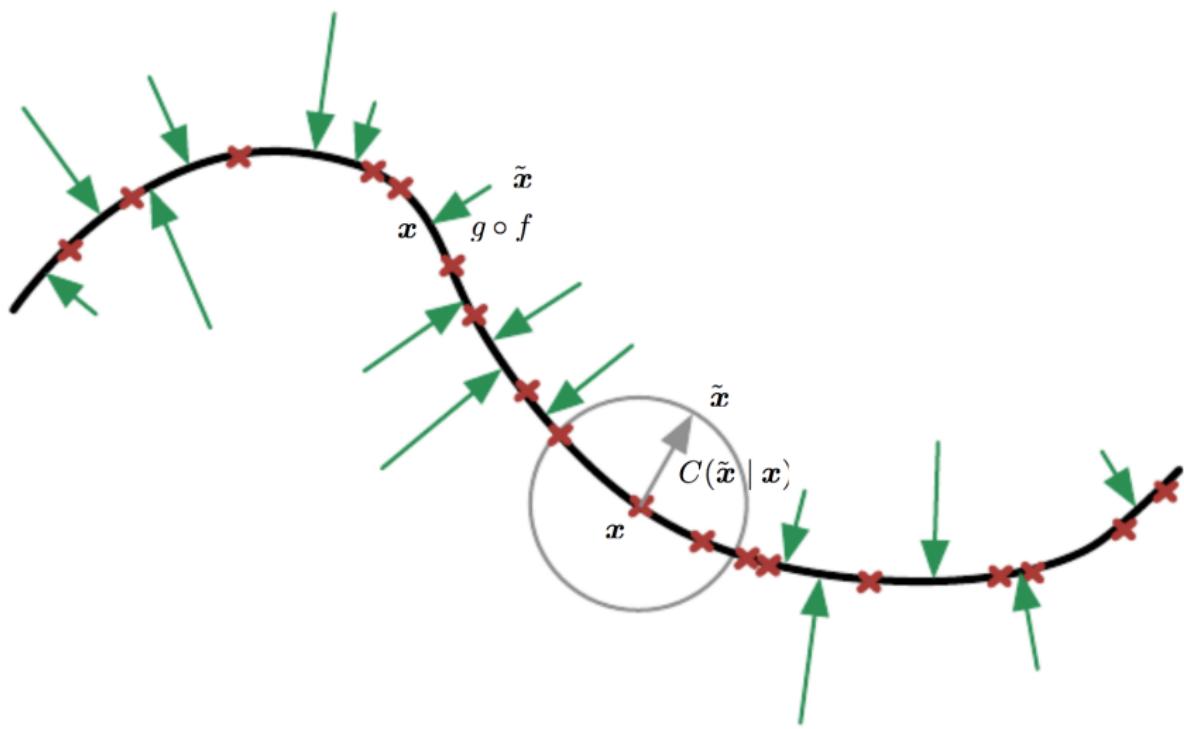
# Denoising autoencoder performance on MNIST



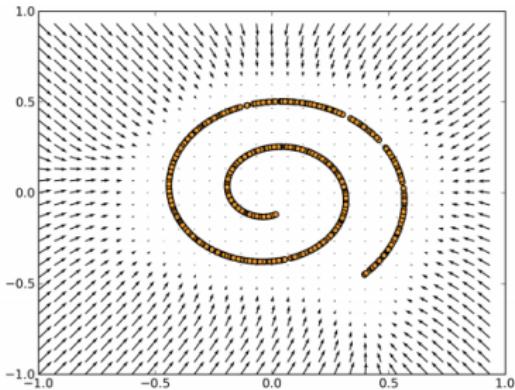
# Denoising autoencoder performance on MNIST



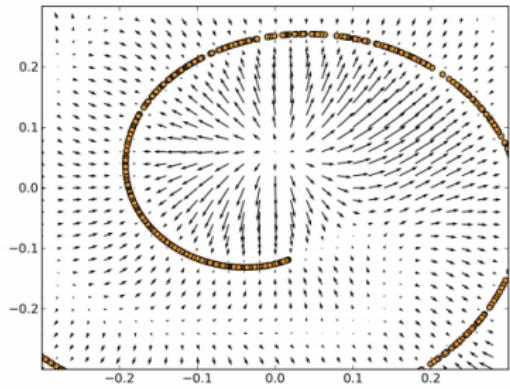
# Robust representations! (*Goodfellow et al., 2016*)



# Robust representations! (*Alain and Bengio, 2014*)



(a)  $r(x) - x$  vector field, acting as sink, zoomed out

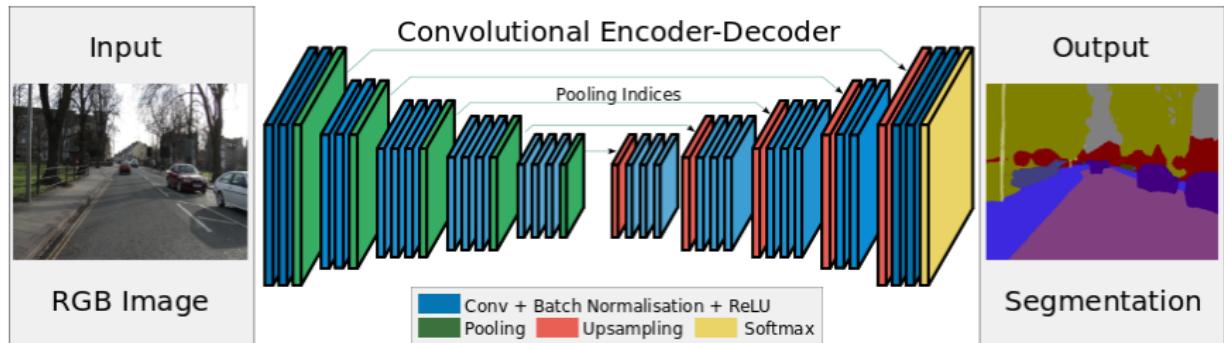


(b)  $r(x) - x$  vector field, close-up

## Aside: encoder-decoder architectures

- ▶ Strictly speaking, a denoising autoencoder is not performing **autoencoding**, i.e. it's not trying to reconstruct its exact input.
- ▶ We can generalise this idea further, to make our network have a similar structure to an autoencoder, but actually perform some useful *transformation* on the input!
- ▶ This gives rise to **encoder-decoder architectures**, which are now used across the board for tasks such as segmentation.

# An encoder-decoder architecture for segmentation



**SegNet** (*Kendall et al., 2015*)

# Generative modelling returns

- ▶ To conclude our exploration of autoencoders, we will focus our attention on *generative models*.
- ▶ The autoencoder architectures covered so far will do a good job at learning an appropriate dimensionality reduction of inputs similar to our training set's.
- ▶ But what does this *tell us* about the underlying properties of the data? Can we plug in arbitrary codes  $\vec{z}$  into the decoder and expect useful results?

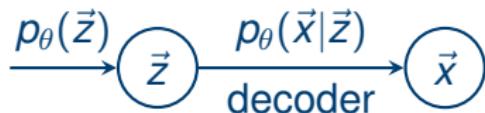
**“What I cannot create, I do not understand.”** —Richard Feynman

# A probabilistic generative framework

- ▶ We will think of the code,  $\vec{z}$ , as an object that captures the essential properties of our input data,  $\vec{x}$ .
- ▶ We can make it easy to generate new, useful, codes—namely, we can engineer the *prior* distribution of codes  $\mathbb{P}(\vec{z})$  to be a “tractable” probability distribution (such as a  $\mathcal{N}(\vec{0}, \mathbf{I})$ , the unit Gaussian).
- ▶ Once we have the code, generating data implies sampling the probability distribution  $\mathbb{P}(\vec{x}|\vec{z})$ . This can also be designed to be tractable (e.g. a learnt decoder neural network).
- ▶ This allows us to generate new data points!

# A probabilistic generative framework

- ▶ Let's assume that the code prior  $\mathbb{P}(\vec{z})$  and the generating distribution  $\mathbb{P}(\vec{x}|\vec{z})$  are both parametrised by  $\theta$  (these could be, say, the weights of a neural network).
- ▶ We will denote the prior by  $p_\theta(\vec{z})$  and the generating distribution by  $p_\theta(\vec{x}|\vec{z})$  to make this explicit.
- ▶ What do we have so far?



# Learning the generative framework

- ▶ Note that, having specified  $p_\theta(\vec{z})$  and  $p_\theta(\vec{x}|\vec{z})$ , we can also derive an expression for  $\mathbb{P}(\vec{x}) = p_\theta(\vec{x})!$
- ▶ This roughly corresponds to: “how likely is it that our model will generate  $\vec{x}$ ?” and sounds like a great objective to maximise!
- ▶ We could train our model to maximise  $p_\theta(\vec{x})$  over all the samples  $\vec{x}$  in our training set.

## Maximising the *evidence*

- ▶ Unfortunately, computing this quantity (sometimes called the *evidence*) requires *integrating over all possible codes*:

$$p_{\theta}(\vec{x}) = \mathbb{P}(\vec{x}) = \int_{\vec{z} \in \mathcal{Z}} \mathbb{P}(\vec{x}, \vec{z}) d\vec{z} = \int_{\vec{z} \in \mathcal{Z}} p_{\theta}(\vec{x}|\vec{z}) p_{\theta}(\vec{z}) d\vec{z}$$

and this is intractable in all except the simplest of cases!

- ▶ We could try approximating it using *Monte Carlo* methods, but this does not scale for large datasets and large networks.
- ▶ It is possible to make this objective work—and we will see how! For now, we need to address an even more serious issue...

# Performing inference

- ▶ For *inference* purposes, we also would very much like to be able to *attach codes* to known inputs  $\vec{x}$ .
- ▶ This would allow us to use the code for other purposes, such as *dimensionality reduction*, obtaining new inputs similar to  $\vec{x}$  (e.g. for *data augmentation*) by modifying the code, etc.
- ▶ This involves sampling codes from the *posterior distribution*  $\mathbb{P}(\vec{z}|\vec{x})$ , and is almost always **insanely hard!**

# Why is the posterior *so hard*?

- ▶ At a glance, it might seem like we could use Bayes' theorem to help us evaluate the posterior:

$$p_{\theta}(\vec{z}|\vec{x}) = \mathbb{P}(\vec{z}|\vec{x}) = \frac{\mathbb{P}(\vec{x}|\vec{z})\mathbb{P}(\vec{z})}{\mathbb{P}(\vec{x})} = \frac{p_{\theta}(\vec{x}|\vec{z})p_{\theta}(\vec{z})}{p_{\theta}(\vec{x})}$$

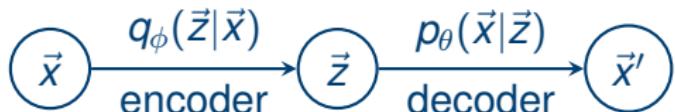
- ▶ However, the pesky *evidence* reappears in the denominator :( and this time we need to *sample* from something involving it, not just evaluate it . . .

# Variational inference

- ▶ This issue is circumvented by employing **variational inference**.
- ▶ We accept that the posterior is hard, and rather than explicitly sampling it, we choose to sample from something *simpler*.
- ▶ Generally, we choose to approximate the posterior with a *recognition model*,  $q_\phi(\vec{z}|\vec{x})$ , parametrised by  $\phi$ . Typically, this will also be a neural network, with  $\phi$  being its weights!
- ▶ The parameters  $\phi$  should be selected to make  $q_\phi(\vec{z}|\vec{x})$  as close to the true posterior  $p_\theta(\vec{z}|\vec{x})$  as possible!

# Variational autoencoder (*Kingma & Welling, 2015*)

- Our model of the world now looks like this:



- Looks like an *autoencoder*, doesn't it?
- Indeed—when  $p_\theta$  and  $q_\phi$  are neural networks, this is the general setup of a **variational autoencoder** (VAE).
- The loss function is now a bit more complicated...

# Towards a VAE loss

- As discussed, a VAE has **two objectives** that need to be *simultaneously satisfied*. For a training example,  $\vec{x}$ :
  - We want to make our *decoder* highly likely to generate this example—this implies **maximising**

$$\log p_{\theta}(\vec{x})$$

- Simultaneously, we would like our *encoder* to not stray too far from the *true posterior*  $p_{\theta}(\vec{z}|\vec{x})$ . This implies **minimising**

$$D_{KL}(q_{\phi}(\vec{z}|\vec{x}) \| p_{\theta}(\vec{z}|\vec{x}))$$

where  $D_{KL}$  is the Kullback-Leibler (KL) divergence:

$$D_{KL}(q(x) \| p(x)) = \int_{x \in \mathcal{X}} q(x) \log \frac{q(x)}{p(x)} dx$$

# The evidence lower bound

- Combined, these objectives give us the **evidence lower bound** (ELBO), which our network needs to maximise:

$$ELBO(\theta, \phi) = \log p_\theta(\vec{x}) - D_{KL}(q_\phi(\vec{z}|\vec{x})\|p_\theta(\vec{z}|\vec{x}))$$

- The troublesome posterior  $p_\theta(\vec{z}|\vec{x})$  is **still** here...
- Luckily, we can rewrite ELBO in a form that *completely eliminates the posterior!*

$$ELBO(\theta, \phi) = \underbrace{\mathbb{E}_{\vec{z} \sim q_\phi(\vec{z}|\vec{x})} [\log p_\theta(\vec{x}|\vec{z})]}_{\text{Reconstruction accuracy}} - \underbrace{D_{KL}(q_\phi(\vec{z}|\vec{x})\|p_\theta(\vec{z}))}_{\text{Regularisation}}$$

Derivation to follow in the next two slides—likely omitted.

# ELBO derivation

Recall:  $ELBO(\theta, \phi) = \log p_\theta(\vec{x}) - D_{KL}(q_\phi(\vec{z}|\vec{x})\|p_\theta(\vec{z}|\vec{x}))$

$$\begin{aligned}\log p_\theta(\vec{x}) &= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log p_\theta(\vec{x}) d\vec{z} \\&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{p_\theta(\vec{z}|\vec{x})} d\vec{z} \\&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} \frac{q_\phi(\vec{z}|\vec{x})}{p_\theta(\vec{z}|\vec{x})} d\vec{z} \\&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} d\vec{z} + \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{q_\phi(\vec{z}|\vec{x})}{p_\theta(\vec{z}|\vec{x})} d\vec{z} \\&= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} d\vec{z} + D_{KL}(q_\phi(\vec{z}|\vec{x})\|p_\theta(\vec{z}|\vec{x}))\end{aligned}$$

Looks like we can cancel out the term containing the posterior! :)

# ELBO derivation

Recall:  $ELBO(\theta, \phi) = \log p_\theta(\vec{x}) - D_{KL}(q_\phi(\vec{z}|\vec{x})\|p_\theta(\vec{z}|\vec{x}))$

$$\log p_\theta(\vec{x}) = \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} d\vec{z} + D_{KL}(q_\phi(\vec{z}|\vec{x})\|p_\theta(\vec{z}|\vec{x}))$$

Finally,

$$\begin{aligned} ELBO(\theta, \phi) &= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}, \vec{z})}{q_\phi(\vec{z}|\vec{x})} d\vec{z} \\ &= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{p_\theta(\vec{x}|\vec{z})p_\theta(\vec{z})}{q_\phi(\vec{z}|\vec{x})} d\vec{z} \\ &= \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log p_\theta(\vec{x}|\vec{z}) d\vec{z} - \int_{\vec{z} \in \mathcal{Z}} q_\phi(\vec{z}|\vec{x}) \log \frac{q_\phi(\vec{z}|\vec{x})}{p_\theta(\vec{z})} d\vec{z} \\ &= \boxed{\mathbb{E}_{\vec{z} \sim q_\phi(\vec{z}|\vec{x})} [\log p_\theta(\vec{x}|\vec{z})] - D_{KL}(q_\phi(\vec{z}|\vec{x})\|p_\theta(\vec{z}))} \end{aligned}$$

# The ELBO function—analysed

$$ELBO(\theta, \phi) = \underbrace{\mathbb{E}_{\vec{z} \sim q_{\phi}(\vec{z}|\vec{x})} [\log p_{\theta}(\vec{x}|\vec{z})]}_{\text{Reconstruction accuracy}} - \underbrace{D_{KL}(q_{\phi}(\vec{z}|\vec{x}) \| p_{\theta}(\vec{z}))}_{\text{Regularisation}}$$

- ▶ The first term can be interpreted as the usual *autoencoder reconstruction loss*—given input  $\vec{x}$ , use the encoder to sample a code  $\vec{z}$  from  $q_{\phi}(\vec{z}|\vec{x})$ , then use the decoder to compute  $p_{\theta}(\vec{x}|\vec{z})$ . We want to maximise this value!
- ▶ Minimising the second term forces the distribution of generated codes to remain close to the prior—prevents the network from “cheating” (by, e.g., assigning distant codes to every example)!

## Practical choice of $p_\theta(\vec{x}|\vec{z})$

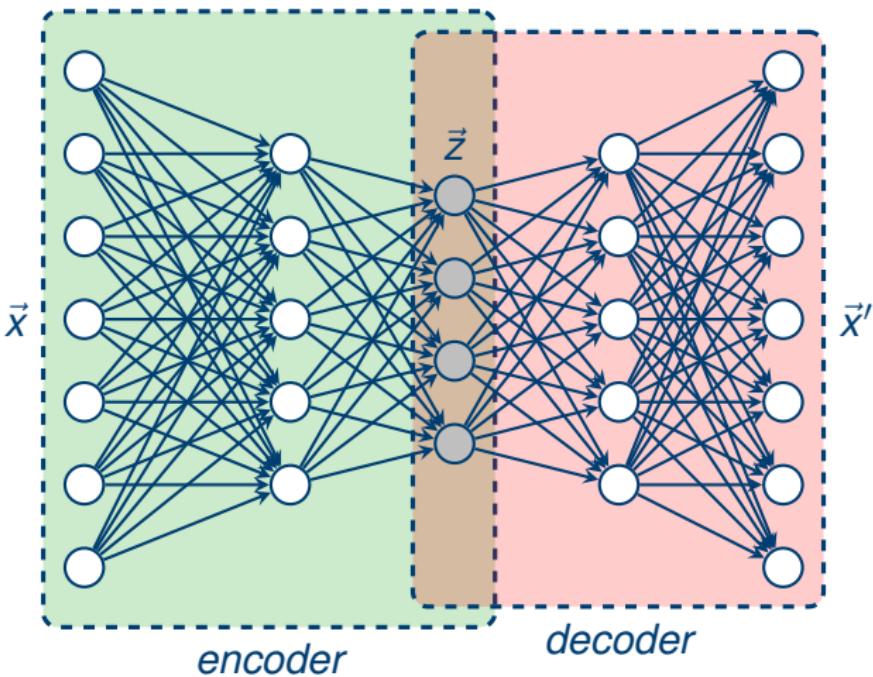
- ▶ Now I will present a specific example of a VAE implementation (which is also most common).
- ▶ For the decoder, we will utilise a neural network (with weights  $\theta$ ) of exactly the same kind as before—with **logistic output units**; these can compute a reconstruction,  $\vec{x}'$ .
- ▶ We can then use the cross-entropy of  $\vec{x}$  and  $\vec{x}'$  as the reconstruction loss.

## Practical choice of $q_\phi(\vec{z}|\vec{x})$

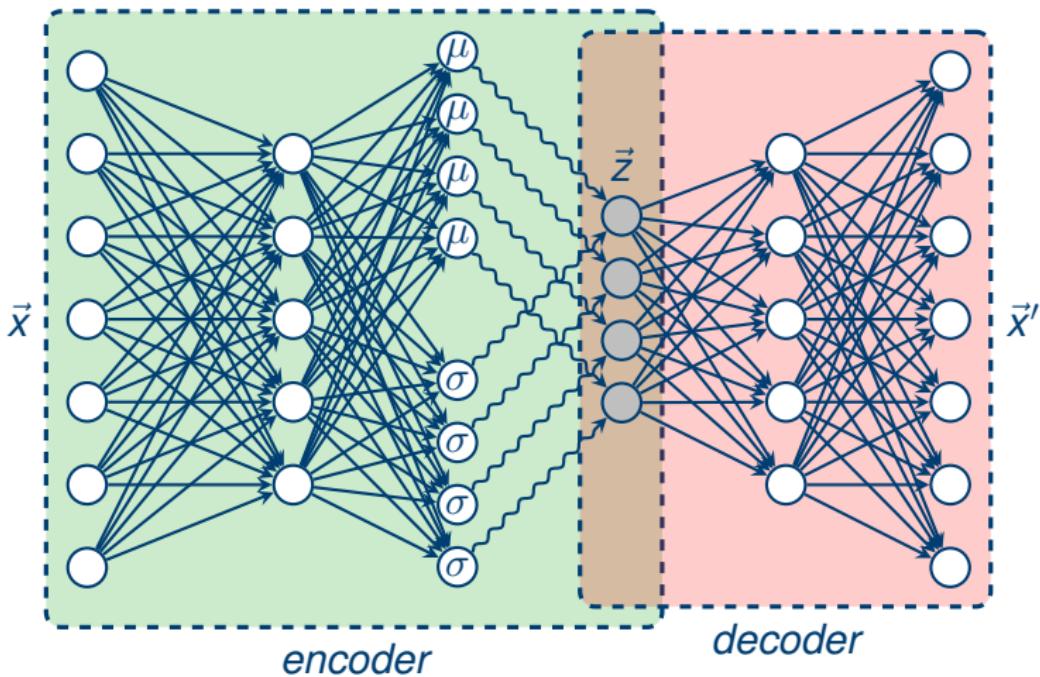
- ▶ For the encoder, we will use  $m$  Gaussian distributions to generate each of the  $m$  elements of the code.
- ▶ The parameters of these distributions ( $\vec{\mu}, \vec{\sigma}$ ) are computed by a neural network (with weights  $\phi$  and **linear output units**). This implies  $q_\phi(\vec{z}|\vec{x}) = \mathcal{N}(\vec{\mu}, \vec{\sigma}^2 \mathbf{I})$ .
- ▶ If we use the prior  $p_\theta(\vec{z}) = \mathcal{N}(\vec{0}, \mathbf{I})$ , then the regularisation term has an expression we can easily work with!

$$-D_{KL}(\mathcal{N}(\vec{\mu}, \vec{\sigma}^2 \mathbf{I}) \| \mathcal{N}(\vec{0}, \mathbf{I})) = \sum_{j=1}^m 1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2$$

# Variational autoencoder (VAE)



# Variational autoencoder (VAE)



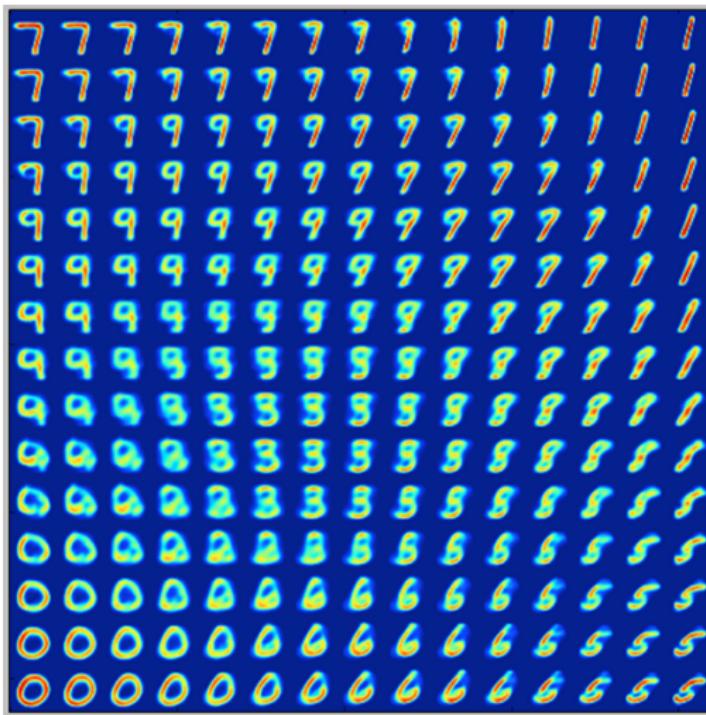
# Reparametrisation trick

- ▶ Naïvely plugging in the sampling operation into the encoder *will not work*, because the sampling operation **has no gradient!**
- ▶ To make it work, we can instead sample upfront  $\vec{\varepsilon} \sim \mathcal{N}(\vec{0}, \mathbf{I})$ , feed it as an *external input*, and transform appropriately:

$$\vec{z} = \vec{\mu} + \vec{\varepsilon} \odot \vec{\sigma}$$

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(batch_size, m),
                               mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var) * epsilon
z = Lambda(sampling)([z_mean, z_log_var])
```

# Generative power of VAEs on MNIST ( $m = 2$ )



# Using VAEs to generate fake faces



<https://youtu.be/XNZIN7Jh3Sg>

# Combatting *entanglement*

Can we *make sense* of what the code **represents**?  
What happens if we *shift* one of the code elements slightly?



# Tackling the black box problem

- ▶ Neural networks have achieved state-of-the-art results across a variety of domains, but *interpreting* their exact mode of operation remains difficult.
- ▶ For some domains (e.g. autonomous vehicles and medicine), **interpretability is key!**
- ▶ Autoencoders typically suffer from the **entanglement** problem—each code element encodes “a little bit of everything” about the output! Shifting one code element ends up blurring the entire result...
- ▶ Making sense of entangled codes is **hard!**

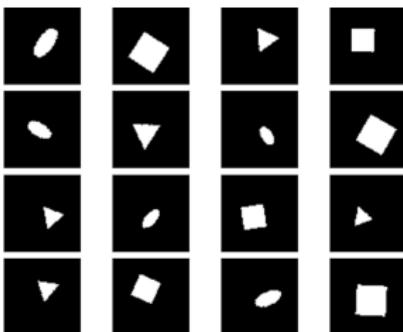
## Disentangled VAE (*Higgins et al., 2016*)

- ▶ Disentanglement parameter  $\beta \geq 0$  – controls the relative importance of the reconstruction accuracy and the regularisation error.
  - ▶ Scales the pressure put on  $q_\phi(\vec{z}|\vec{x})$  to approximate  $p_\theta(\vec{z})$ :

$$\mathcal{L}(\theta, \phi) = \mathbb{E}_{\vec{z} \sim q_\phi(\vec{z}|\vec{x})} [\log p_\theta(\vec{x}|\vec{z})] - \bigcircledcirc \beta D_{KL}(q_\phi(\vec{z}|\vec{x}) \| p_\theta(\vec{z}))$$

- ▶ Under some *continuity* assumptions in the training data, this model is capable of learning to **disentangle** its representations (for large enough  $\beta$ )!
- ▶ (Recall: in our case,  $p_\theta(\vec{z}) = \mathcal{N}(\vec{0}, \mathbf{I})$ —so the prior has *no covariance* between code elements!)

## Example Toy Problem (*Peychev et al., unpublished*)

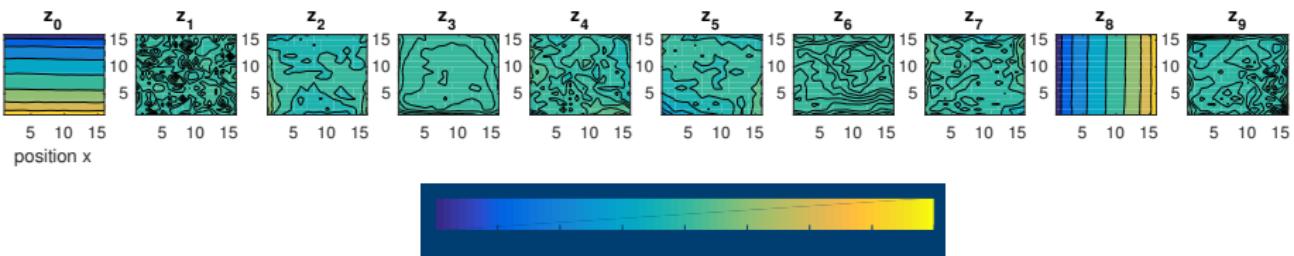


267,021 synthetically generated binary images in total (size: 64 x 64) after removing duplicates. Input generative factors:

- ▶ Shape (ellipse, square, triangle)
- ▶ Position X and Y (16 values each)
- ▶ Scale (6 values)
- ▶ Rotation (60 values over the  $[0; \pi]$  range)

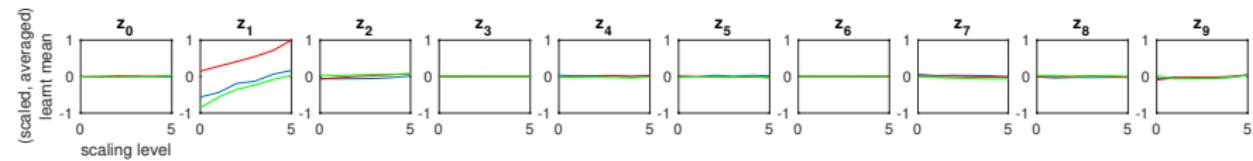
# Results ( $\beta = 4$ )

Learnt means of each code element  $z_i$  as a function of all 16x16 **locations**, averaged across objects, rotations and scales.

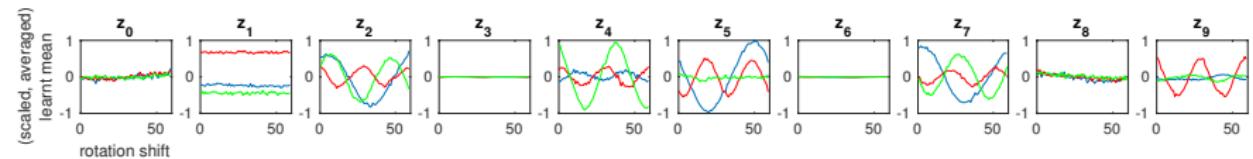


# Results ( $\beta = 4$ )

Learnt means of each code element  $z_i$  as a function of the **scale** factor, averaged across rotations and positions.



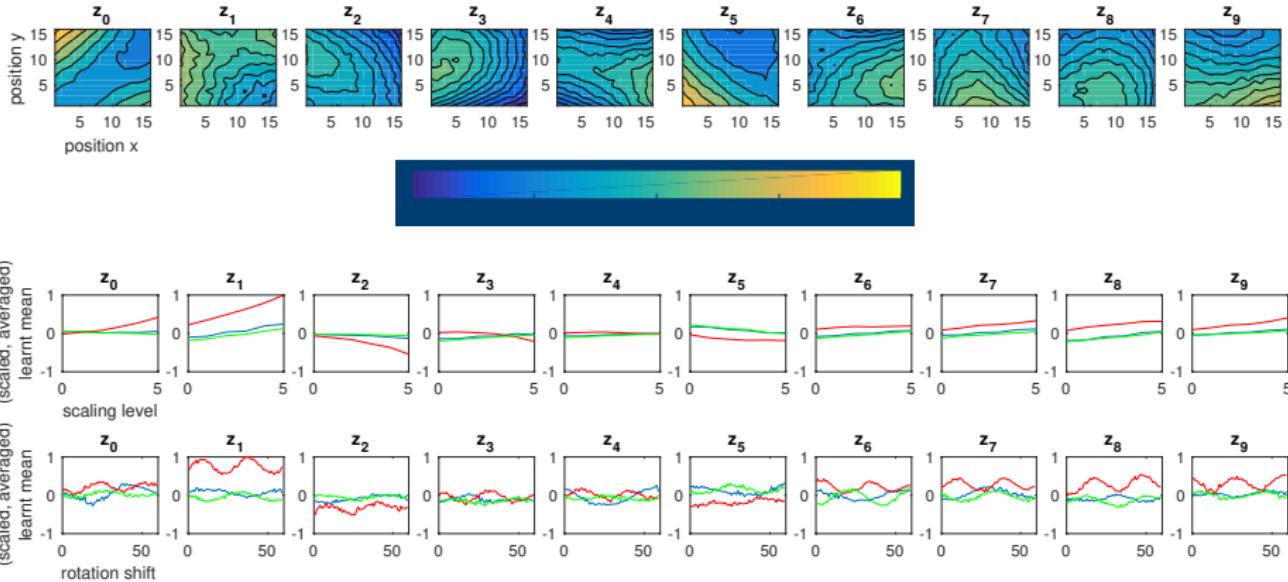
Learnt means of each code element  $z_i$  as a function of the **rotation** factor, averaged across scales and positions.



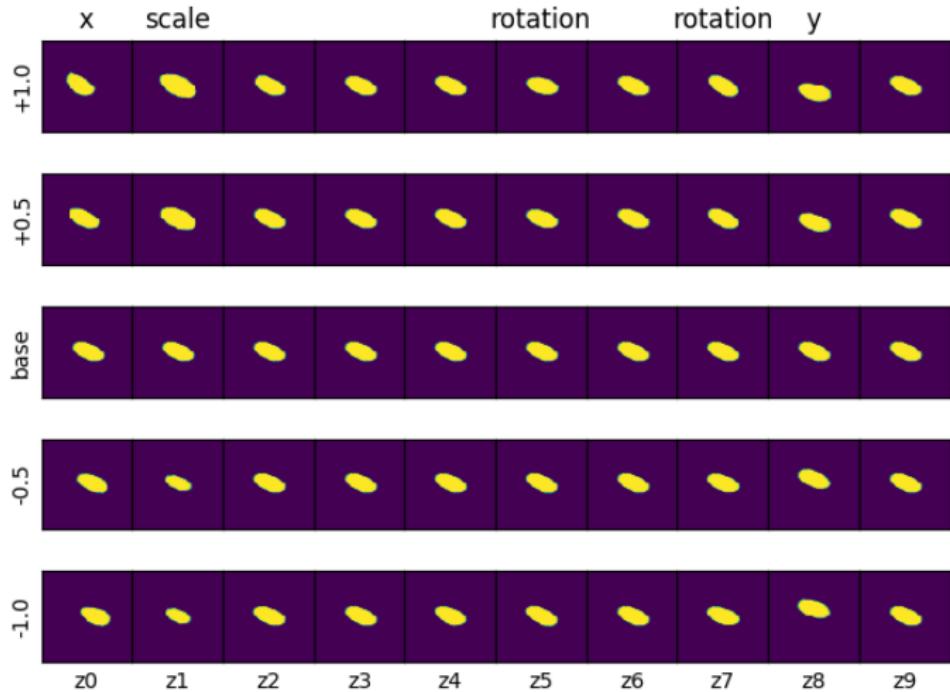
The colours correspond to **shapes** (ellipse, square, triangle).

# Results ( $\beta = 0$ )

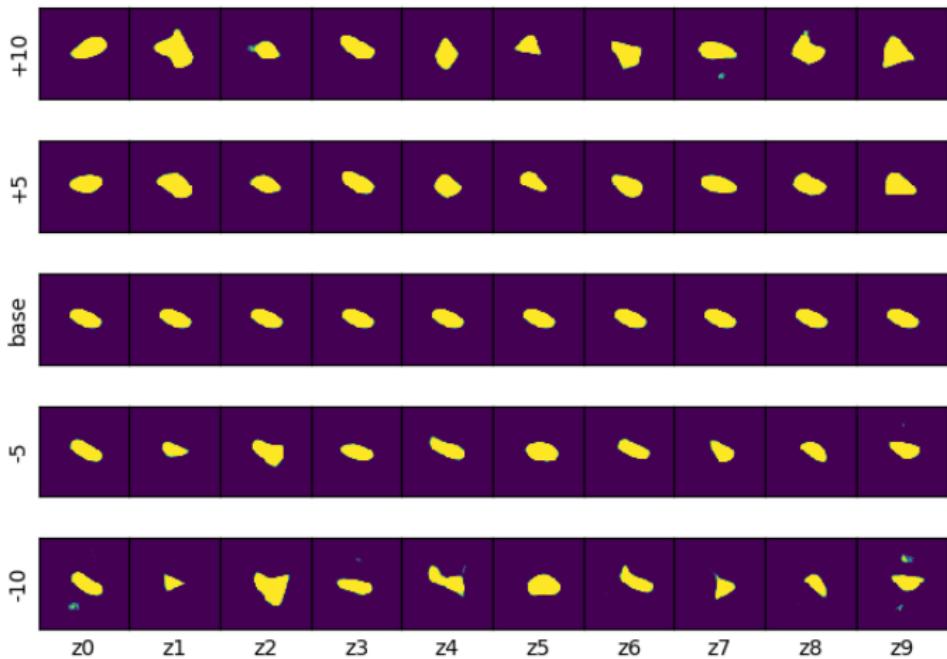
► This autoencoder learns an **entangled** representation.



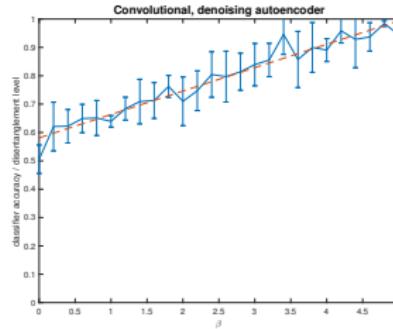
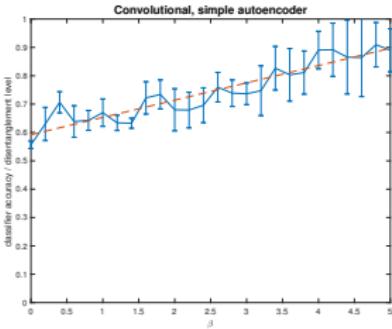
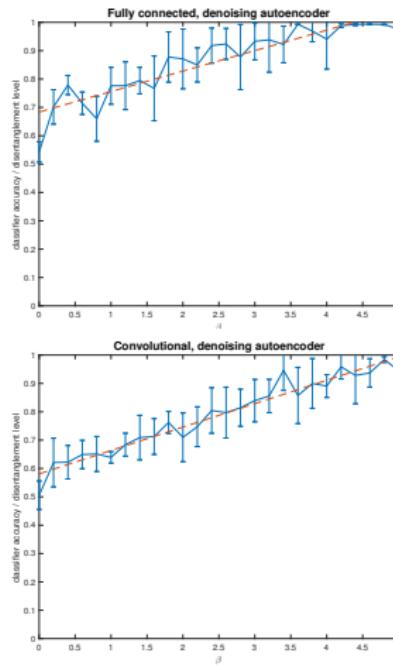
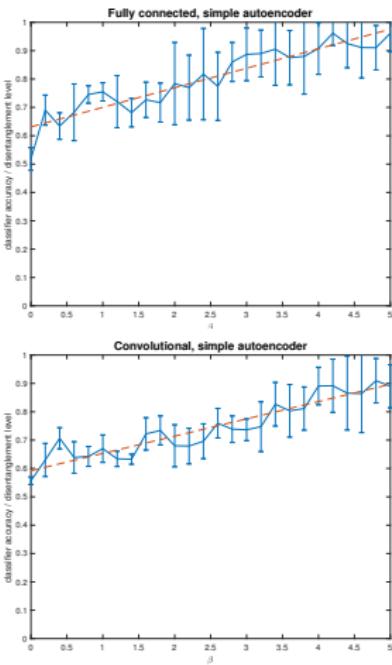
# Shifting the code ( $\beta = 4$ )



# Shifting the code ( $\beta = 0$ )

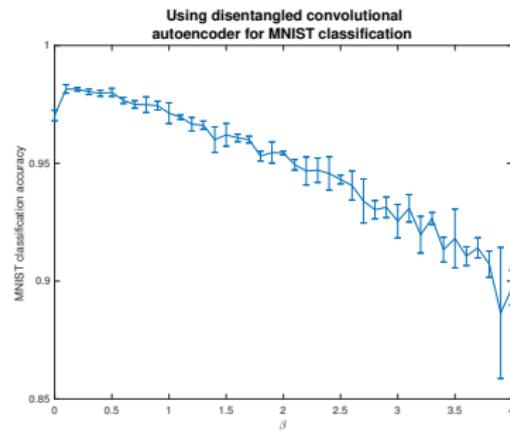
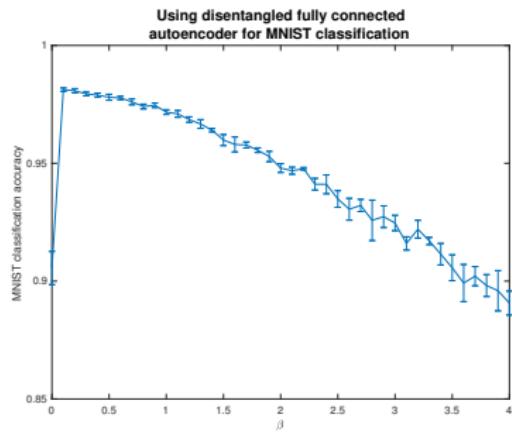


# Disentanglement as a function of $\beta$



# Classification with disentangled VAEs

- ▶ Using the VAE's code to classify on MNIST – does not explicitly satisfy the required assumptions made for the disentangled autoencoder to be successful, but tradeoffs evident.



# Disentangling Atari (*Higgins et al., 2016*)

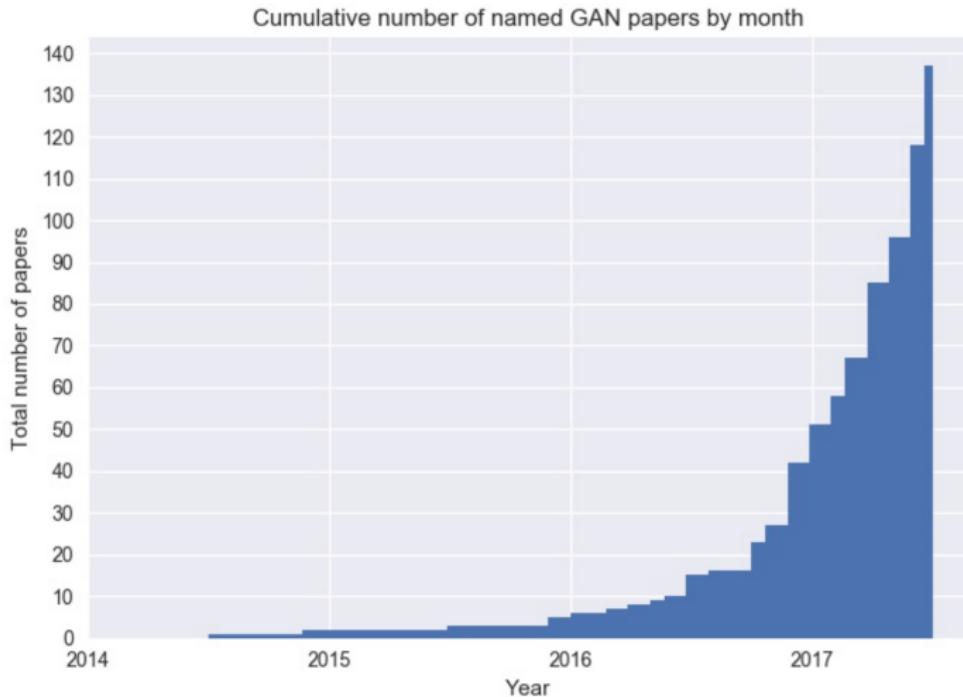


<http://tinyurl.com/jgbyzke>

# Generative Adversarial Networks

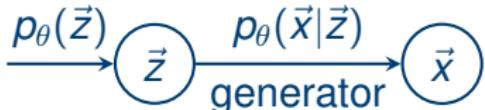
- ▶ We will conclude with a brief overview of a bleeding-edge generative modelling approach—one of the most popular ideas to hit deep learning in the past decade.
- ▶ *“The most important one, in my opinion, is adversarial training (also called GAN for **Generative Adversarial Networks**). This, and the variations that are now being proposed is the most interesting idea in the last 10 years in ML, in my opinion.”*  
—Yann LeCun

# GANs are **everywhere**



# A probabilistic generative framework, revisited

- ▶ Recall the basic generative framework from before...



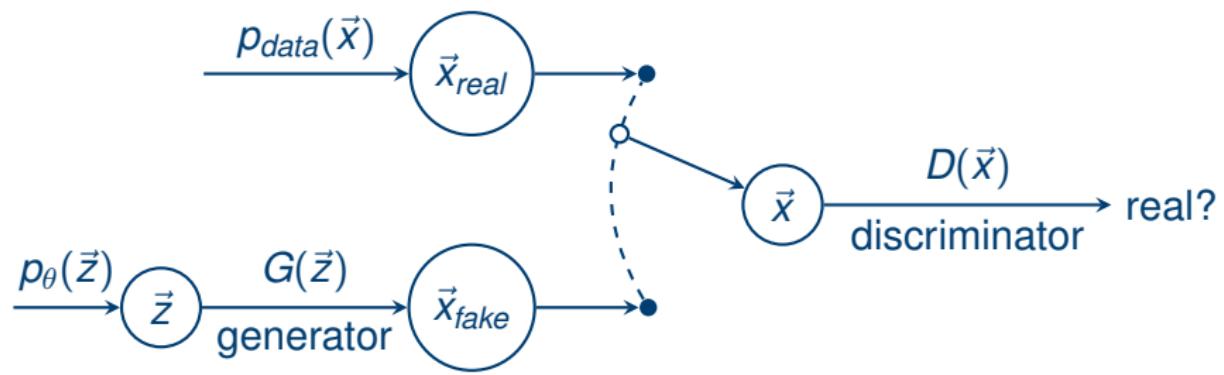
(**N.B.**  $p_{\theta}(\vec{x}|\vec{z})$  is called a **generator**,  $G(\vec{z})$ , this time!)

- ▶ What we **really** want to do is make the distribution of this generator *approach* the **true data distribution**,  $p_{data}(\vec{x})$ .
- ▶ With VAEs, we used the *KL-divergence* as an objective for enforcing one distribution to approach another.
  - ▶ Tractable only if we **know** our target distribution!
  - ▶ But we only have empirical samples from  $p_{data}(\vec{x})$ ...

# Simplifying the distributions

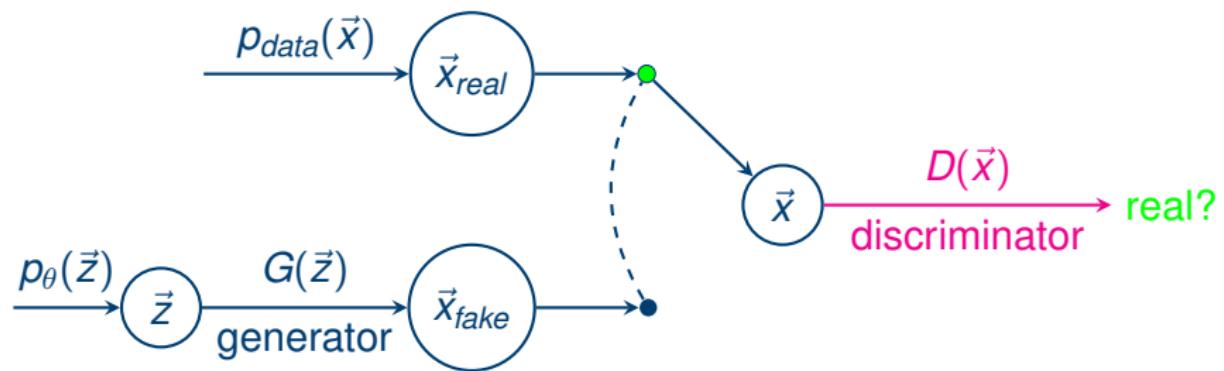
- ▶ Looking over full data distributions (e.g. individual image pixels) is *hard*.
- ▶ We can utilise a *neural network* to extract underlying features from inputs of these two distributions, and examine those.
- ▶ If these features are “good enough”, the network should be capable of *telling the two distributions apart!*
- ▶ Call this network the **discriminator**,  $D(\vec{x})$ .
- ▶ Essentially, a *binary classifier*, telling whether  $\vec{x}$  came from  $p_{data}(\vec{x})$  or  $G(\vec{z})$ .

# The GAN framework



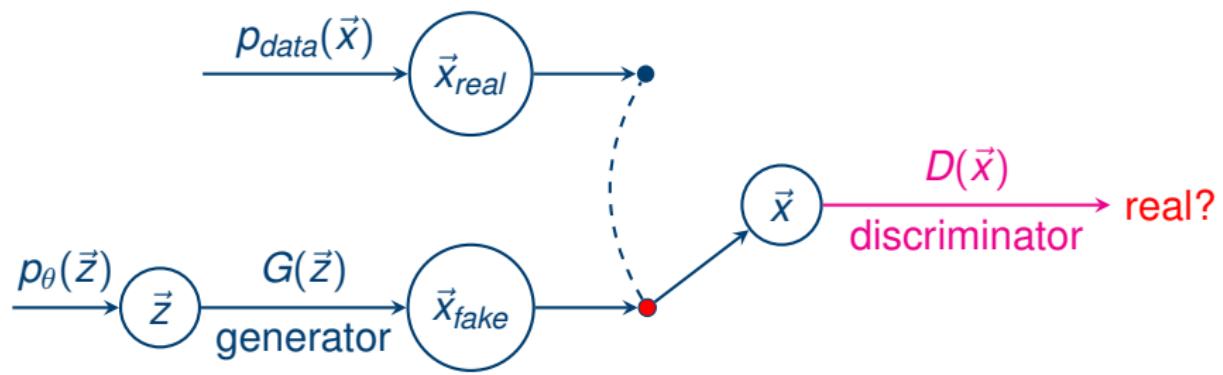
Two neural networks *playing a game...*

# The GAN framework—update step 1



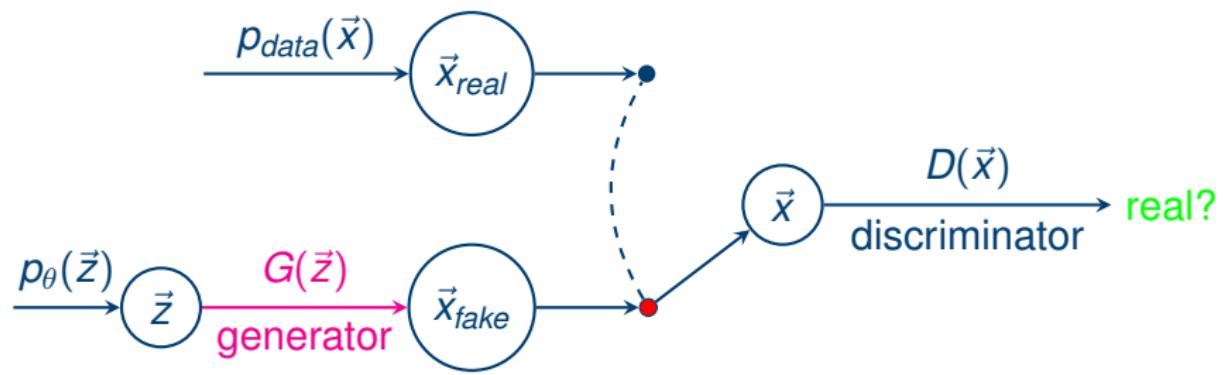
Train discriminator to maximise probability of 'real' on real data.

## The GAN framework—update step 2



Train discriminator to maximise probability of '**fake**' on **fake** data.

## The GAN framework—update step 3



Train generator to maximise probability of '**real**' on **fake** data.

## Two potential distance metrics

- ▶ The Jensen-Shannon (JS) divergence,  $D_{JS}$ :

$$D_{JS}(q(x)\|p(x)) = \frac{1}{2}D_{KL}(q(x)\|m(x)) + \frac{1}{2}(p(x)\|m(x))$$

where  $m(x) = \frac{1}{2}(p(x) + q(x))$

- ▶ Optimised in the original GAN paper (*Goodfellow et al., 2014*), *assuming optimal discriminator*.
- ▶ Therefore, want the discriminator to be *very good*.
- ▶ However, suffers from vanishing gradients when discriminator is *too good*. Tradeoffs?!

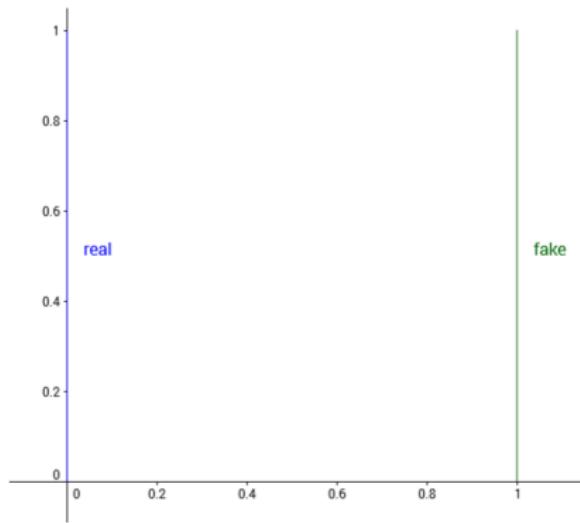
## Two potential distance metrics

- ▶ The Wasserstein/Earth Mover (EM) distance,  $W$ :

$$W(p(x), q(x)) = \inf_{\gamma \in \Pi(p, q)} \mathbb{E}_{(x, y) \sim \gamma} [\|x - y\|]$$

- ▶ Takes into account the underlying geometry of the distributions
- ▶ Indicates the cost of transforming  $p$  into  $q$  under an “optimal transport plan”.
- ▶ Intractable (but various interesting developments)!

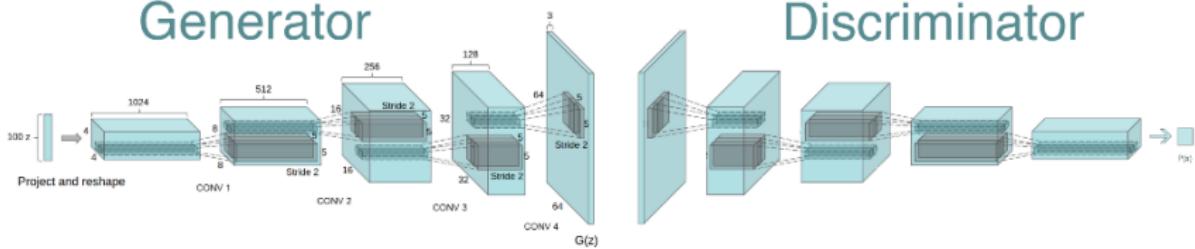
## Aside: Why Wasserstein is desirable



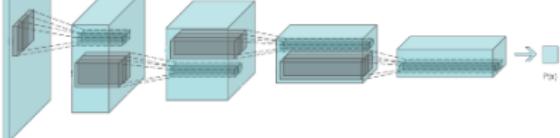
KL-divergence is  $+\infty$ , JS-divergence is a constant ( $\log 2$ ),  
Wasserstein distance is equal to the distance between the lines!

# A deep convolutional GAN (*Radford et al., 2015*)

Generator



Discriminator

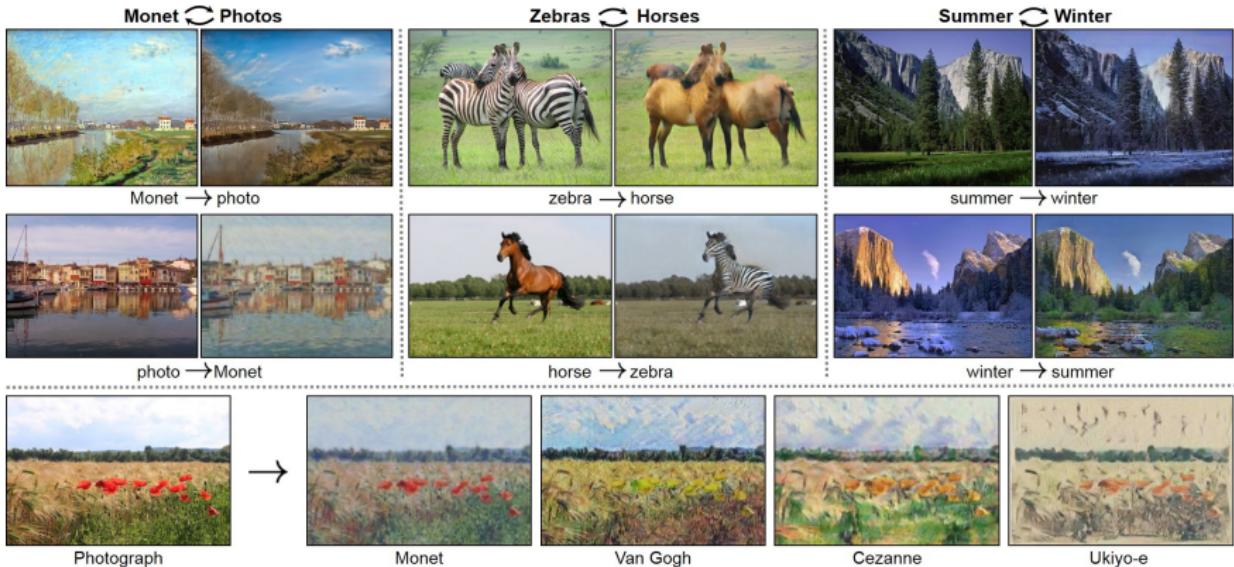


Architectures like this allow us to create some really interesting applications with image distributions...

# Generating fake celebrities (*Hjelm et al., 2017*)



# Domain transfer: *CycleGAN* (*Zhu et al., 2017*)



<https://junyanz.github.io/CycleGAN/>

# An overview of historical deep learning ideas

- ▶ Initially, we needed to extract hand-crafted features before applying a machine learning model to them.
  - ▶ **Deep neural networks** can perform feature extractions by themselves.
- ▶ Then, we needed to select a hand-crafted loss function to optimise.
  - ▶ **GANs** use a neural network (the *discriminator*) to compute a customised loss!
- ▶ We need to figure out a correct way to perform the optimisation of the loss function.
  - ▶ **Learn how to learn?**

Thank you!

# Questions?

`petar.velickovic@cst.cam.ac.uk`

## Special thanks:

Momchil Peychev (*University of Cambridge*)

Devon Hjelm (*Montréal Institute for Learning Algorithms*)

## Reading material:

‘Deep Learning’, Chapter 13 (PCA)

‘Deep Learning’, Chapter 14 (AEs, DAEs)

‘Deep Learning’, Chapter 20 (RBMs, DBNs, VAEs, GANs)