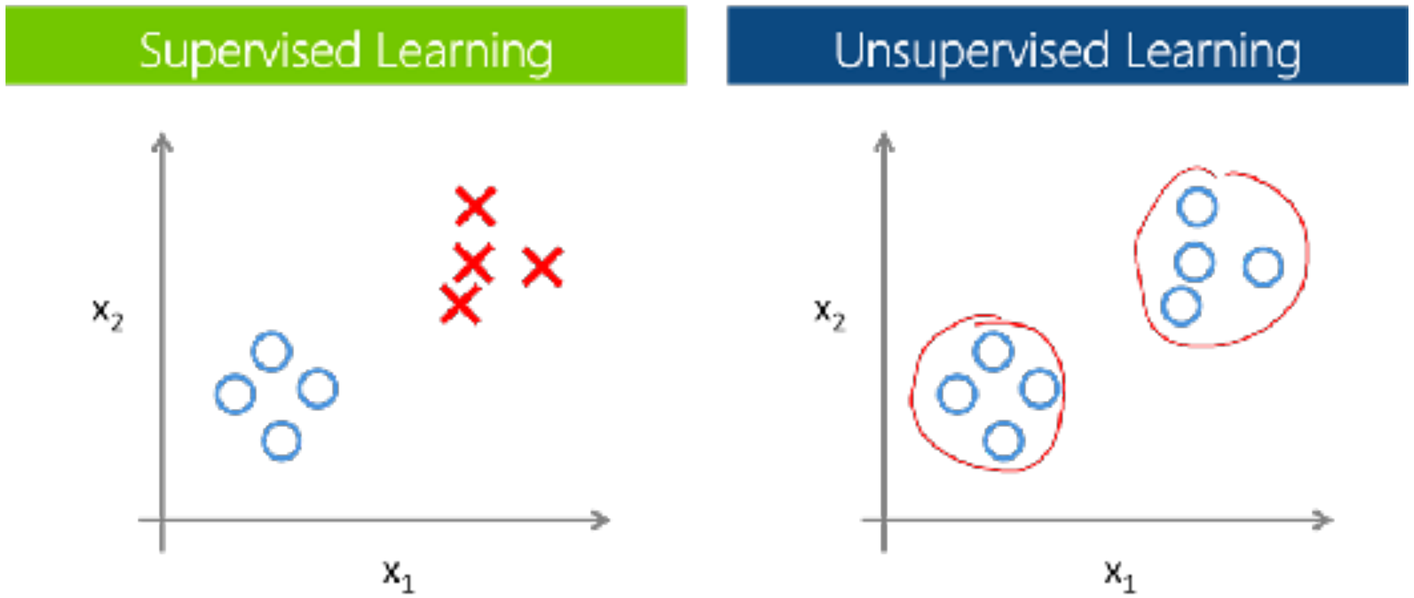
Deep Learning: Unsupervised Learning

Adam Moss
School of Physics and Astronomy
adam.moss@nottingham.ac.uk

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

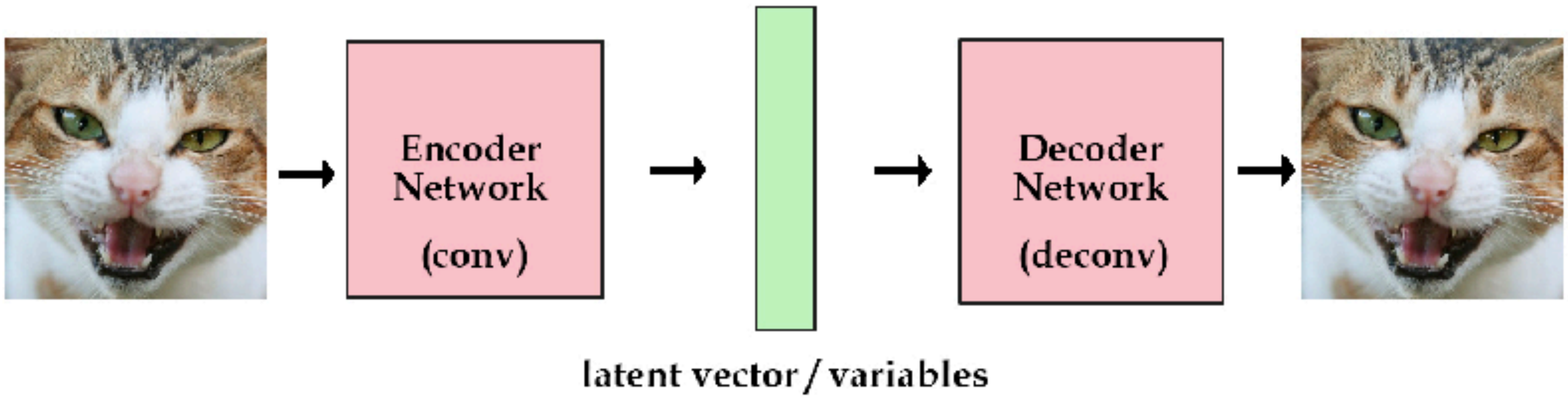
Unsupervised Learning

- **Much** more unlabelled than labelled data
- Labelling can be subjective
- Unsupervised learning finds the most important features in the dataset



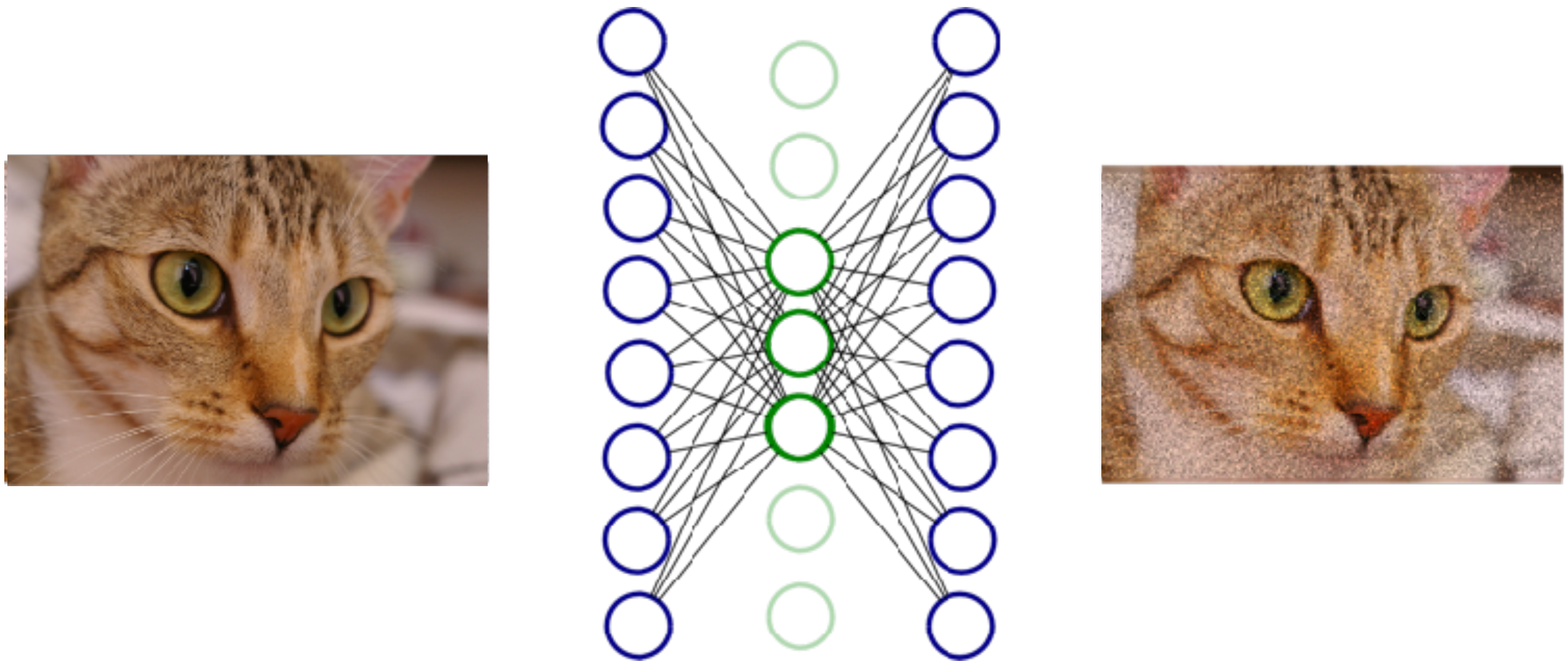
- Examples of unsupervised learning
  - Auto-encoders and variational auto-encoders (generative)
  - Adversarial networks (generative)
  - Boltzmann machines (generative)
  - Reinforcement learning (self-supervised)

Auto-encoders



- Data compression algorithm
  - Data specific (will only be able to compress data similar to what they are trained on)
  - Lossy (decompressed output degraded compared to input)
  - No labelling required

Auto-encoders



- Latent space should have fewer dimensions
  - Under-complete representation
  - Bottleneck architecture
- Otherwise over-complete representation might just learn the identity function

MNIST

- Used Keras Python library
- Discard labels
- Normalised values between 0 and 1 and reshape into 784 vector as before
- Encoding dimension 32, corresponds to compression factor of 24.5
- Can visualise reconstructed inputs from the test set

```python
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print x_train.shape
print x_test.shape

encoding_dim = 32

input_img = Input(shape=(784,))
encoded = Dense(encoding_dim, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)

autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))

decoded_imgs = autoencoder.predict(x_test)
```

- Losing detail with this approach - how do we improve it?

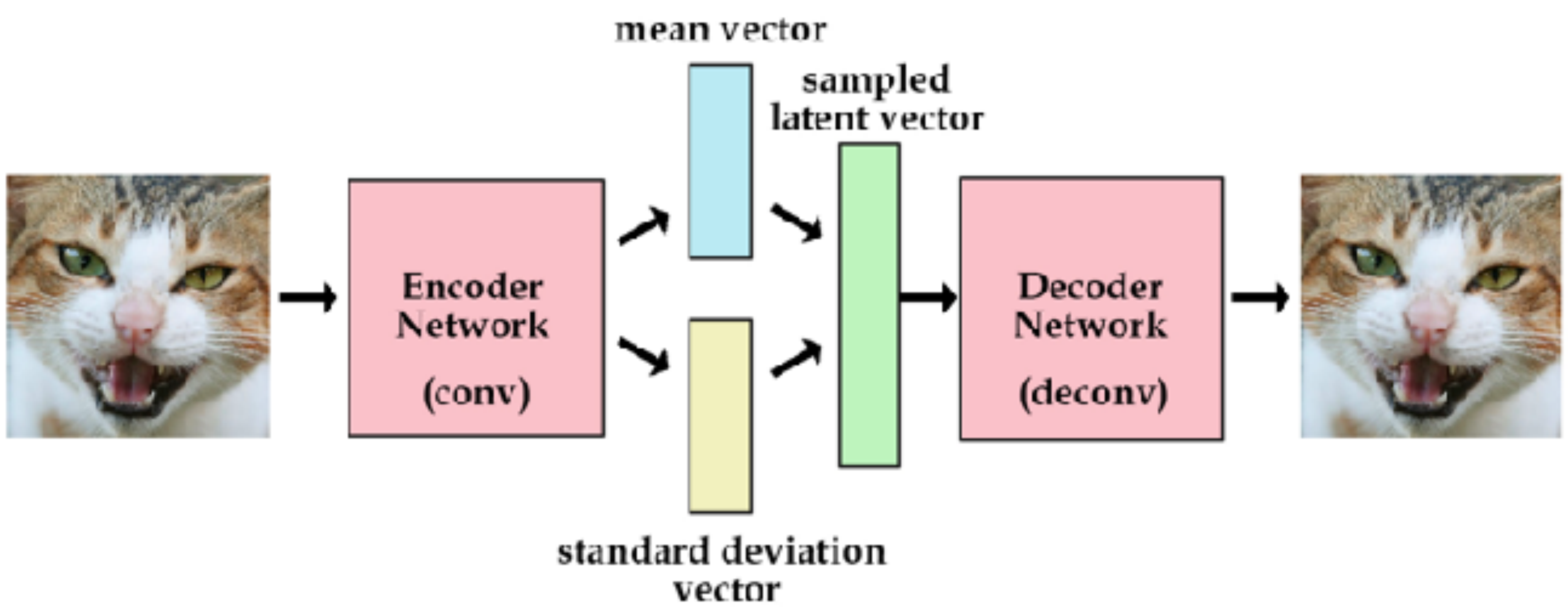
CNN Auto-encoder

- Since inputs and outputs are images it makes sense to use convolutional network as encoder and decoder
- We also do not have to limit ourselves to a single latent layer
- Both encoder and decoder are stacked CNNS
- Loss function significantly improved (0.094 vs 0.11)

```python
from keras.layers import Input, Dense
from keras.models import Model
from keras.layers import Conv2D, MaxPooling2D, UpSampling2D
from keras import backend as K
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

input_img = Input(shape=(28, 28, 1))

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, x_test))
```

Variational AE

- An auto-encoder represents data in latent space
- Can't generate anything yet, since we do not know how to create latent vectors
- Variational AE adds constraints on latent space, i.e. it learns a latent model such as a Gaussian probability density
- If you then sample points from this probability density, you can generate new samples



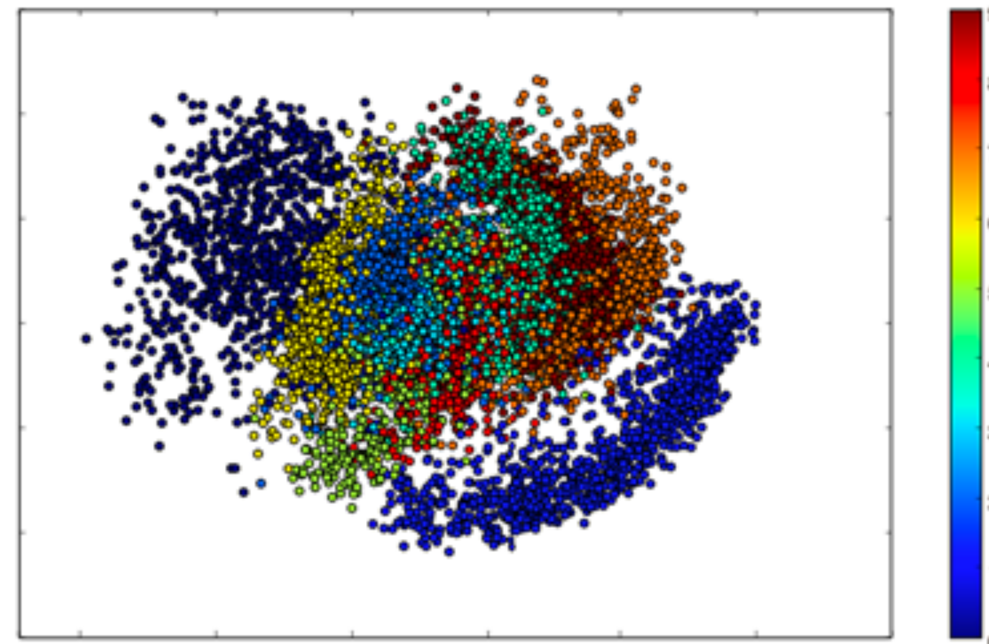
Variational AE

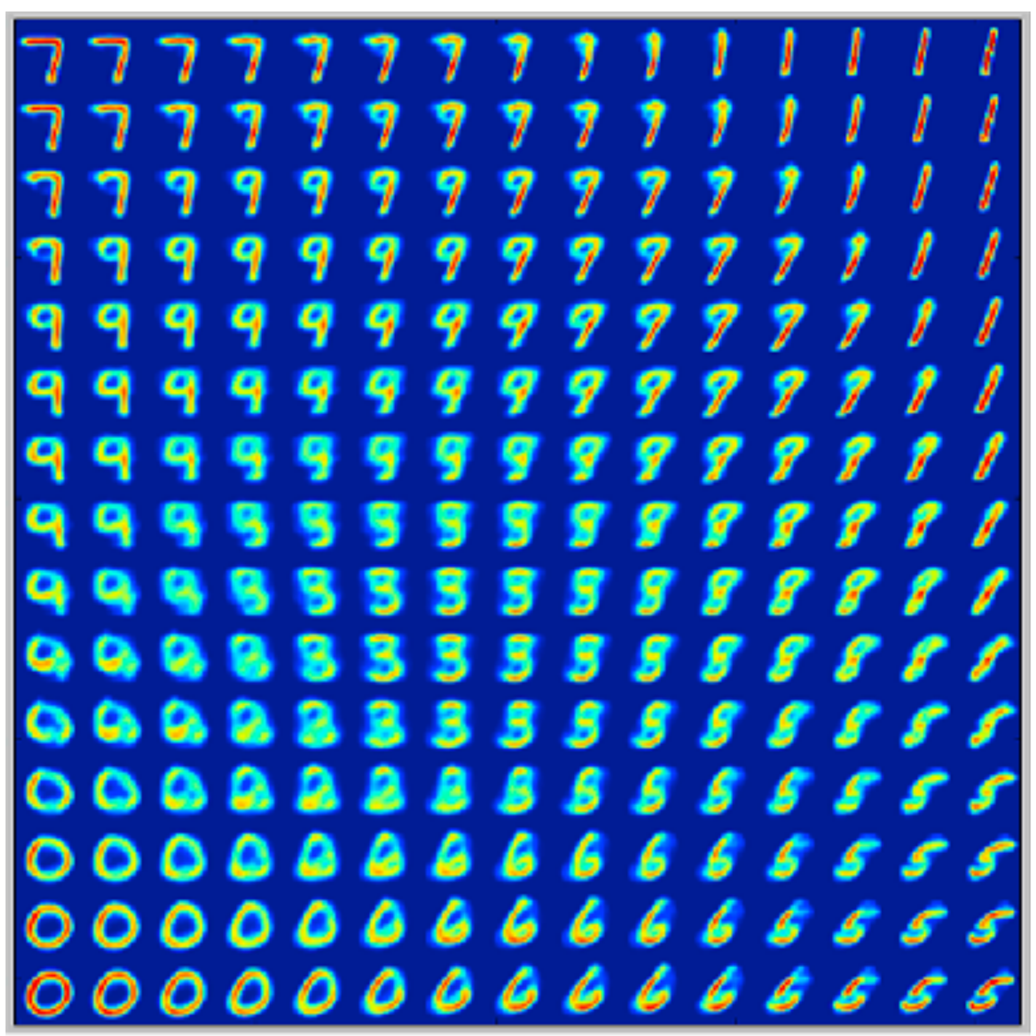
- First encoder turns inputs into 2 parameters in latent space - z_mean and z_var_log
- Randomly sample from latent space which is assumed to generate data
- Decoder maps these back to original inputs
- Loss function has 2 components - usual reconstruction loss and Kullback-Leibler divergence between learnt latent distribution and latent model

```python
batch_size = 100
original_dim = 784
latent_dim = 2
intermediate_dim = 256
epochs = 50
epsilon_std = 1.0


x = Input(shape=(original_dim,))
h = Dense(intermediate_dim, activation='relu')(x)
z_mean = Dense(latent_dim)(h)
z_log_var = Dense(latent_dim)(h)


def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim), mean=0.,
                              stddev=epsilon_std)
    return z_mean + K.exp(z_log_var / 2) * epsilon

# note that "output_shape" isn't necessary with the TensorFlow backend
z = Lambda(sampling, output_shape=(latent_dim,))([z_mean, z_log_var])

# we instantiate these layers separately so as to reuse them later
decoder_h = Dense(intermediate_dim, activation='relu')
decoder_mean = Dense(original_dim, activation='sigmoid')
h_decoded = decoder_h(z)
x_decoded_mean = decoder_mean(h_decoded)


# Custom loss layer
class CustomVariationalLayer(Layer):
    def __init__(self, **kwargs):
        self.is_placeholder = True
        super(CustomVariationalLayer, self).__init__(**kwargs)

    def vae_loss(self, x, x_decoded_mean):
        xent_loss = original_dim * metrics.binary_crossentropy(x, x_decoded_mean)
        kl_loss = - 0.5 * K.sum(1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
        return K.mean(xent_loss + kl_loss)

    def call(self, inputs):
        x = inputs[0]
        x_decoded_mean = inputs[1]
        loss = self.vae_loss(x, x_decoded_mean)
        self.add_loss(loss, inputs=inputs)
        # We won't actually use the output.
        return x

y = CustomVariationalLayer()([x, x_decoded_mean])
vae = Model(x, y)
vae.compile(optimizer='rmsprop', loss=None)
```

Variational AE

- Since latent space is 2D can visualise it (each coloured cluster is digit)





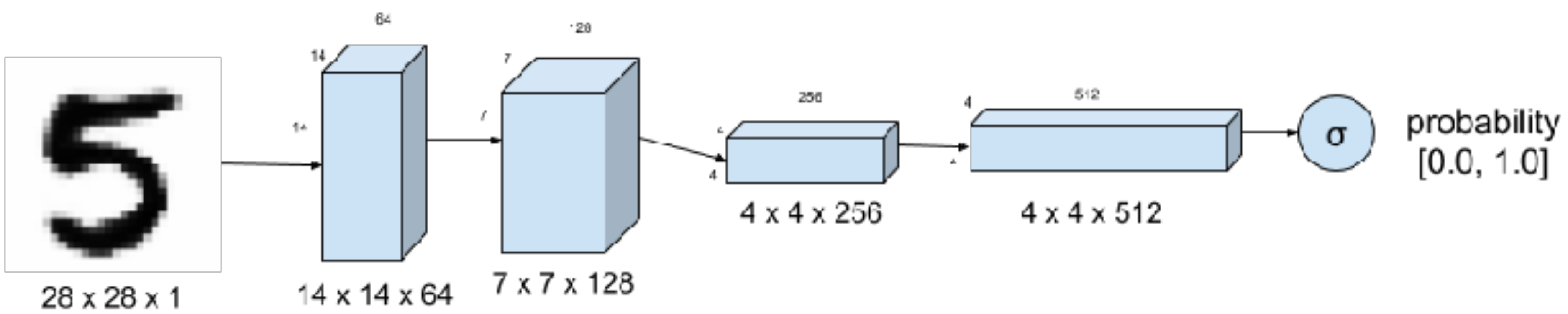
- Generated digits

Adversarial Networks

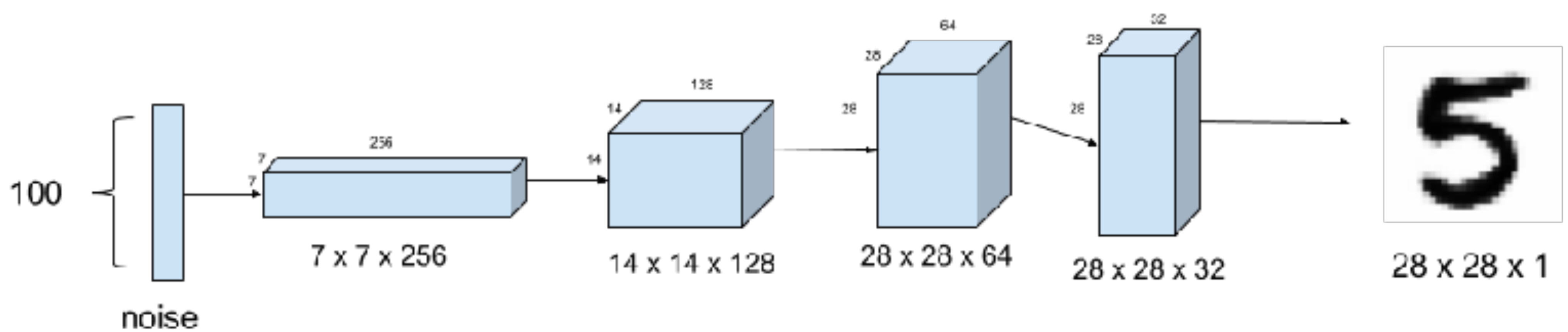
- VAEs are easy to train and have a tractable likelihood, but can produce images with lower fidelity
- GANs (Generative Adversarial Network) produce richer images but are much more difficult to train
- A GAN consists of 2 networks - a generator and discriminator, that compete and co-operate
- Real word example - counterfeiter (generator) and police (discriminator)
- Counterfeiter produces fakes, police give feedback why money is fake
- Counterfeiter improves fakes to trick police based on feedback
- At same time police are improving how to better tell apart real and fake money

GAN

- Discriminator tells if something is generative or real, e.g money, MNIST digits. Example is CNN



- Generator synthesises fakes. Input is generated from noise

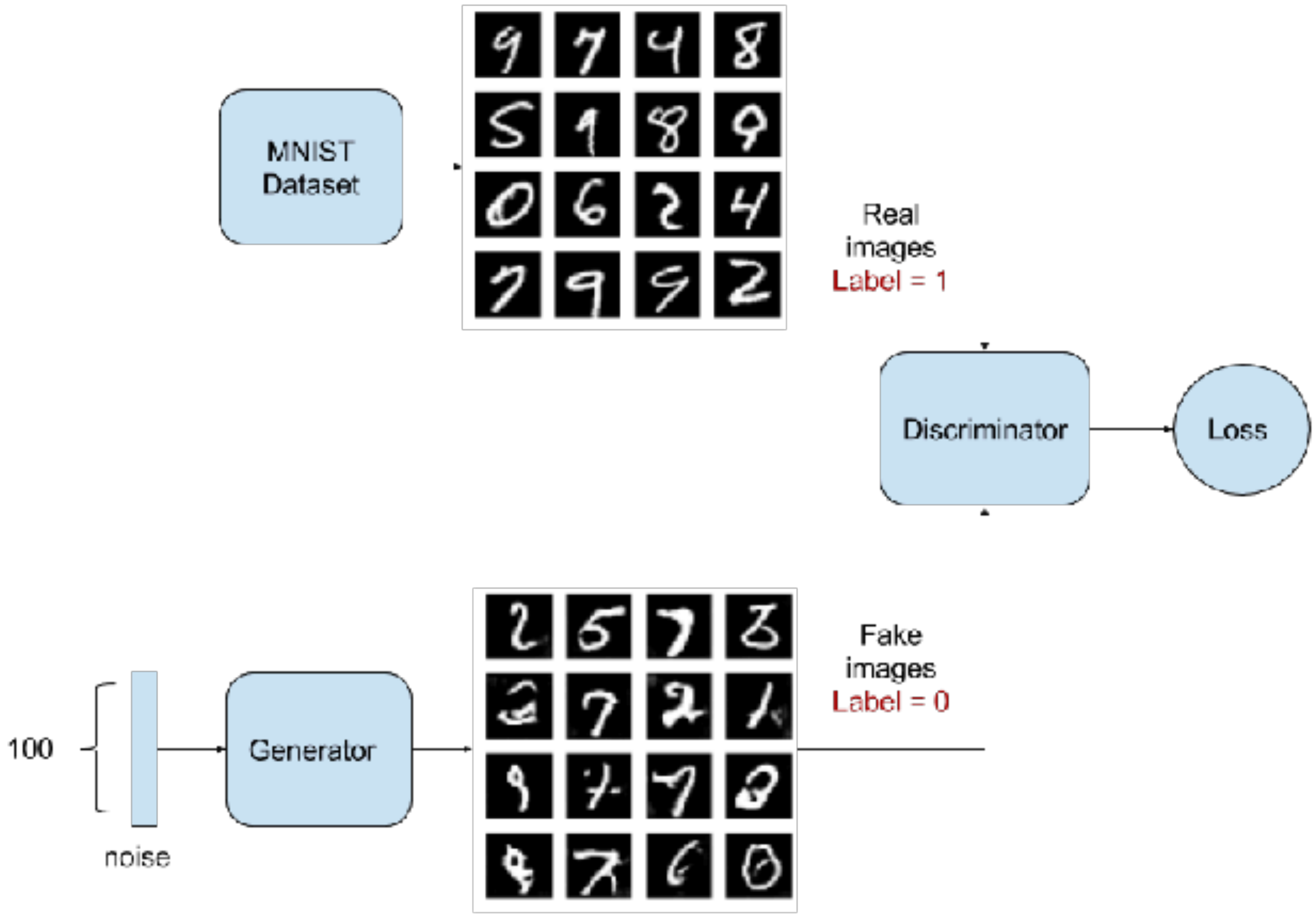


GAN

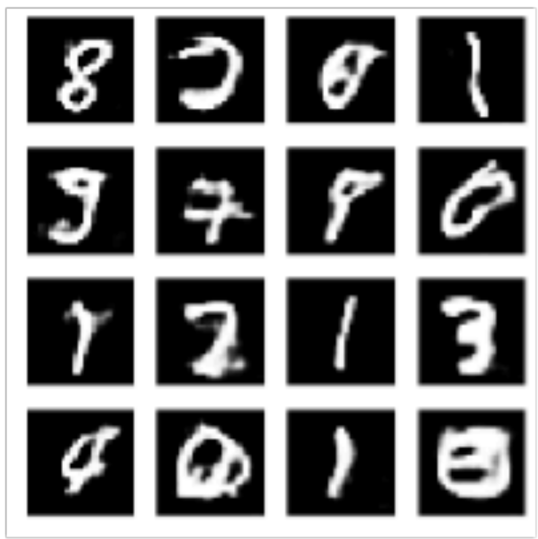
- Adversarial model is generator + discriminator stacked together with generator trying to fool the discriminator (images are labelled as real when in fact they are fakes)

GAN

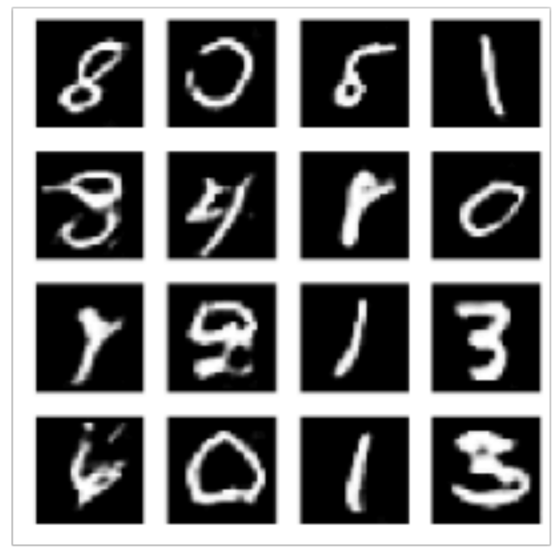
- Full discriminator model is generator + discriminator with both real and fake data correctly labelled



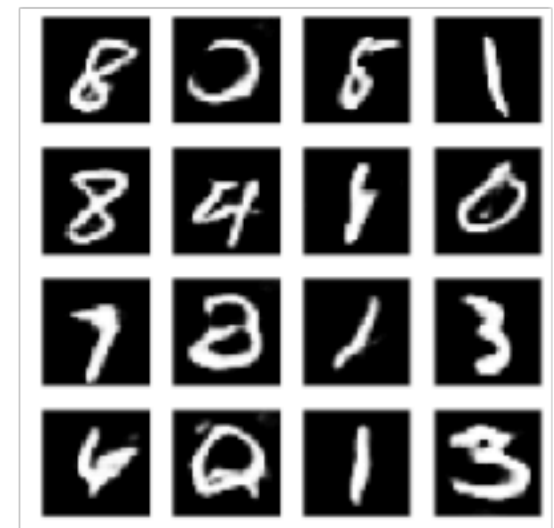
Samples



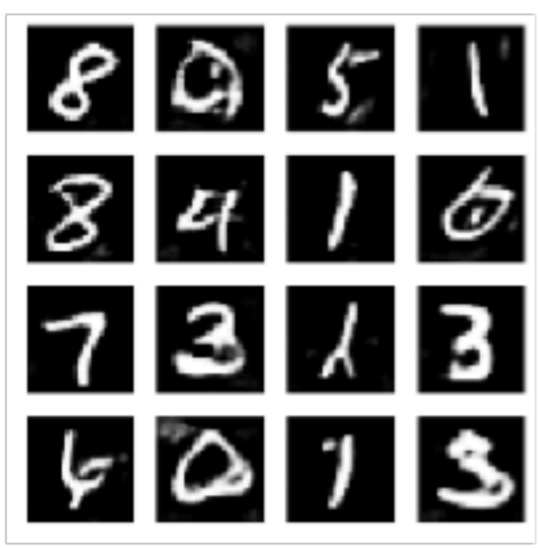
1000 steps



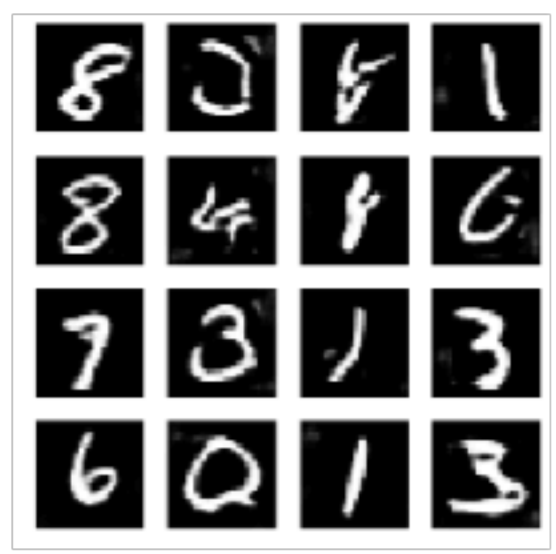
3000 steps



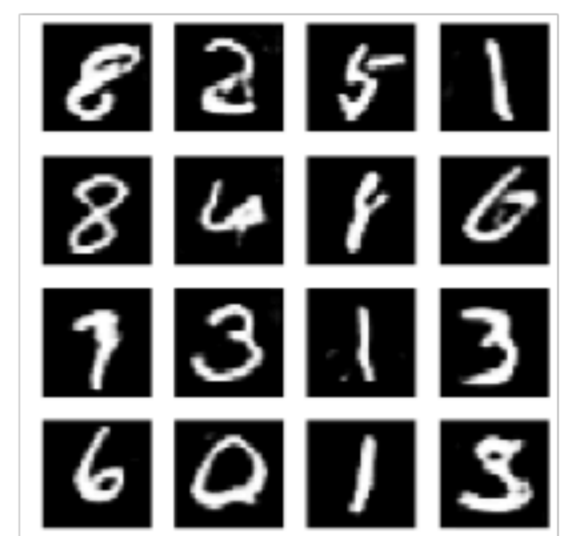
5000 steps



7000 steps



9000 steps



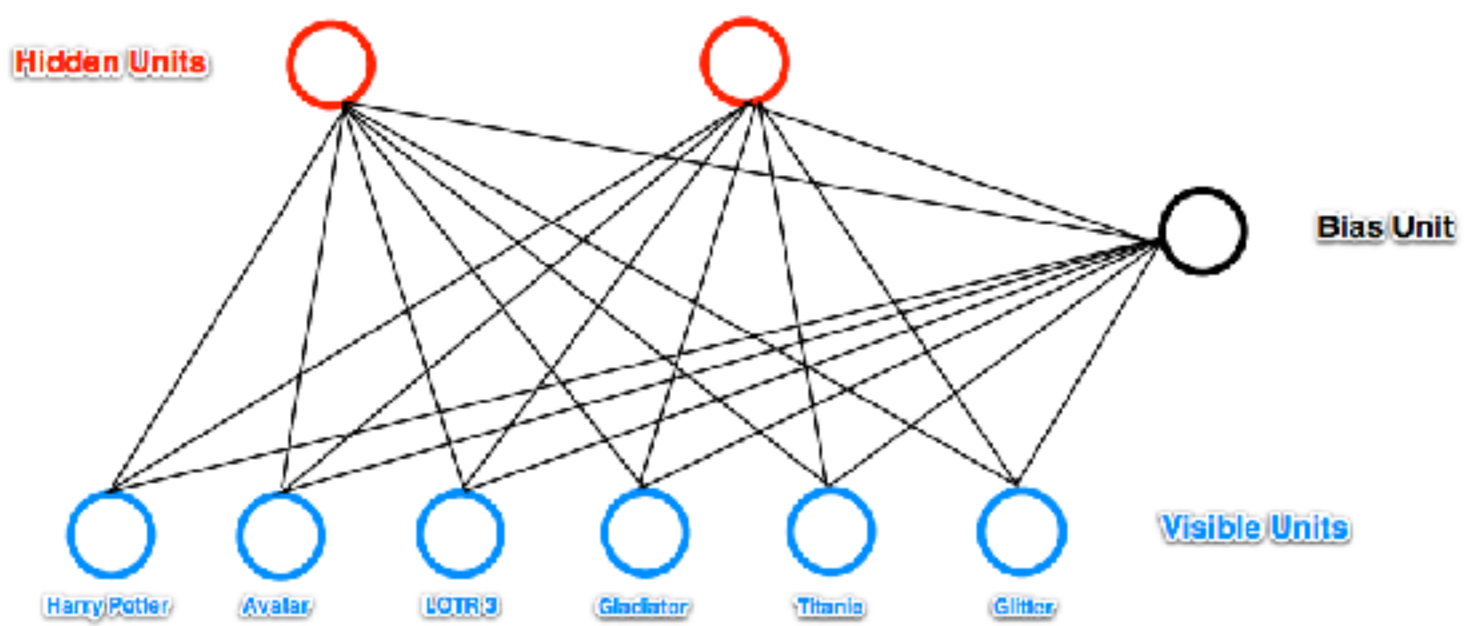
10000 steps

Other Examples

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

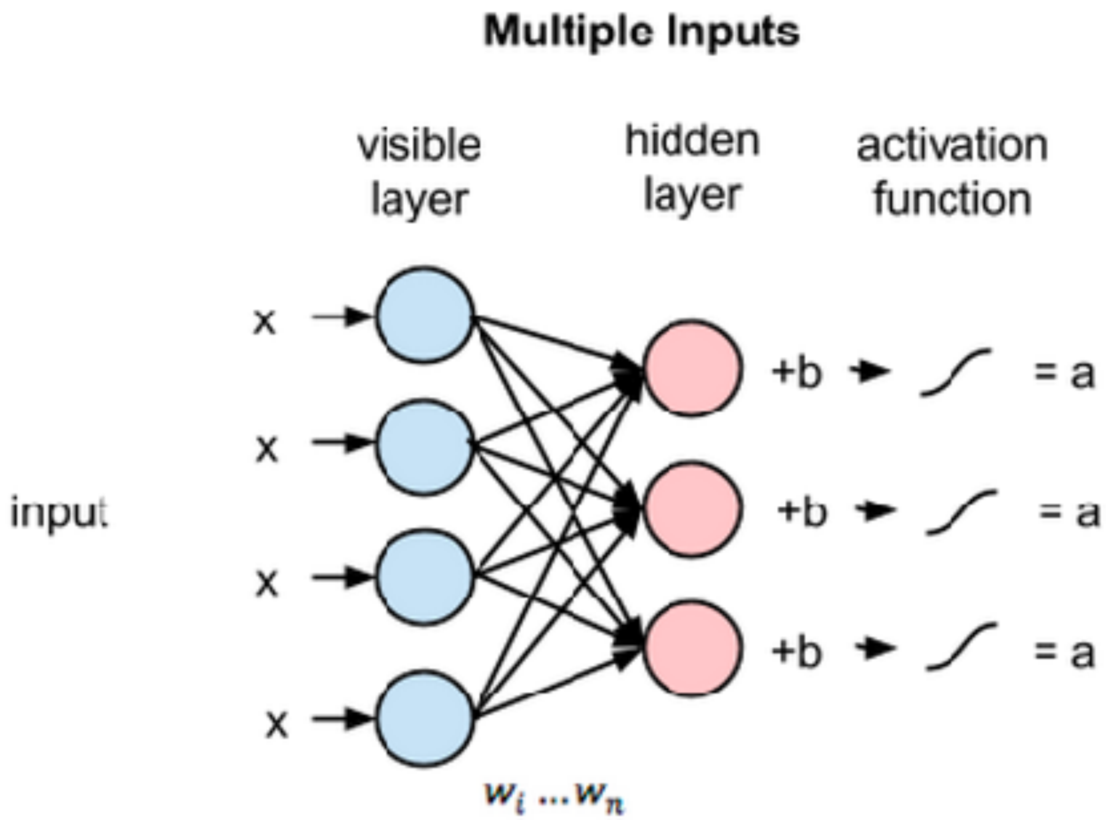
Boltzmann Machines

- Another type of generative network
- Important difference is that it is **stochastic**, i.e. activations in network have a probabilistic element
- Restricted Boltzmann Machine (RBM) consists of one visible layer and one hidden layer (contained latent factors)
- E.g. we have 6 movies and ask people which ones they want to watch. We will learn 2 latent variables



Boltzmann Machines

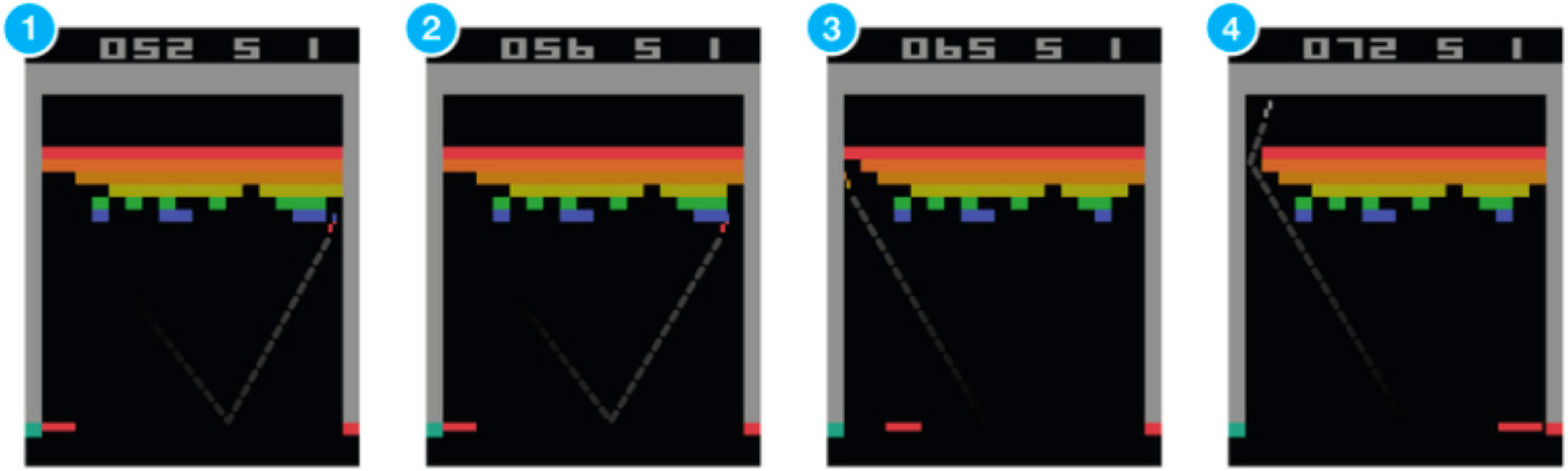
- In forward pass compute weighted activation a
- Let $p_i = \sigma(a_i)$ be probability that we turn on unit i
- In update of weights during training this means that it doesn't guarantee a hidden unit will be turned on
- In backwards pass data is generated from hidden units, but will not always get the same data



Reinforcement Learning

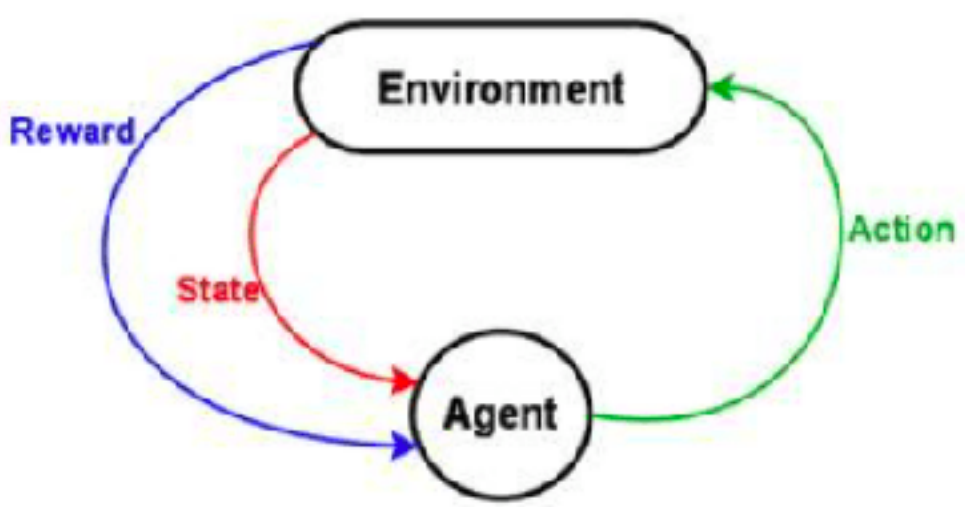
The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

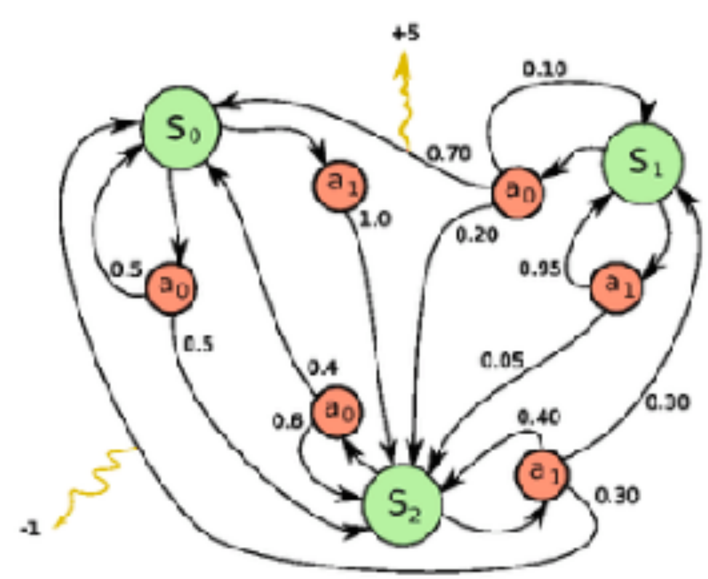
- Self-supervised learning, reward based feedback
- E.g. breakout game. Goal is to get ball though the blocks
- Can teach neural network how to play by giving the image of the screen as an input to the network
- Network will then decide which action to take (move paddle left or right) to maximise future possible reward



Reinforcement Learning

- In general have an agent in an environment
- Environment is in certain state
- Agent can perform actions. These sometimes result in a reward
- Actions transform environment and lead to new state
- Sequence of states, actions, rewards $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$
- A full sequence before terminal state is called an episode
- Rules for how to choose actions is called a policy





Q Learning

- Total discounted future reward in episode from time t onwards

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

- Define Q function as maximum possible discounted future reward when we perform an action a in state s at time t

$$Q(s_t, a_t) = \max R_t$$

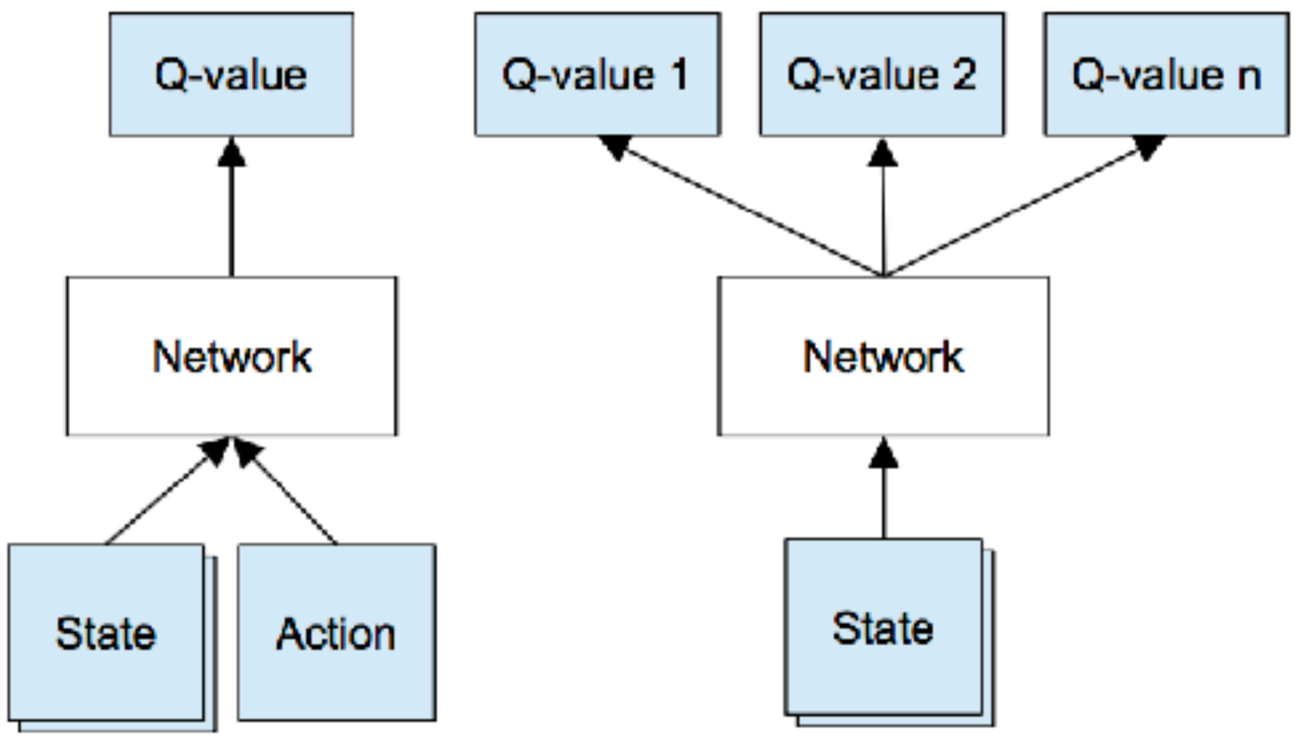
- Can be iteratively estimated using the **Bellman equation**

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

- No neural network yet - could be done with a table of states vs actions

Deep Q Network

- Can train a neural network to estimate the Q function by playing out many episodes of a game



- Several subtleties - e.g. training using experience replay