Machine Learning Applications

Lecture 7: Generative Neural Networks

Lecture Notes

M.Sc. in Financial Markets

ACADEMIC YEAR 2021-2022

Dr. Dominic O'Kane

Version: January 2022

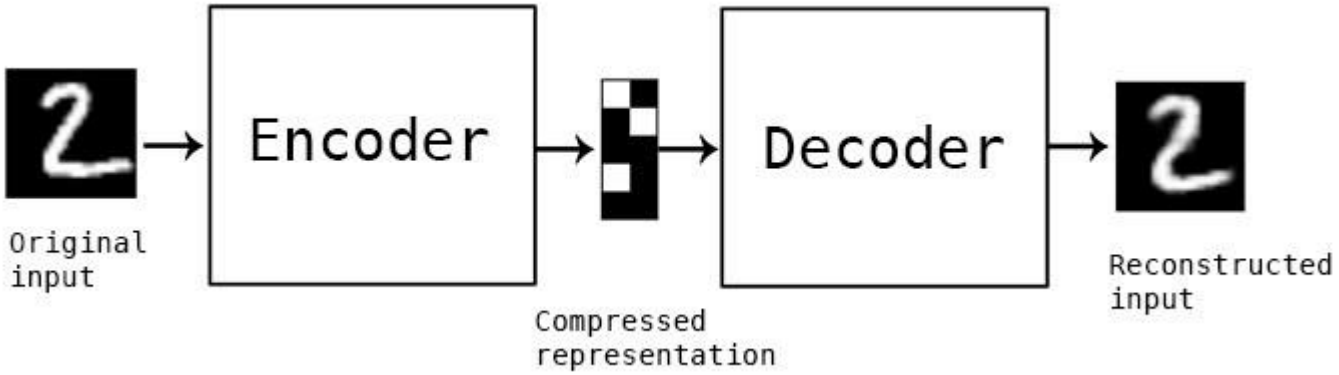
Introduction

- Generative models generate new data
- This is one of the newest areas of machine learning
- They have been used to generate Deep Fakes – fake videos with the faces of famous actors
- In finance they can be used to generate realistic data when there is insufficient actual data
- This can be time series data, transaction data, etc…
- There are three main sets of models
  - Autoencoders
  - Variational Autoencoders
  - GANs – General Adversarial Networks
- We will consider Autoencoders and GANs

Autoencoders

Autoencoders – Semi-supervised Learning

- An autoencoder is a semi-supervised neural network
- Its purpose is to output its input data
- But in such a way that it compresses the input data
- It's a little bit like principal component analysis
- In the context of the MNIST data set, it is about finding a low-level representation of each handwritten number



- But it can be done with any data, including financial data

Autoencoder Uses

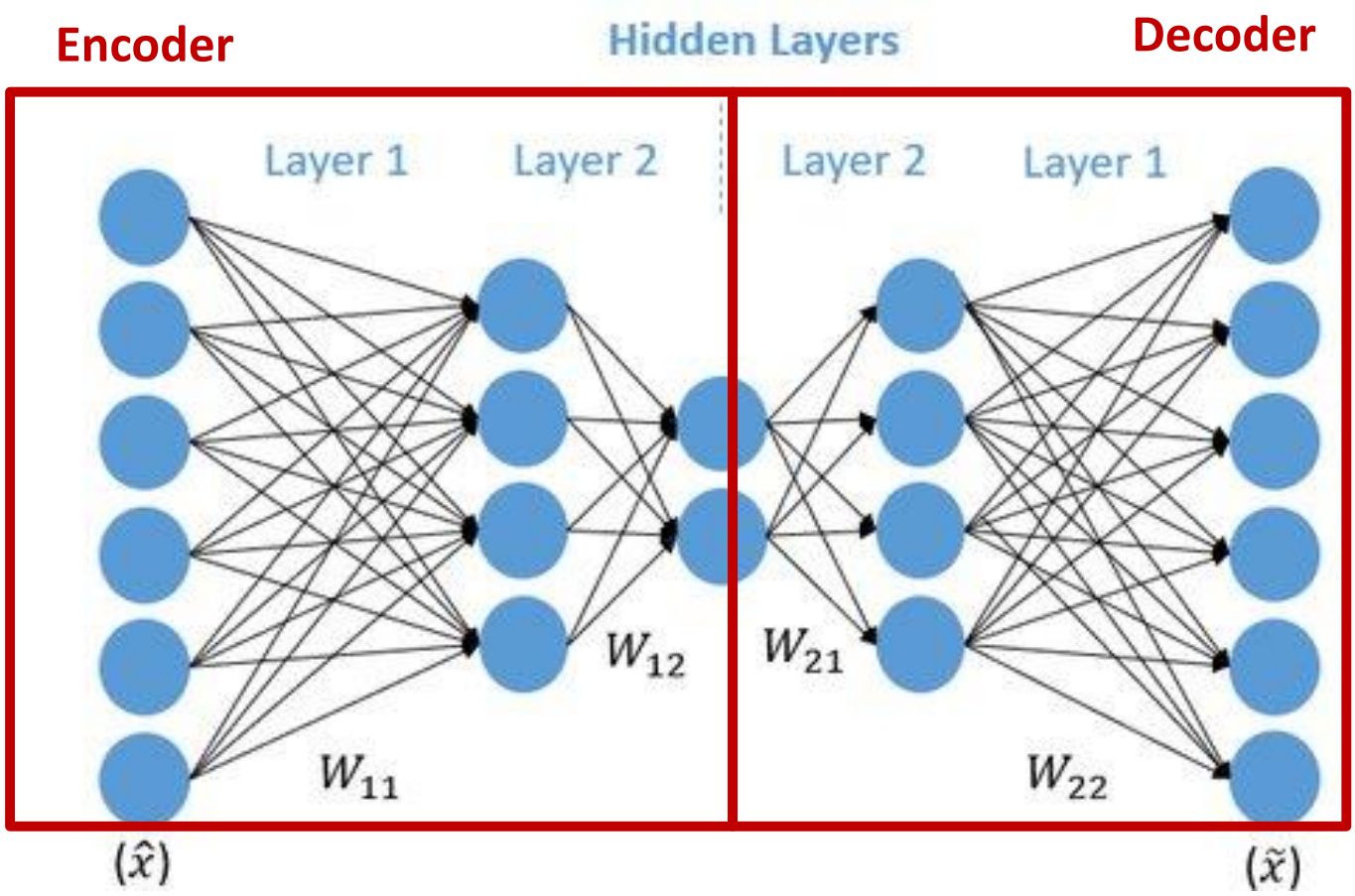
- Main purpose is "dimensionality reduction"
  - Compress data
  - Remove noise
  - Visualise data in lower dimensions
  - Reveal hidden relationships not seen clearly in higher dimensions
- How does it do this ?

The Big Idea of an Autoencoder

- The Autoencoder is a simply a multilayer perceptron
- We require the network to output the input
- **The idea is that we do this after pushing the inputs through a layer with a small number of neurons !**
- If this was not the case, the task would be trivial, and no value would be added by the GAN
- By reducing the number of neurons, we make the NN learn to represent the information more efficiently
- The output is not a simple copy of the input but a **reconstruction**

Architecture is a Symmetric Multilayer Perceptron

- We have a series of Dense layers – an **encoder** + **decoder**



- A vector **x** goes in and we want to get the same vector out

The Autoencoder Architecture

The Autoencoder is like a simple multilayer NN with 4 components:

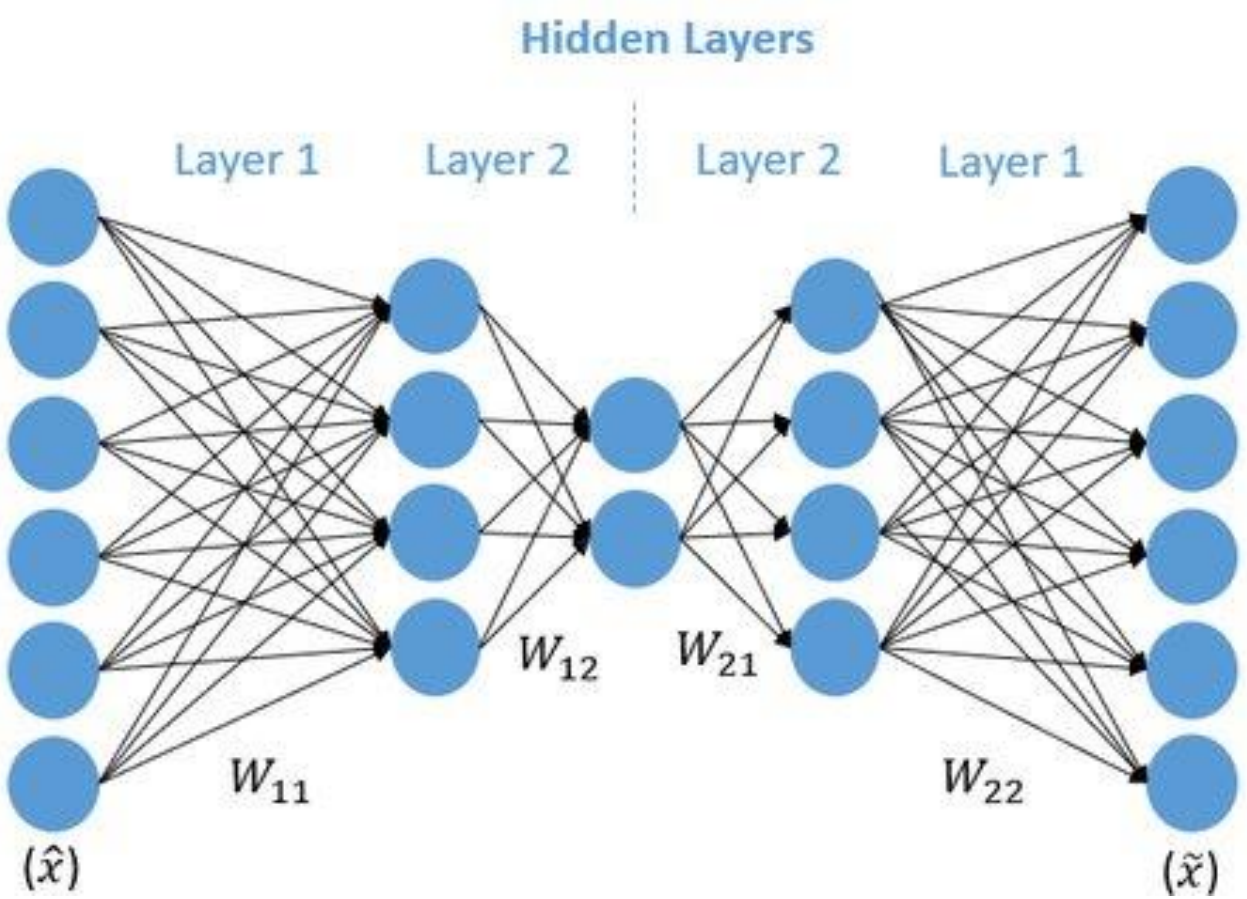
1. **Encoder** – this is where the model reduces the dimensionality of the representation – in an ANN setting this means reducing the number of neurons in each subsequent layer
2. **Bottleneck** –the part of the model where the representation is lowest – this is the layer with the lowest number of neurons
3. **Decoder** – this is where the model seeks to reconstruct the data from the lowest representation – it usually has the same shape as the input layer
4. **Reconstruction Loss** – this is the method of measuring that the decoder is reconstructing the initial input

What is happening in an Auto-Encoder

- The ANN needs to find a way to store the representations
- It reduces the dimensionality of the input representation
- It does this by teaching the **factors** to the weights which learn about the nature of the input data
- The centre data encodes the data in terms of these factors
- This is what happens when we train it to output the input
- If the activations were linear, this would be PCA
- In a non-linear world, we have a different and sometimes better representation
- The model is not throwing away features but is combining them in a way that detects common relationships

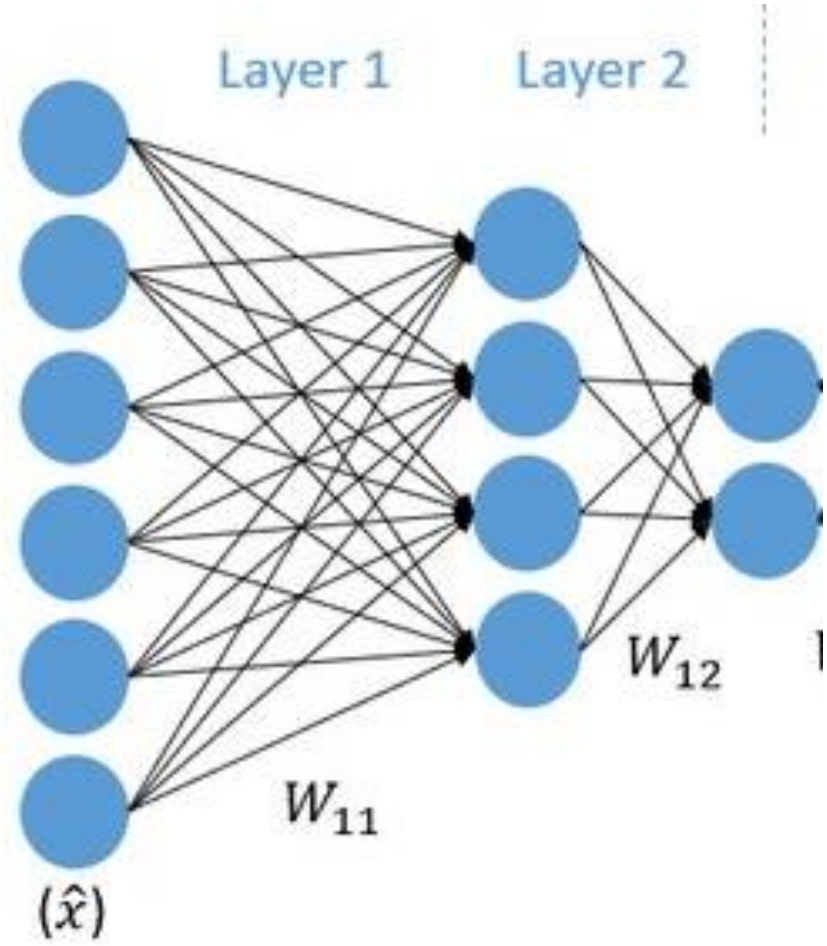
Step 1: We Train the Auto-Encoder

- We use backpropagation to train the layers to match inputs and outputs



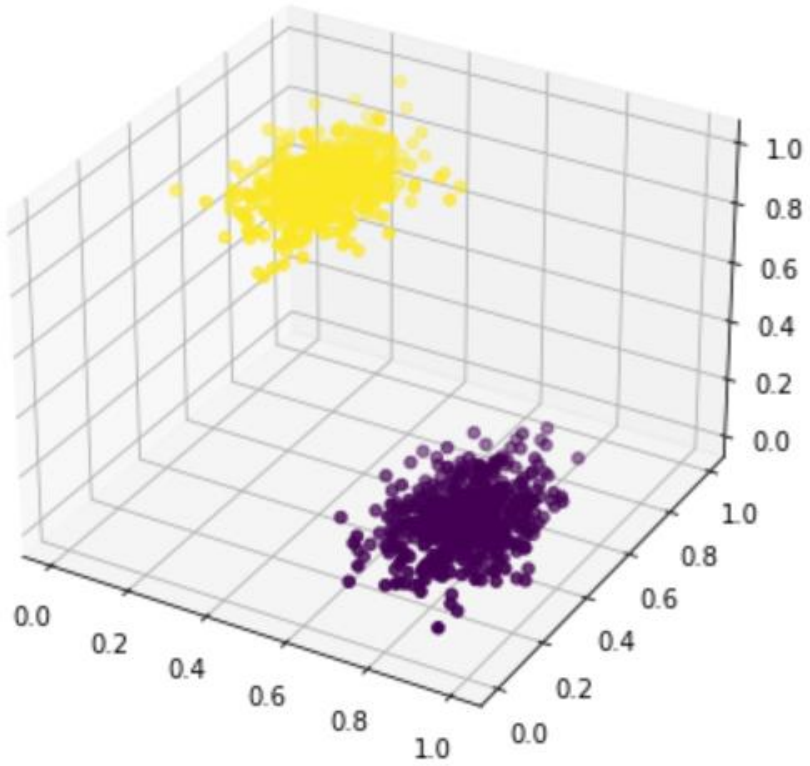
Step 2 - Then We only Use the Encoder

- We then use the Encoder to encode (compress) new data



Simple Example: Separating Blobs

- We define a set of spatially separated blobs with 2 centres in 3D
- We can see that these are separated by a surface in the x-y plane
- The z-plane (height) can also be used but it is not really needed



How to Make Blobs using Scikit Learn

- The code to do this is fairly simple

```
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from mpl_toolkits.mplot3d import Axes3D
```

```
data = make_blobs(n_samples=1000,
                  n_features=3,
                  centers=2,
                  random_state=101)
```

```
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data[0])
```

```
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
data_x = scaled_data[:,0] #col 1
data_y = scaled_data[:,1] #col 2
data_z = scaled_data[:,2] #col 3
ax.scatter(data_x, data_y, data_z, c=data[1]);
```

- We also do a min-max scaler

We Build the Corresponding Auto-Encoder

- We have 3 inputs but want to reduce to 2 dimensions
- I split the network into 3 layers
  - Input Layer with 3 neurons
  - Dense Encoder layer with 2 neurons
  - Dense Decoder layer with 3 neurons
- We therefore have an internal hidden layer with 2 neurons

```
inputs = Input(3)
encoder = Dense(2)
decoder = Dense(3)

model = Sequential([inputs, encoder, decoder], name='autoencoder')
```

- The encoder layer is the bottleneck

Set up the Training

- I use Adam in a way that lets me specify the learning rate

```
learning_rate = 0.1
optimizer = tf.keras.optimizers.Adam(learning_rate)
model.compile(optimizer=optimizer, loss='mean_squared_error', metrics=['mse'])
```

- The resulting auto-encoder is as follows

```
model.summary()
```

```
Model: "autoencoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_6 (Dense)             (None, 2)                 8

 dense_7 (Dense)             (None, 3)                 9

=================================================================
Total params: 17
Trainable params: 17
Non-trainable params: 0
_________________________________________________________________
```

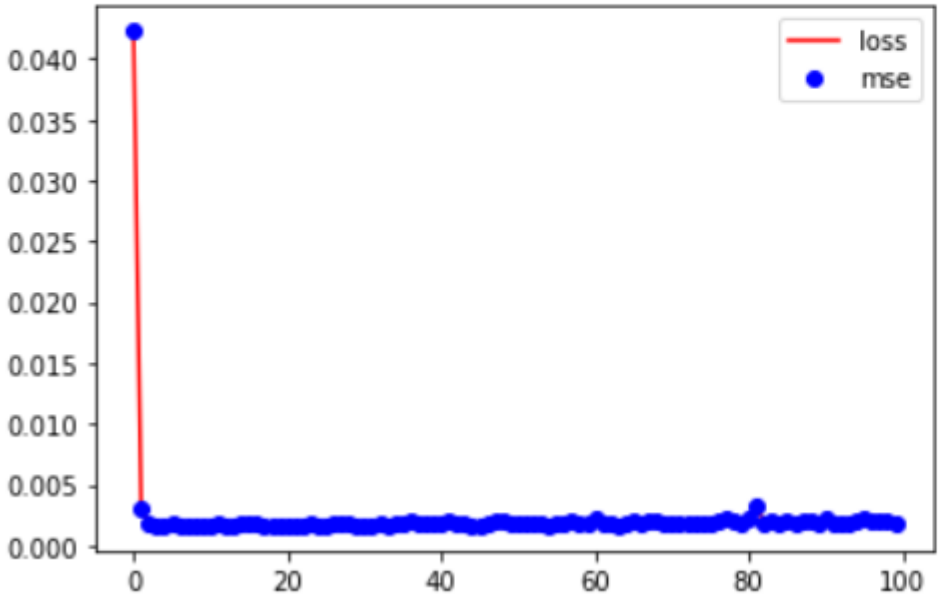
3 neurons to 2 neurons is 6 weights + 2 biases = 8 params

2 neurons to 3 neurons is 6 weights + 3 biases = 9 params

Fitting to the Input Data

- The convergence is fast and then it flat lines

```
history = model.fit(x=scaled_data, y=scaled_data, epochs=100, verbose=False)
plt.plot(history.history['loss'], 'r-', label='loss')
plt.plot(history.history['mse'], 'bo', label='mse')
plt.legend();
```



- Looks good – what has it found for the internal representation?

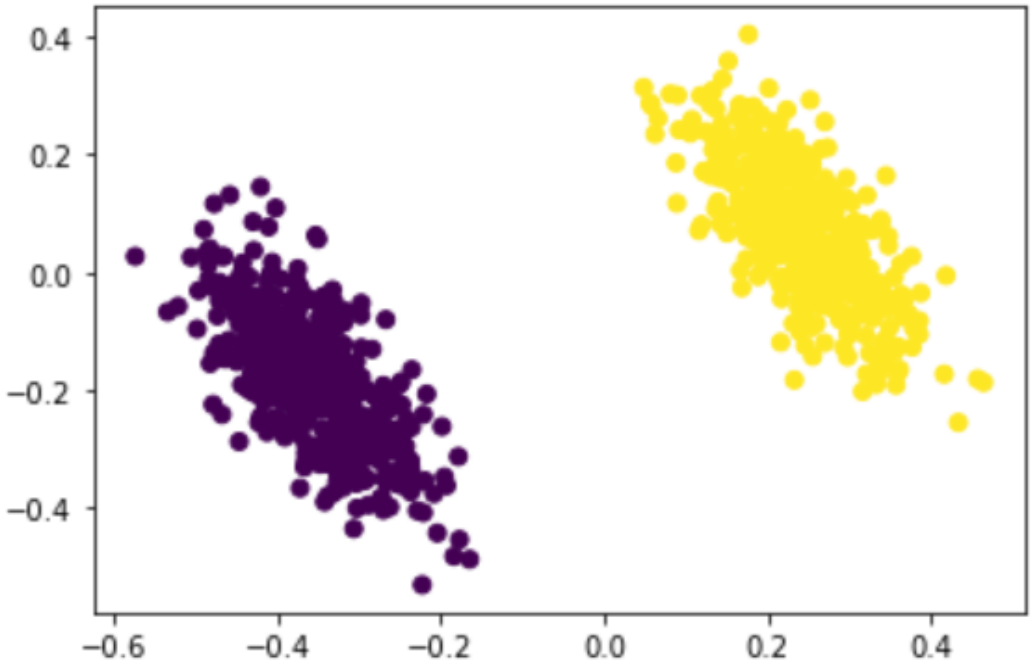
Data has been Compressed from 3D to 2D

- We examine the output of the encoding layer

```
encoded = encoder(scaled_data)
```

We see what the input data is represented as.

```
plt.scatter(encoded[:,0], encoded[:,1], c=data[1]);
```



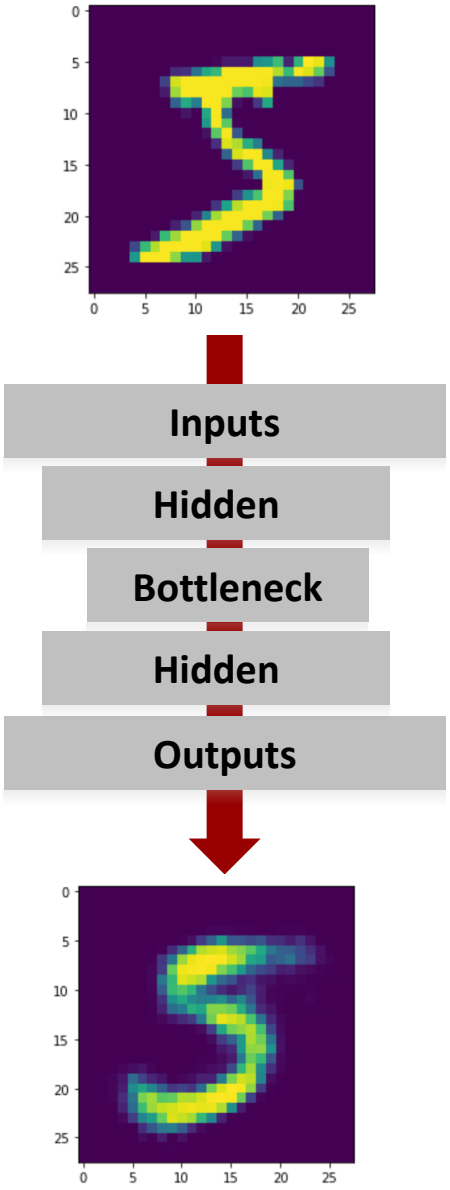
Summary of Example

- We did this trivial example just to show how Autoencoders work
- We already saw that the classes could be separated
- But the Auto-Encoder made it simpler by removing unnecessary dimensions
- If we were going from 10 dimensions then it would not be so easy to visualise
- Let's consider a more challenging problem – the MNIST handwritten digit data

Example: MNIST Auto-Encoder

MNIST Auto-Encoder Task

- The challenge here for the NN is to be able to reconstruct the digits
- It needs to learn the important features in each digit
- It needs to learn which of the 784 pixels are important
- Each pixel is a feature
- For example it will learn that the top-most and bottom-most rows have no information
- It will learn what is signal and what is noise and maybe remove the noise !



Building a Stacked Auto-Encoder

- Build a sequential feedforward ANN with several hidden layers
- We split the network into an encoder and a decoder

The Encoder

- The first encoder layer takes in the 28 x 28 image i.e. 784 pixels
- Subsequent layers drop to 400, 200, 100, 50 and then 20 neurons

The Decoder

- This stars with 20 inputs
- It then grows to 50, 100, 400 neurons
- The final layer has 784 sigmoidal outputs

The Building Code in TensorFlow-Keras

- The code is very straightforward

```
encoder = Sequential()
encoder.add(Dense(400,  activation='relu', input_shape=(784,)))
encoder.add(Dense(200,  activation='relu'))
encoder.add(Dense(100,  activation='relu'))
encoder.add(Dense(50,   activation='relu'))
encoder.add(Dense(20,   activation='relu'))
```

```
decoder = Sequential()
decoder.add(Dense(50,  input_shape=[20], activation='relu'))
decoder.add(Dense(100,  activation='relu'))
decoder.add(Dense(200,  activation='relu'))
decoder.add(Dense(400,  activation='relu'))
decoder.add(Dense(784,  activation='sigmoid'))
```

```
autoencoder = Sequential([encoder, decoder])
```

- Doing the encoder and decoder separately makes it easier

Compiling the Model

- As the output digits take a number between 0 and 1 we use binary cross entropy – we could also have used MSE

```
autoencoder.compile(loss='binary_crossentropy',
                    metrics = ['accuracy'],
                    optimizer = Adam())
```

```
autoencoder.summary()
```

```
Model: "sequential_34"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_28 (Sequential)  (None, 20)                420370

 sequential_33 (Sequential)  (None, 784)               421134

=================================================================
Total params: 841,504
Trainable params: 841,504
Non-trainable params: 0
_________________________________________________________________
```

We fit the Output to the Input

- Aim is to find a low level representation for handwritten digits
- We then fit the model to output the input training data
- We also give it the test set to use to as a validation set

```
trained_model = autoencoder.fit(x_train_s, x_train_s, batch_size=1024,
                                epochs=10, verbose=1,
                                validation_data=(x_test_s, x_test_s))
```

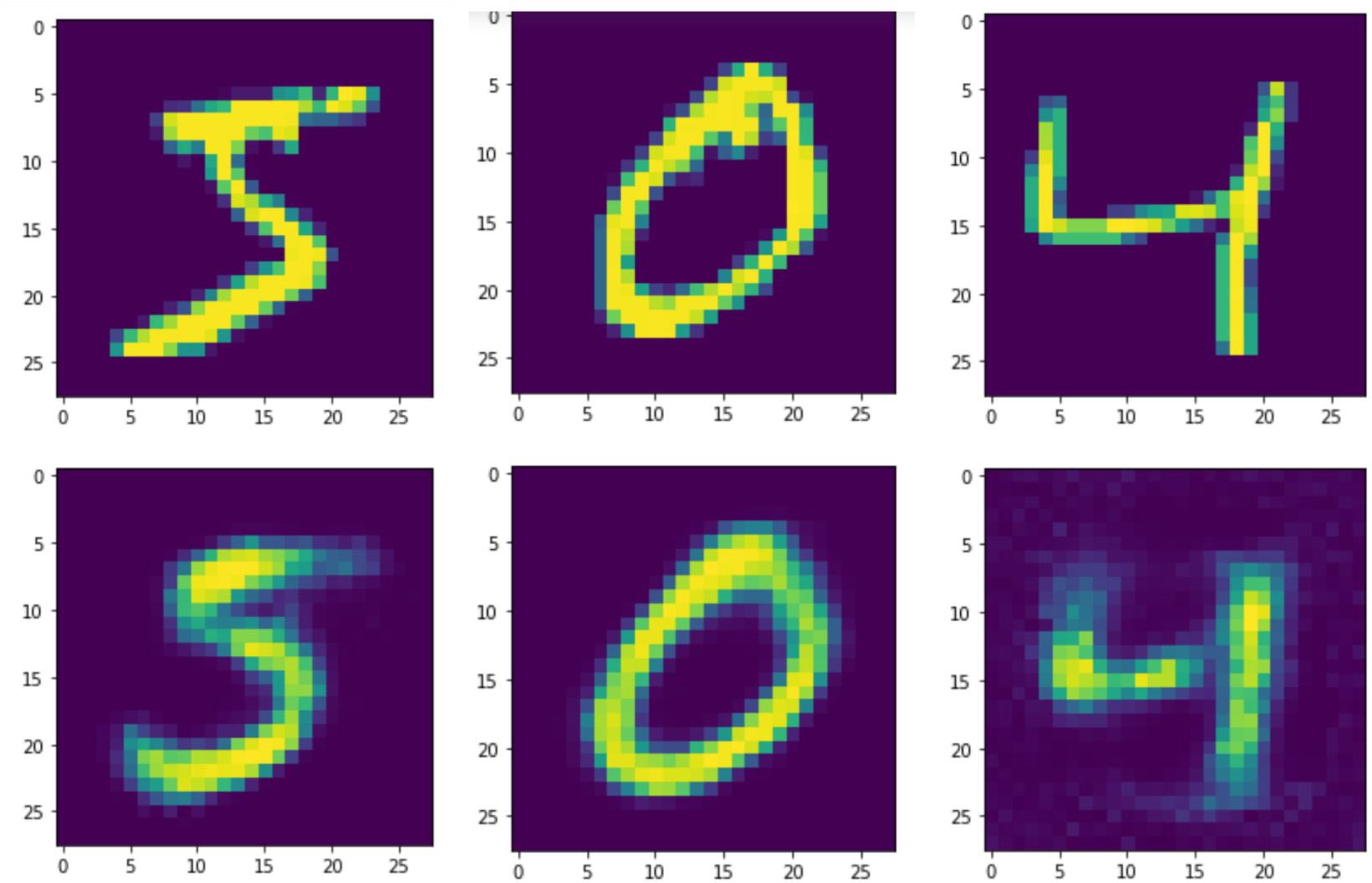
Reconstruction

- Remember – we have squeezed our 784 pixel image through a layer of just 20 neurons
- That means that we have compressed it to 20/784=2.6% of its initial size
- The question now is how close is the output image to the input image
- We compare input and output of the whole auto-encoder

It works ! This is impressive.

Input

Output



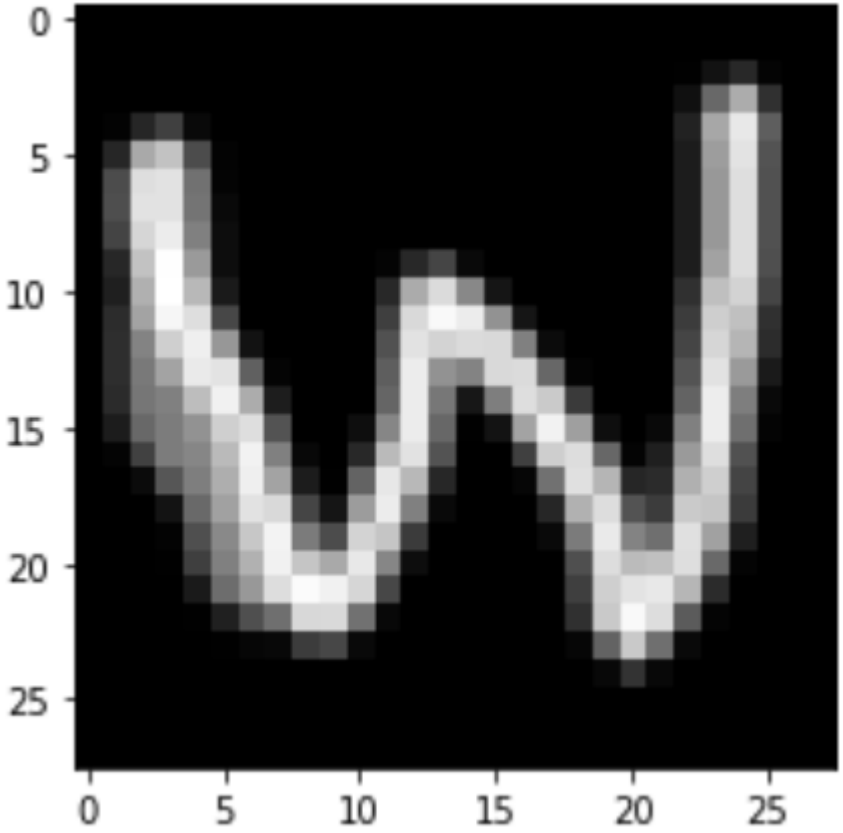
[5.20605446]

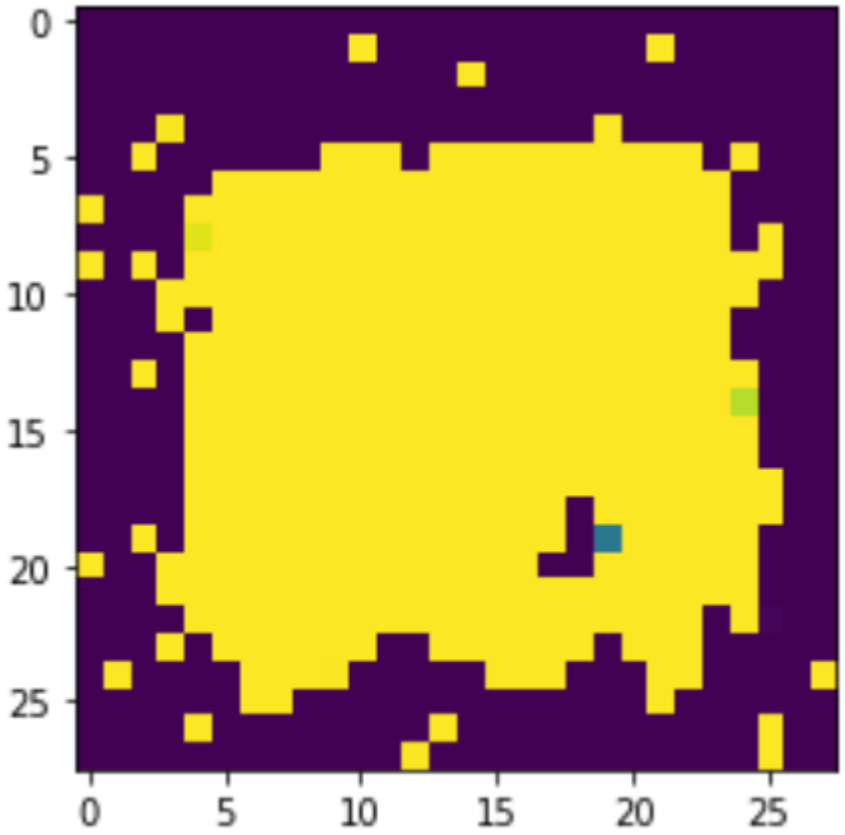
[3.27266968]

[5.07881695]

What if we pass in a "non-digit" ?

- I made a 28 x 28 image by hand
- It cannot be reconstructed – error is 2312 – ANOMALY !





[2312.5696]

Conclusions

- Not only is the image being recovered, it is also coming back "cleaner"
- It can do this because it "knows" what the average trained image looks like
- If it sees something it does not know it fails
- This is a powerful way to detect anomalies – things that do not conform to a trained pattern
- Convolutional autoencoders work better for large images
- You can also construct recurrent autoencoders for time series or text-based ML

Case Study: Detecting Credit Card Fraud using an Auto-Encoder

Auto-Encoder for Detecting Credit Card Fraud

- The creditcard.csv dataset contains records of actual credit card transactions with anonymized features
- Therefore, we will have to rely on end-to-end learning methods in order to build a good fraud detector
- We load the data and drop the Time feature and make the class feature the label to predict
- We then scale all the features to be in the range [0,1]
- We create a simple auto-encoder with one hidden layer as follows where we reduce the number of neurons to 12

```
data_in = Input(shape=(29,))
encoded = Dense(12,activation='tanh')(data_in)
decoded = Dense(29,activation='sigmoid')(encoded)
autoencoder = Model(data_in,decoded)
```

Training the Auto-Encoder

- As before, we simply train it to itself – minimising the mean squared error – we could use the binary cross entropy but I think MSE is better

```
autoencoder.compile(optimizer='adam',loss='mean_squared_error')
```

```
autoencoder.fit(X_train,
                X_train,
                epochs = 20,
                batch_size=128,
                validation_data=(X_test,X_test))
```

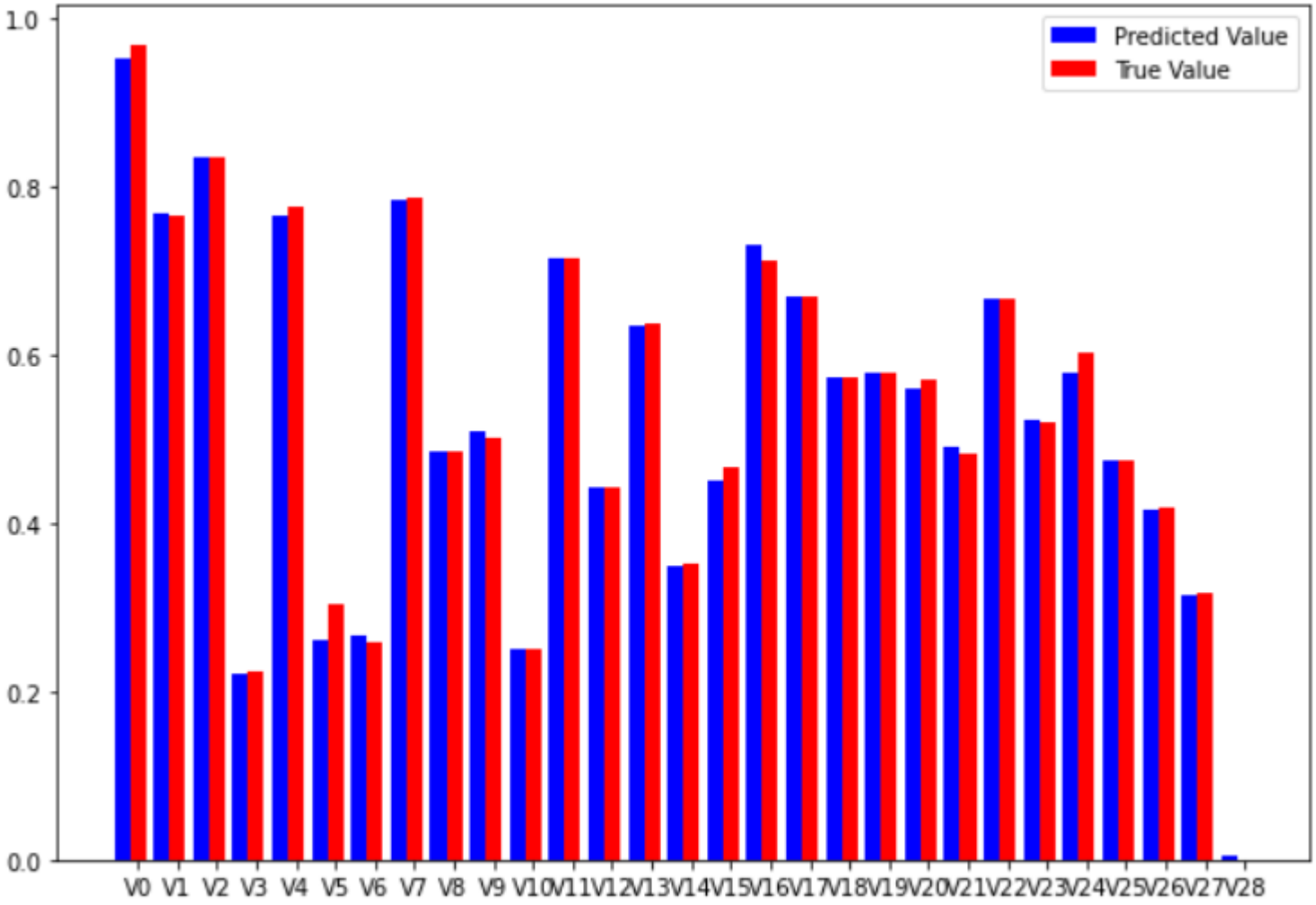
- To see how well it does, we predict for a subset of our data set and compare it to the inputs it is attempting to learn to recover

```
pred = autoencoder.predict(X_test[0:10])
```

- We then compare the predicted features

Agreement looks good for the chosen input

- Comparison of the input features with the output reconstruction



How do we identify Fraudulent Transactions ?

- We believe the fraudulent transactions will map differently from the non-fraudulent transactions in the encoded representation
- But there are 12 neurons in this representation ?
- We map it to a 2D space using a method called T-Distributed Stochastic Embedding or TSNE for short
- We build the encoder model from the first two layers

```
encoder = Model(data_in, encoded)
```

- We predict the internal representation for all transactions

```
enc = encoder.predict(X_test)
```

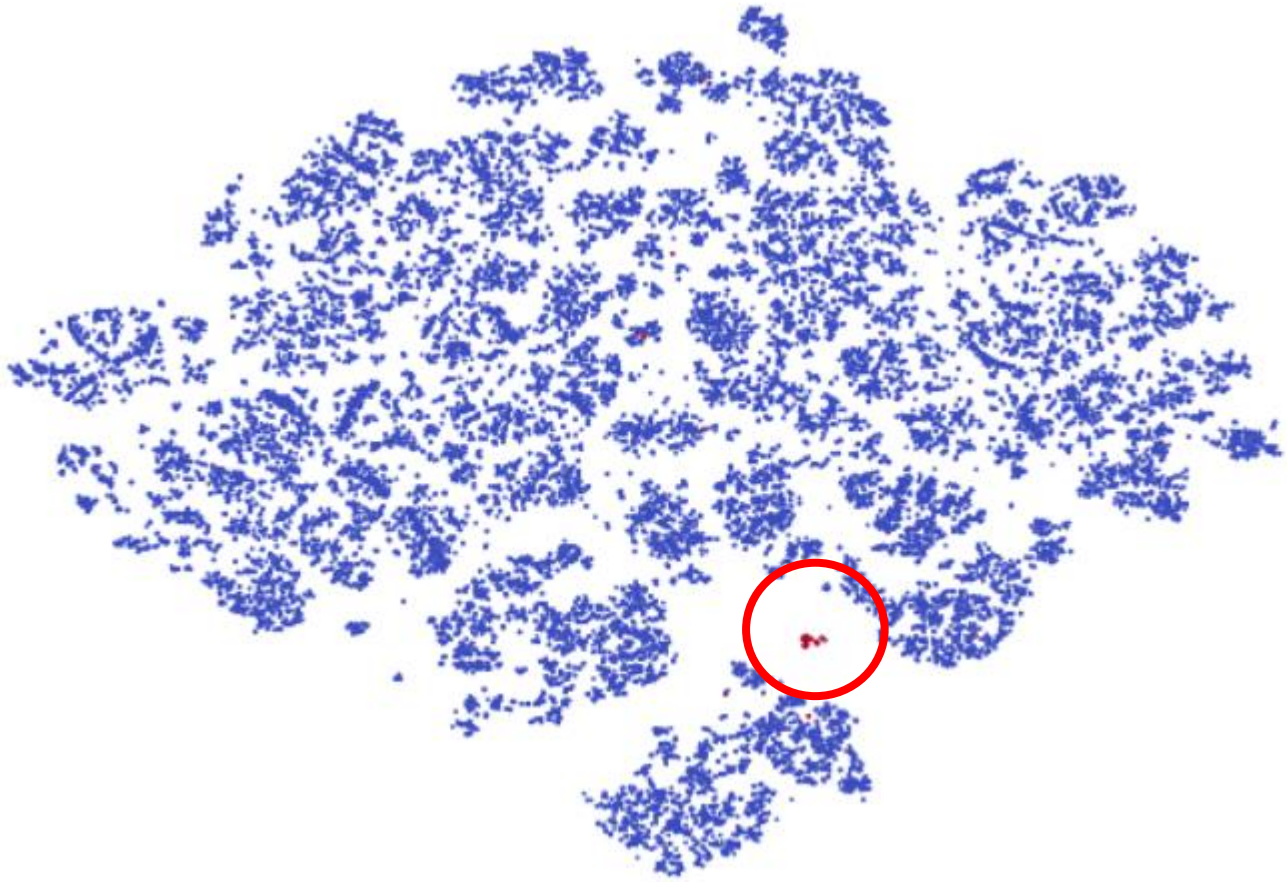
- Then we map it to a 2D space

```
tsne = TSNE(verbose=1, n_iter=5000)
```

```
res = tsne.fit_transform(enc)
```

Identifying Fraudulent Transactions

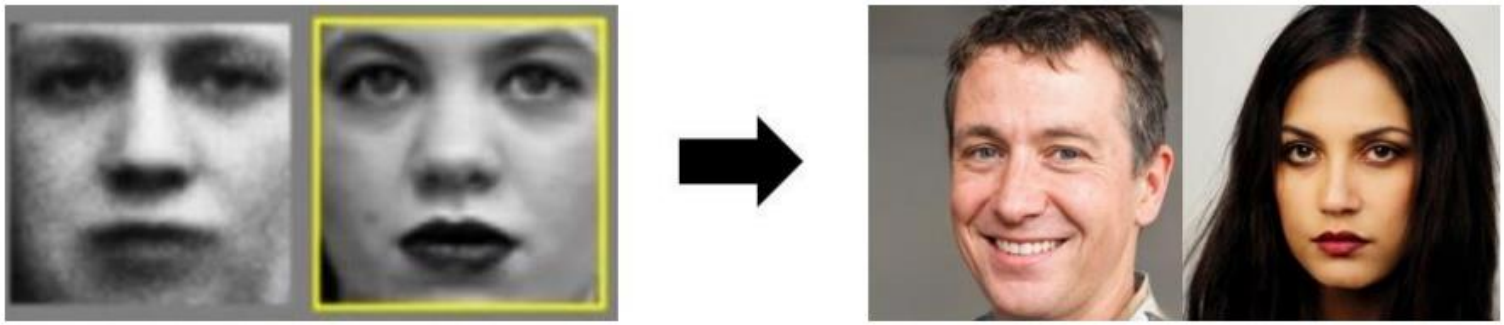
- We notice that in this space, fraudulent transactions appear to be mostly clustered in one region



Generative Adversarial Networks

Introduction to General Adversarial Networks

- GANs were invented in 2014 by Ian Goodfellow
- Class of neural network models used for unsupervised learning
- Consists of two competing ANNs – **generator** and **discriminator**
  - The generator tries to generate realistic data from noise
  - The discriminator trues to tell real data from fake data
- GANs are most frequently used for image-based problems
- They are the technology behind many of the "deep fake" images you may have seen – examples from 2014 and 2020 ! Progress!



GANs in Finance

- In Finance they are used for Market prediction, Tuning of trading models, Synthetic data generation, Portfolio Management, Optimization, Fraud Detection
- You can find a good review here

GENERATIVE ADVERSARIAL NETWORKS IN FINANCE:
AN OVERVIEW

A PREPRINT

Florian Eckerli *
School of Engineering
Zurich University of Applied Sciences
Winterthur, Switzerland
e.florian@hotmail.com

Joerg Osterrieder*
School of Engineering
Zurich University of Applied Sciences
Winterthur, Switzerland
joerg.osterrieder@zhaw.ch

The Hightech Business and Entrepreneurship Group
Faculty of Behavioural, Management and Social Sciences
University of Twente
Enschede, Netherlands
joerg.osterrieder@utwente.nl

June 11, 2021

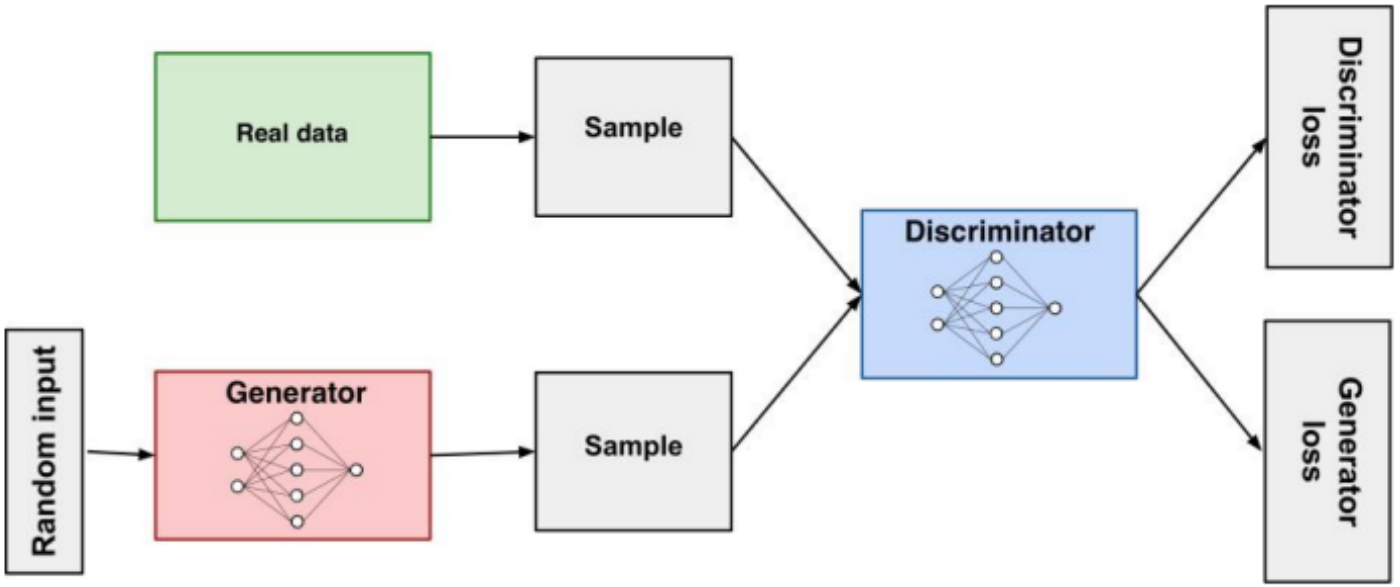
ABSTRACT

The Architecture of a GAN

- Understanding GANs is not easy, but once you do you will see that it's a clever idea
- We have two components:
  - A Generator
  - A Discriminator
- The generator generates data, but it is not real data
- The neural network also receives real data
- The discriminator provides feedback to the generator about whether the image is real or fake – it attempts to learn to distinguish between real and fake

The Process of a GAN

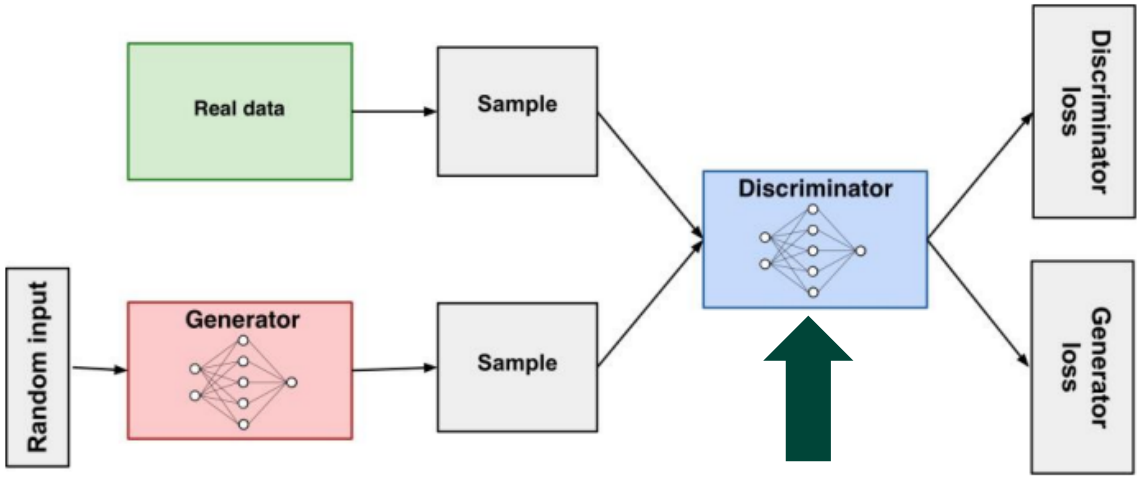
- Random input goes into the **generator**
- It then processes that – it is a feedforward neural network - to produce a synthetic data sample



- The **discriminator** is a NN that attempts to identify real vs synthetic, and the generator is trained to make this task harder

Training the GAN: Step 1

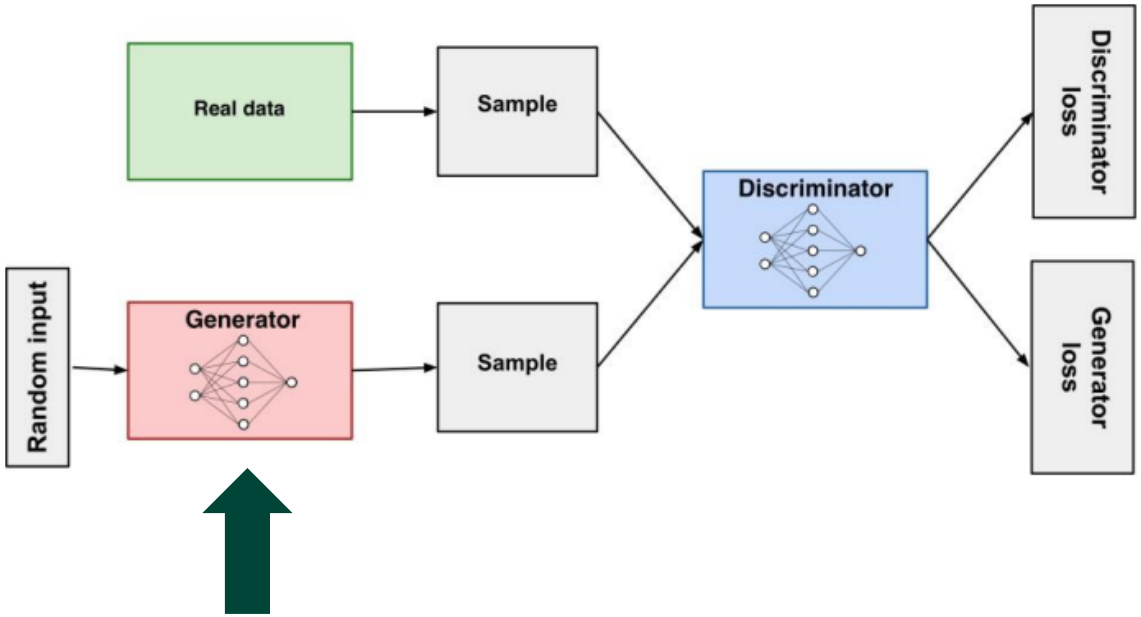
- Step 1 is to train the **discriminator** to detect if an input image is real or fake – the fake images are generated by the generator
- This is done by giving it a both real images and fake images, giving the discriminator labels so it knowns which is which
- Then using backpropagation, the **discriminator** changes its weights that make it better at detecting real versus fake images
- The **discriminator** becomes a good fake image detector



Training

Training the GAN: Step 2

- Step 2 is to use the **generator** to produce fake images
- Only fake images are fed to the discriminator and labelled as real
- The generator is then trained to create images that the discriminator identifies as real
- The backpropagation acts only on the generator weights



Problems with GANs

- The generator is always trying to trick the discriminator
- But sometimes this goes too far and we have **mode collapse**
- This happens if the Generator learns to produce just one image really well - it always fools the discriminator
- It means that the generator is only able to produce one type of image – the richness and power of GANs is lost
- This is usually associated with a local minimum of the training cost function
- GANs can be very unstable to train – they may start out fine but then begin to oscillate without any apparent reason
- They are also sensitive to the choice of hyperparameters

Example: MNIST GAN

Building a GAN to Generate zeros

- Let's load MNIST handwritten numbers and extract just the 0s
- We can do this for any of the 10 choices - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

```
from tensorflow.keras.datasets import mnist
```

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
y_train
```

```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

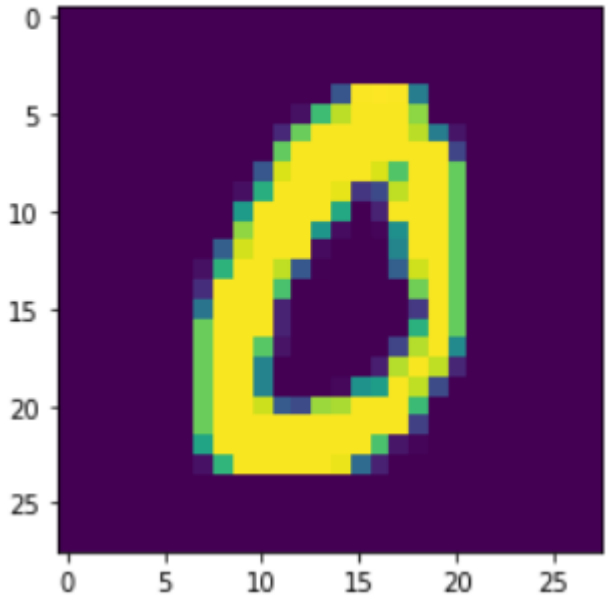
We want to select a particular digit - just zeroesm~

```
only_zeros = X_train[y_train==0]
```

```
only_zeros.shape
```

```
(5923, 28, 28)
```

```
plt.imshow(only_zeros[12]);
```



- As expected, we get back about 6,000 images – about 10% of the 60k Images in the MNIST training data set

Building the GAN: The Discriminator

- Let's start with the Discriminator which takes in a 28 x 28 image and attempts to determine if it is real or fake – a sigmoid output
- We build it as a Sequential ANN with 2 hidden RELU layers of 150 and then 100 neurons

```
discriminator = Sequential()
discriminator.add(Flatten(input_shape=[28,28]))
discriminator.add(Dense(150, activation='relu'))
discriminator.add(Dense(100, activation='relu'))
discriminator.add(Dense(1, activation='sigmoid'))
```

```
discriminator.compile(loss='binary_crossentropy', optimizer='adam')
```

- We compile it – this is a standalone network
- It outputs a binary value – whether it matched the 0 or 1
- We also set it to be non-trainable initially – we will change this

```
discriminator.trainable = False
```

Building the GAN: The Generator

- The Generator takes in noisy input and then generates an image
- We need to decide on how much information the input layer gets
- I call this the codings_size and set it to equal 100
- All of the intermediate layer activations are RELU

```
codings_size = 100
```

```
generator = Sequential()
generator.add(Dense(100, activation='relu', input_shape=[coding_size]))
generator.add(Dense(150, activation='relu'))
generator.add(Dense(784, activation='relu'))
generator.add(Reshape([28,28]))
```

- The final output is a full 28 x 28 generated image
- We don't compile the generator – it is only used together with the discriminator as the GAN

Building and Preparing the GAN

- The GAN combines the generator and the discriminator
- We compile this - our cost function is binary cross entropy

```
GAN = Sequential([generator, discriminator])
```

```
GAN.compile(loss='binary_crossentropy', optimizer='adam')
```

```
GAN.summary()
```

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_10 (Sequential)  (None, 28, 28)            143634

 sequential_7 (Sequential)   (None, 1)                 132951

=================================================================
Total params: 276,585
Trainable params: 143,634
Non-trainable params: 132,951
_________________________________________________________________
```

Batching the Data

- We want to train the GAN in batches – it can do it in parallel
- We decide on batch sizes of 32
- Hence, we break up the 5,923 images into batches of 32
- This gives us 185.09375 batches (we discard the decimal places)

```
batch_size = 32
my_data = only_zeros
dataset = tf.data.Dataset.from_tensor_slices(my_data).shuffle(buffer_size = 1000)
dataset = dataset.batch(batch_size, drop_remainder = True).prefetch(1)
```

The number of batches is

```
len(only_zeros)/32
```

```
185.09375
```

```
len(dataset)
```

```
185
```

- Each batch has dimensions 32 x 28 x 28

Training the GAN: The whole process

The algorithm is as follows:

1. Create random inputs (noise) for the generator
2. Pass these random inputs into the generator
3. Extract the 28 x 28 fake image from the generator output
4. Combine this fake image with a real image and label the first fake one as False and the second real one as True
5. Train discriminator to detect which image is real, which is fake
6. Freeze the discriminator weights
7. From new random inputs, use the generator to output images
8. Train the generator weights of the GAN to learn to output images that the discriminator thinks are real
9. Go to 1 until all batch images trained

Training the GAN - code

- The code for training the GAN is as follow

```
for X_batch in dataset:
    i = i + 1

    if i%10 == 0:
        print("Epoch:", epoch, "Batch ", i)

    noise = tf.random.normal(shape = [batch_size, codings_size])
    gen_images = generator(noise)
    X_batch_32 = tf.dtypes.cast(X_batch, tf.float32)
    X_fake_vs_real = tf.concat([gen_images, X_batch_32], axis=0)
    y1 = tf.constant(labels)
    discriminator.trainable = True
    discriminator.train_on_batch(X_fake_vs_real, y1)

    noise = tf.random.normal(shape = [batch_size, codings_size])
    y2 = tf.constant([1.0] * batch_size)
    discriminator.trainable = False
    GAN.train_on_batch(noise, y2)
```

- Labels is the vector of 1s and 0s to label real and fake images

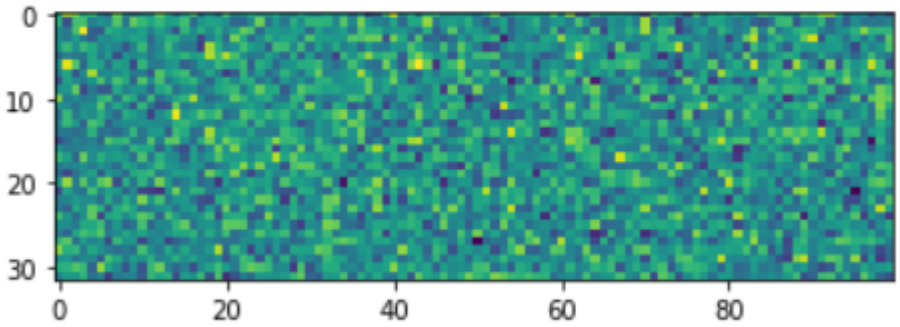
```
labels = [[0.0]] * batch_size + [[1.0]] * batch_size
```

Generating Images from the Generator

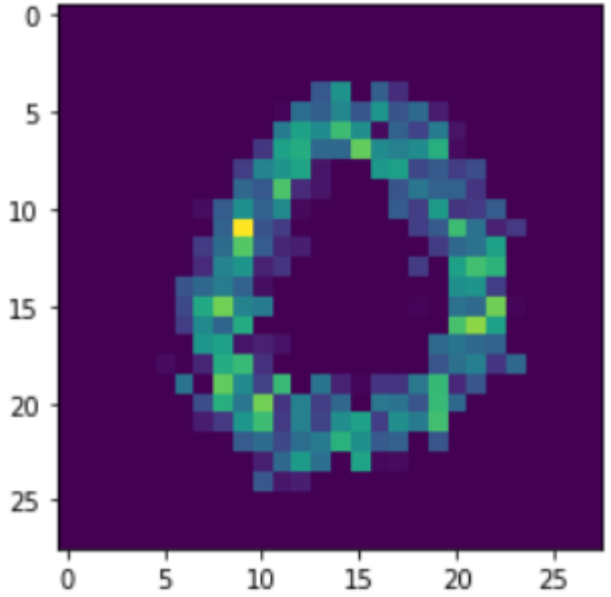
- I pass in some noise and see what the generator produces

```
noise = tf.random.normal(shape = [10, codings_size])
```

```
plt.imshow(noise);
```




```
images = generator(noise)
```



- I get a noisy zero! From noise.
- The generator has learned to generate a zero.

Problems with Training GANs

- Sometimes we train GANs to generate different classes of image
- This does not always work – the system can end up just being able to recover one class
- There's a game between the generator and discriminator
- The generator needs to find a set of weights that produces realistic images until the discriminator is only right half the time
- This is the final equilibrium state – but is it always reached ?

Machine Learning Applications

Lecture 8: Reinforcement Learning

Lecture Notes

M.Sc. in Financial Markets

ACADEMIC YEAR 2021-2022

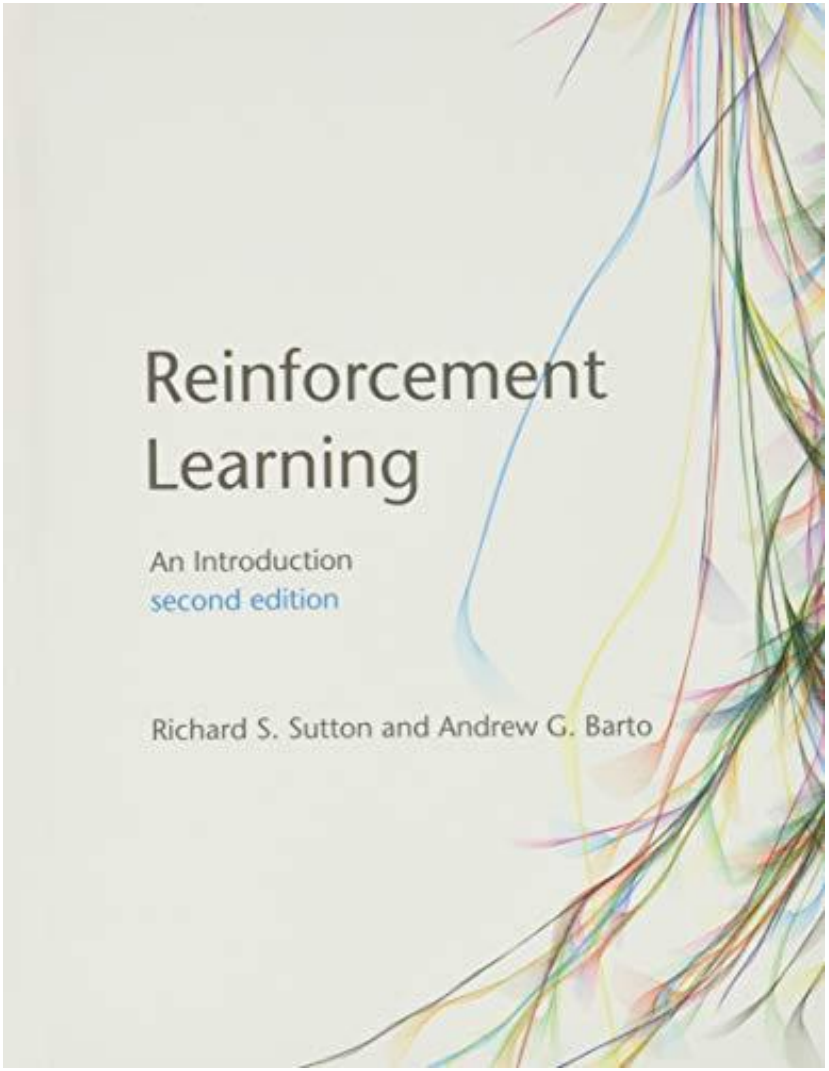
Dr. Dominic O'Kane

Version: January 2022

What is Reinforcement Learning (RL) ?

- RL is one of the most exciting and most active areas of ML
- It has a lot of applications in finance and so I want to spend a reasonable amount of time showing you how it works
- It has been shown that it is possible for reinforcement learning to master video games like Breakout and Pacman
- RL is not new – however the combination of RL and Deep Learning has led to some revolutionary breakthroughs in AI
- In 2016 a reinforcement learning model called AlphaGo developed by DeepMind was able to beat the world Go champion
- Google paid $500m to acquire DeepMind in 2018

The Bible of Reinforcement Learning



Applications of RL

- Huge range of applications being worked on
  - Autonomous Driving
  - Training Robots to Move
  - Text summarization and Q&A
  - Task Scheduling

Finance Applications of RL

- In Finance RL is used for the following applications
  - Securities trading – trading bots
  - Chatbots
  - Multi-period Portfolio Optimization
  - Portfolio management
  - Optimal stock execution
  - Option hedging

Latest: DeepMind are applying RL to Fusion Research

Article

Magnetic control of tokamak plasmas through deep reinforcement learning

https://doi.org/10.1038/s41586-021-04301-9

Received: 14 July 2021

Accepted: 1 December 2021

Published online: 16 February 2022

Open access

Check for updates

Jonas Degrave[1,3], Federico Felici[2,3]✉, Jonas Buchli[1,3]✉, Michael Neunert[1,3], Brendan Tracey[1,3]✉, Francesco Carpanese[1,2,3], Timo Ewalds[1,3], Roland Hafner[1,3], Abbas Abdolmaleki[1], Diego de las Casas[1], Craig Donner[1], Leslie Fritz[1], Cristian Galperti[2], Andrea Huber[1], James Keeling[1], Maria Tsimpoukelli[1], Jackie Kay[1], Antoine Merle[2], Jean-Marc Moret[2], Seb Noury[1], Federico Pesamosca[2], David Pfau[1], Olivier Sauter[2], Cristian Sommariva[2], Stefano Coda[2], Basil Duval[2], Ambrogio Fasoli[2], Pushmeet Kohli[1], Koray Kavukcuoglu[1], Demis Hassabis[1] & Martin Riedmiller[1,3]

Nuclear fusion using magnetic confinement, in particular in the tokamak configuration, is a promising path towards sustainable energy. A core challenge is to shape and maintain a high-temperature plasma within the tokamak vessel. This requires high-dimensional, high-frequency, closed-loop control using magnetic actuator coils, further complicated by the diverse requirements across a wide range of plasma configurations. In this work, we introduce a previously undescribed architecture for tokamak magnetic controller design that autonomously learns to command the full set of control coils. This architecture meets control objectives specified at a high level, at the same time satisfying physical and operational constraints. This approach has unprecedented flexibility and generality in problem specification and yields a notable reduction in design effort to produce new plasma configurations. We successfully produce and control a diverse set of plasma configurations on the Tokamak à Configuration Variable[1,2], including elongated, conventional shapes, as well as advanced configurations, such as negative triangularity and 'snowflake' configurations. Our approach achieves accurate tracking of the location, current and shape for these configurations. We also demonstrate sustained 'droplets' on TCV, in which two separate plasmas are maintained simultaneously within the vessel. This represents a notable advance for tokamak feedback control, showing the potential of reinforcement learning to accelerate research in the fusion domain, and is one of the most challenging real-world systems to which reinforcement learning has been applied.

Many Different Types of Reinforcement Learning

- Markov Decision Processes (MDPs)
  - **Value and Q-Value Iteration**
  - Temporal Difference Learning
  - Q Learning
  - Approximate Q-Learning
  - Deep Q-Learning
- **Policy Gradients**
- There is a lot! We will only be able to do a small bit today.
  - Policy Gradients using a simple CartPole example
  - Q-Value iteration using a simple example
- Finance applications would require a full course on RL

Reinforcement Learning is Different

- You have done a lot of supervised learning
- In Reinforcement learning the approach is very different
- The model learns to play Pong using a simple reward system
- The game playing can be automated
- RL teaches the model to learn via trial and error
- The model needs to explore and engage with the environment
- Allowing the model to choose the model state and reward is key to getting a working model

When should we use Reinforcement Learning ?

- Sometimes we don't have a lot of training examples
- But we do know how the system works – we can simulate the impact of an action on the state of the system
- Examples:
  - A rules-based game like Chess or Go
  - A physics simulation of a robot learning to walking
  - A video game like Breakout, Space Invaders, Pong
- Rather than learn from examples, we learn by experiencing the environment
- It uses **feedback** in the form of a **reward** – we don't get the answer, just an indication that one action is better than another
- We find this one framework can solve many different problems!

The RL Framework

Reinforcement Learning

- There is a certain amount of mathematical formality
- Do not let it confuse you too much
- The idea is very simple
- We have a system that is operated by an agent who takes actions that change the system and produce rewards

The Terminology of Reinforcement Learning

- **Environment:** The world containing the agent and other objects
- **State:** The state of the environment S
- **Agent:** The thing in the environment that performs the actions
- **Policy:** The algorithm that decides upon actions given states
- **Action:** The act by the agent that may change the state
- **Rewards:** A metric for scoring the new state of the environment

Terminology: The State

- A state $\mathbf{S_t}$ is the description of the environment
- It may have few or many parameters – depends on the problem
- State variables may be discrete or continuous
- We may have an infinite number of possible states

Examples

- In chess the state is the position of all of the pieces on the board
- In poker we only have partial information.
- In a robotic arm it is the position-orientation of arm and joints
- In a trading algorithm it is all of the information you want to include

Terminology: The Policy-Action

- A policy is how the choice of a specific action is made
- Mathematically it is a mapping from the state $\mathbf{S_t}$ to the action **a**
- We can have a deterministic state-dependent policy

$$\mathbf{a = \pi(s)}$$

- This means we always do the same thing in the same state
- We can have a stochastic policy

$$\mathbf{\pi(a|s) = Pr(A=a|S=s)}$$

- A stochastic policy means we do not always do the same thing in the same state – the action only occurs with some probability

Terminology: The Reward

- After an action there is a reward R
- It may be positive or negative or zero – this depends on the environment and what you are trying to achieve
- The sequence of events is
  - State $\mathbf{S_t}$
  - Action $\mathbf{a_t}$
  - Reward $\mathbf{R_{t+1}}$
  - State $\mathbf{S_{t+1}}$
  - This then repeats until …
- The summing of the rewards is the Gain

Terminology: An Episode

- A trajectory is the sequence of the process stages
  - **State 0 -> Action 0 -> Reward 1 -> State 1 -> Action 1 -> Reward 2 -> State 2 -> Action 2 -> Reward 3 …..**
- This is often written as

$$\tau = S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3$$

- The episode is the start and end states

$$e = S_0 \rightarrow S_3$$

- The trajectory may or may not be never ending
- If the trajectory is never-ending, what happens to the Gain ?

Terminology: The Gain Function

- We define the **gain** as follows

$$G_t^{\pi}(S_t) = \sum_{i=0}^{T-t-1} \gamma^i R_{t+1}$$

- Rewards start being received at time t+1 and go out to time T
- The weight γ is a discount factor on the reward
- The further into the future the more the reward is discounted

Example

- In Chess the reward is taking the pieces
- Losing a piece is a negative reward !

Markov Decision Processes

The Markov Property

- A discrete time stochastic control process
- Stochastic as future states depend only partially on actions taken
- One basic assumption of MDP Reinforcement Learning is that the environment is **Markovian**
- The probability of being in a state today conditional on the previous state is the same as the probability of being in that state conditional on all previous states

$$\Pr\left[S_{t+1} \middle| S_t\right] = \Pr\left[S_{t+1} \middle| S_1, S_2, S_3, \ldots, S_t\right]$$

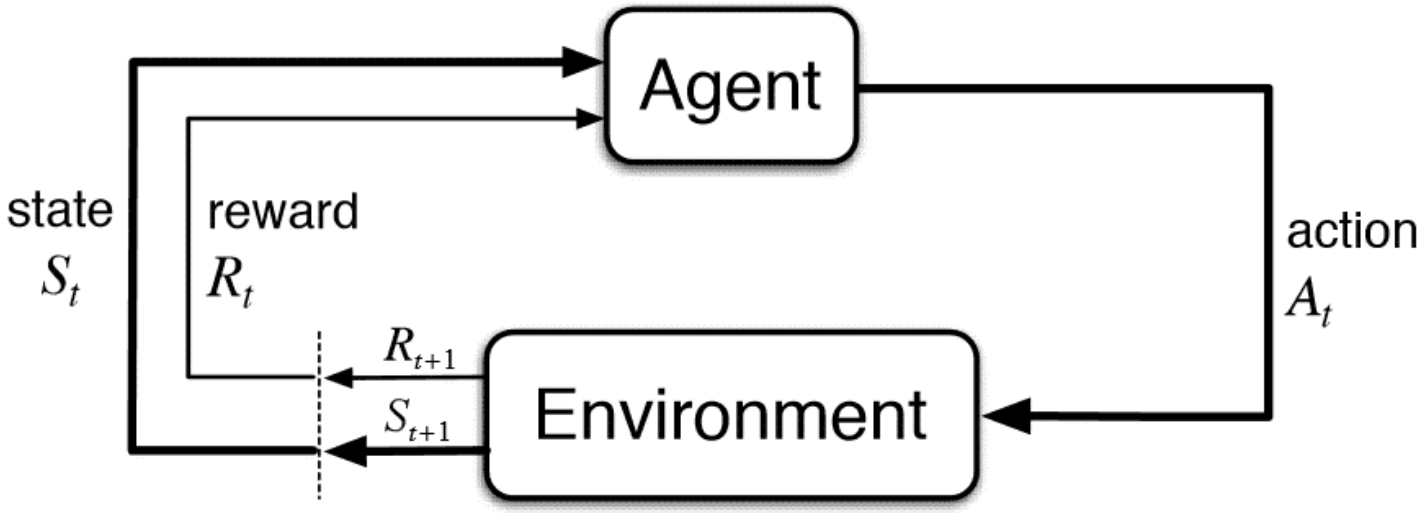
- We can throw away all information before the last state
- The state today tells us everything we can know about the future

Types of Markov Decision Process

- Finite MDP vs Infinite
  - Number of states, actions and rewards are finite
  - One of these three values is infinite
  - Examples:
    - Finite MDP – 5x5 Maze
    - Infinite MDP – Automated cars
- Episodic or Continuous
  - Episodic finishes when we reach some termination condition
  - Example: Chess – checkmate

Markov Decision Process

1. The **agent** performs an **action**
2. The **action** changes the environment
3. We have a new **state**, and a **reward** is calculated
4. The **agent** takes a new action based on the state and the reward



- We wish to wrap an algorithm around this to find what the optimal actions are – those that maximise the **reward**

The Bellman Equation

The Bellman Equation

- From now on, all RL problems are solving the Bellman Equation
- This is the fundamental equation of Dynamic Programming Control Problems and is based on two quantities
  - The Value
  - The Quality Value (Q-value)
- It is then possible to write an optimal control problem in terms of both the value and Q-value
- We can write the Optimal Control problem as an iteration on either of these two quantities
- Solving these equations can allow us to determine the optimal Value, Q-value and Policy

The Value Function

- The value function for a policy π and state **s** we want to maximise is the state-dependent **expectation** of the gain

$$V_\pi(s) = E_t^\pi \left[\sum_{i=0}^{T-t-1} \gamma^i R_{t+i+1} \mid S_t = s \right]$$

- The expectation is over the randomness in the system
  - The choice of actions (policy) may be random
  - The transitions may be random
  - The rewards may be stochastic
- Given the probabilities of the actions and the state transition probabilities we can write the expectation mathematically
- Otherwise we can use Monte Carlo to estimate the value

The Value in Terms of Transition Probabilities

- The probability of taking action **a** from state **s** is given by

$$\pi(a \mid s)$$

- The probability of ending in state s' with reward r from state s after action a is given by

$$p(s', r \mid s, a)$$

- We therefore get an equation as follows

$$V_\pi(s) = \sum_a \pi(s,a) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \cdot V_\pi(s')\right]$$

The Quality-Value Function

- The Q-value function for a policy π and state **s** taking action **a** is given by

$$Q_\pi(s,a) = E_t^\pi \left[\sum_{i=0}^{T-t-1} \gamma^i R_{t+i+1} \mid S_t = s, A_t = a \right]$$

- It is a more interesting quantity than the value function
- It gives us information on how the value is action dependent
- If we know the value of the Q-value function we can determine the value of taking a specific action today
- **If we want to act optimally, we should take the action with the highest Q-value**

Optimality

- To maximise the Value, we want a policy that chooses the best action in each state

$$V_*(s) = \max_\pi V_\pi(s)$$

- The best action is the one with the highest Q-value and we find this by searching over the policy (actions)

$$Q_*(s,a) = \max_\pi Q_\pi(s,a)$$

- So, the optimal value is the maximum Q-value over allowed actions in state s

$$V_*(s) = \max_{a \in A(s)} Q_\pi(s,a)$$

Optimality Bellman Equations

- We therefore get an iterative equation as before but where we must select the action that maximises the future value

$$V_*(s) = \max_a \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma \cdot V_*(s')\right]$$

- There is an equivalent optimal Bellman equation for the Q-value
- This time the choice of the immediate action is already known but the future actions are not

$$Q_*(s,a) = \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma \cdot \max_{a'} Q_*(s',a')\right]$$