Chapter 1 : What is Deep Learning

AI = effort to automate intellectual tasks normally performed by humans

Symbolic AI : a sufficiently large set of explicit rules for manipulating knowledge stored in explicit database
→ dominant paradigm 1950 - 1980

Although the Turing Test has sometimes been interpreted as a literal test Turing merely meant it as a conceptual device in a philosophical discussion about the nature of cognition

ML → a very hands-on field driven by empirical findings and deeply reliant on advances in software and hardware

"Deep" → for deep learning means successive layers of representation

loss function = objective function = cost function
↓
Backpropagation algorithm

There were already 2 AI winters → 1360, 1990

Probabilistic ML → application of the principles of statistic to data analysis
Ex. Naive Bayes (it assumes that all the data points are independent)
Logistic regression

Kernel Methods → classification algorithms best known for SVMs
1) the data is mapped to a new high-dimensional

representation where decisions boundaries can be expressed as hyperplanes

2) Find the decision boundary by maximising the distance between the hyperplane and the closest datapoints from each class

The thing is that it can be computationally expensive to map all the points to a higher dimensional space
=> kernel trick allows you to find the distance between two points from the initial space into the target space

Decision Trees -> flowchart-like structure that lets you classify stuff
=> Random forest.
=> Gradient Boosting Machines

Deep learning -> one of the reasons why it is so cool is because it automates feature engineering

Best libraries according to Kaggle Teams:
Keras, LightGBM, XGBoost, PyTorch, TensorFlow, Sklearn, FastAI, Caffe

2009 - 2010 -> better activation functions
↳ better optimisation schemes (RMSProp & Adam)

2011 - 2016 -> improved gradient descent
↳ batch normalisation, skip connections

Chapter 2: mathematical building blocks of neural networks

Softmax classification layer -> return an array of 10 probability scores (summing to 1)

optimiser -> the mecanism through which the model will update itself based on the training data it sees, so as to improve its performance

loss function -> how the model will measure its performance on the training data, and how it will be able to steer itself in the right direction

metrics to monitor during training and testing

tensor = container for data

number -> rank 0 tensor
vector rank-1 tensor
matrix rank-2 tensor
...

Tensor
- number of axes (rank)
- shape: tuple of integers that describes how many dimensions the tensor has along each axis
- datatype (or dtype)

Deep learning -> usually process data in batches (chunks)

Common data tensors

1) Vector data
usually rank 2 tensors (samples, features)

2) Time series data
usually rank 3 tensors (samples, timestamp, features)

3) Image tensors (samples, H, W, colors)
(128, 256, 256, 3)
samples -> height -> width -> RGB

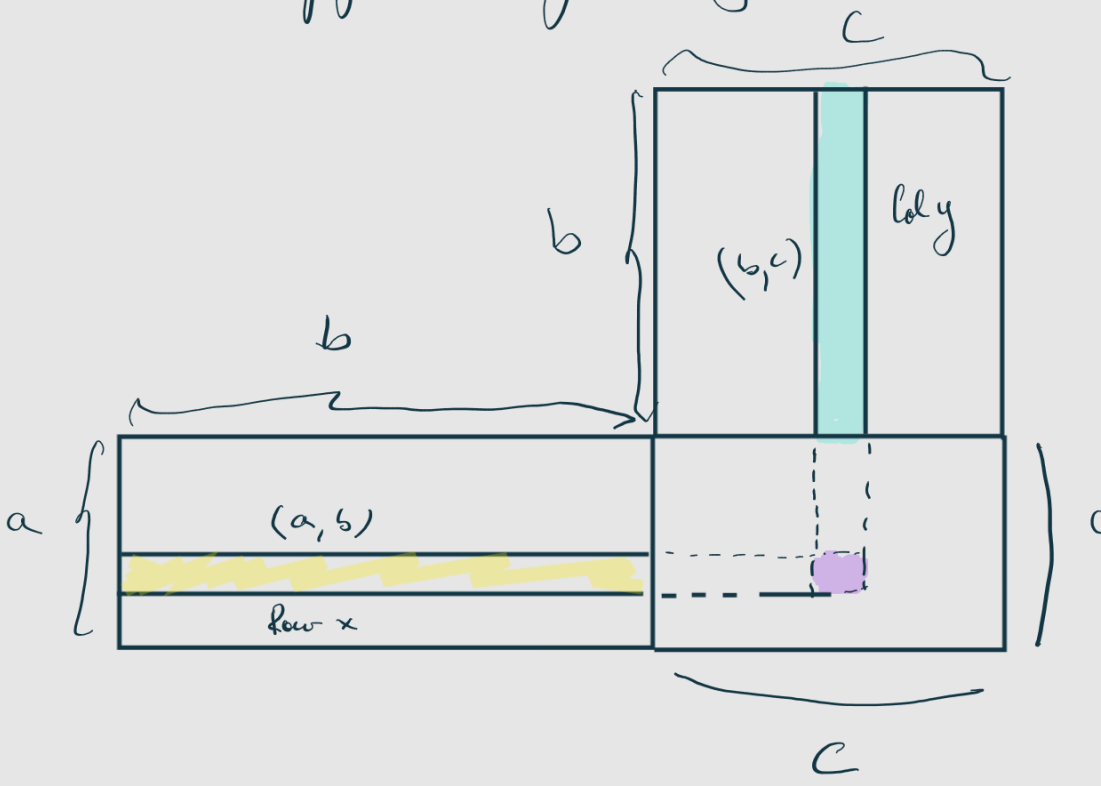
4) Video data
(samples, frames, H, W, colors)

Keras layers Dense(512, activation="relu")

=> identic output = relu(dot(input, W) + b)

Element wise operations -> naive = step by step on each

↳ python style using tensors

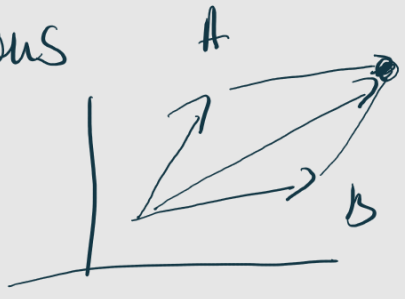


=> Reshaping Tensors

Geometric interpretations

1) adition

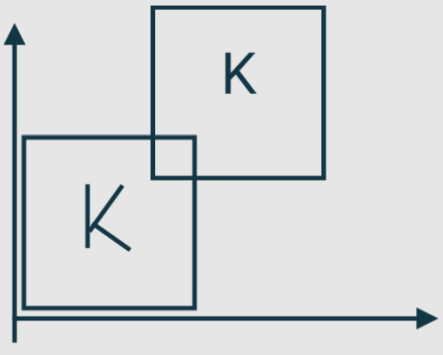
= translating an object



$\begin{bmatrix} \text{Horizontal} \\ \text{Vertical} \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$

2) Translation

= apply adition to a set of points



3) Rotation

=> counter clockwise rotation by angle θ => via dot product

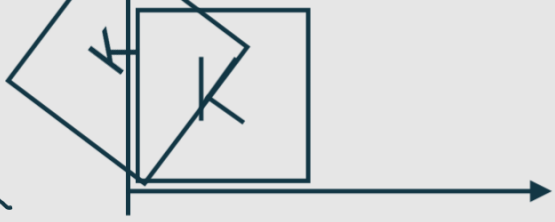
$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$

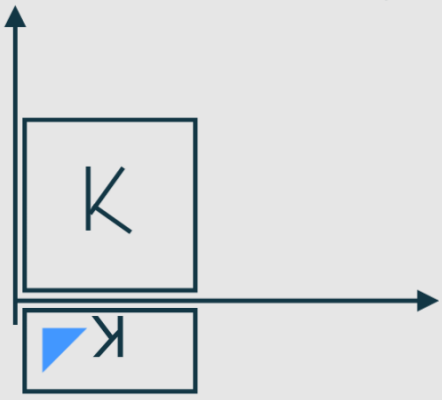
4) Scaling

=> vertical & horizontal

=> with a dot product with a diagonal matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & -0.5 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$



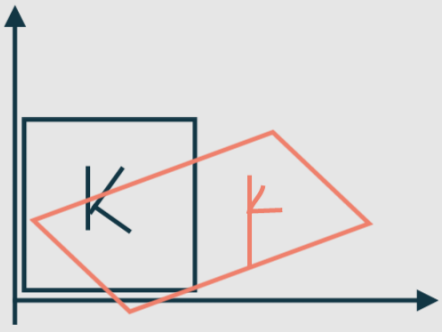


5) Affine transformations

=> combination of linear transformation and translation

$$y = W \cdot x + b$$

This is what a dense layer is performing. A dense layer without an activation function is an affine transformation



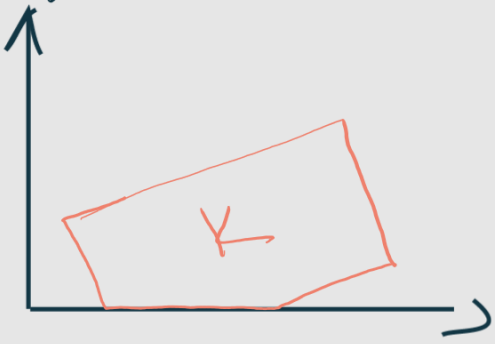
A dense layer with Relu activation

$\text{affine}_1(\text{affine}_2(x))$ Equivalent $\longrightarrow$ $\text{affine}_3(x)$

where affine 3 can be computed

=> no need for multiple layers

affine + Relu $\longrightarrow$



Gradient based optimisation

- weight
- bias

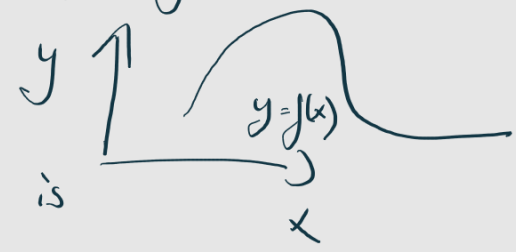
Training loop

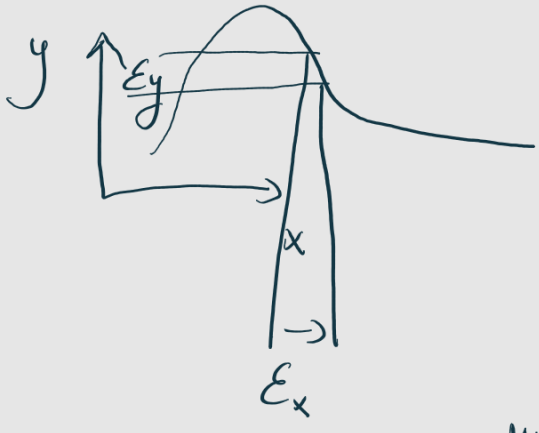
1) Draw a batch of training samples
2) Forward pass => obtain predictions

3) Compute the loss of the model on the batch
4) Update the weights of the model in a way that slightly reduces the loss of the batch

$f(x) = y$

because the fuction is continuous





$f(x + \varepsilon_x) = y + a \cdot \varepsilon_x$

The slope a = derivative

$a < 0 \Rightarrow x \uparrow, f(x) \downarrow$

$a > 0 \Rightarrow x \uparrow, f(x) \uparrow$

magnitude of a $\Rightarrow$ how much it increases/decrease the fuction

$\Rightarrow$ how abrupt is the slope

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y_true)
```

$\Rightarrow$ how do we update W to make the loss value smaller?

loss_value = f(W)

we can compute partial derivatives with respect to some values

grad(loss_value, W0)[i, j] $\Rightarrow$ indicates the direction and magnitude of change when modifying W[i, j]

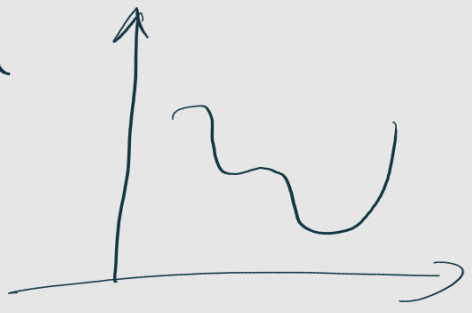
$\Rightarrow$ you move W in the opposite direction of the gradient

$W_1 = W_0 - step * grad(f(W_0), W_0)$

1) Draw a batch of training examples
2) Obtain predictions
3) compute the loss
4) Back propagation (compute the gradient of the loss with regards to the models parameters)
5) Move the parameters a little in the opposite direction from the gradient

True SDG → one sample
mini-batch → with a mini batch
Every step on all data ⇒ batch gradient descent

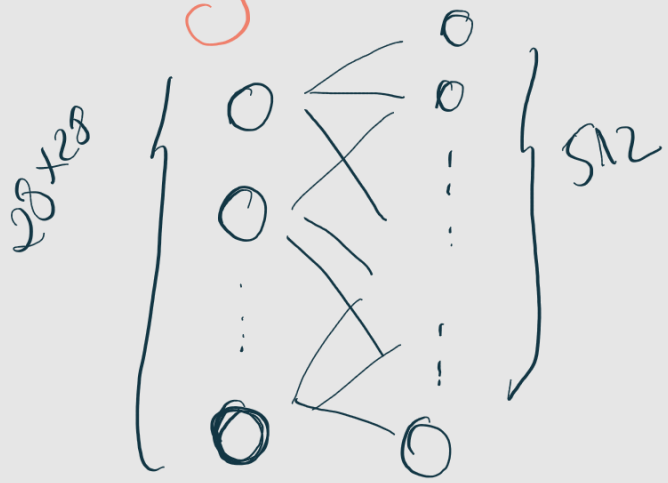
Momentum → convergence speed
→ Local Minima



The chain Rule

$$grad(y, x) = grad(y, x_3) * grad(x_3, x_2) * grad(x_2, x_1) * grad(x_1, x)$$

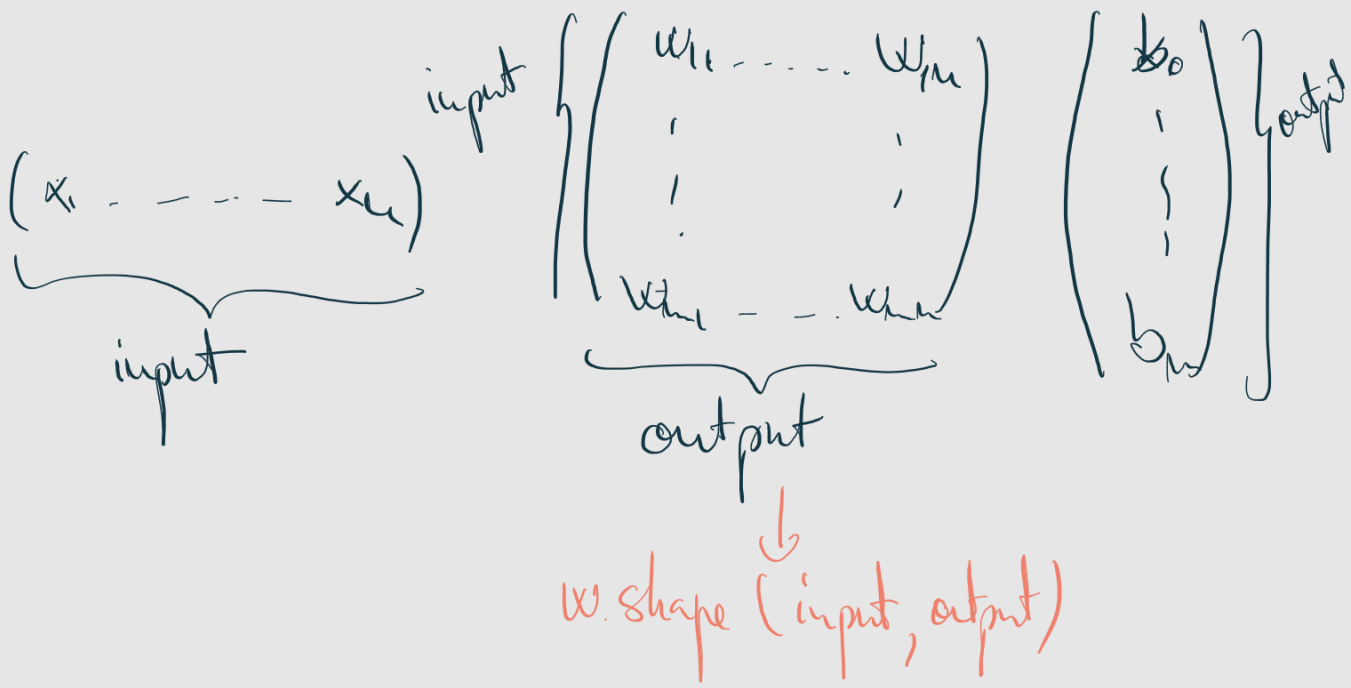
Reimplementing MNIST



Naive Dense
↳ 1 layer

output = activation (dot(W, input) + b)

W b



Chapter 3 : Introduction to Keras & Tensorflow

Tensorflow
- → dev by Google
- → automatically compute gradients of any differentiable equations
- → GPU & TPU
- → entire export to C++, JS or Tensorflow Lite for mobile or embedded

Keras → built on top of Tensorflow

History

March 2015 → Keras (on top of THEANOS)

Nov 2015 → TensorFlow

Early 2016 → Keras (Multi-backend)
- → THEANOS
- → TensorFlow

2017 →

Keras → CNTK (Microsoft)
→ MxNet (Amazon)
→ THEANOS
→ Tensorflow

You could choose which backend you want

To do anything ⇒ you need tensors
TensorFlow tensors are not assignable !

tf.Variable

TF is designed to retrieve the gradient of any differentiable expression

Understanding core Keras API

layer = fundamental building block

Single dense class → init
→ build
where it builds the weights matrix
→ call

This is because the weight matrix dimensions are computed at runtime for the code to be clearer.

model.compile → configures the training process with optimiser, loss ad metrics
you can also pass some custom functions

⇒ validation data

Chapter 4: Getting started with neural networks: classification and regression

In this chapter
- → binary classification
- → multiclass
- → scalar regression

- → Scalar regression: predicting a value (like a price)
- → Vector regression: predicting multiple values
- → mini-batch or batch: a small set of samples (usually between 8 and 128) simultaneously processed. Usually a power of 2 to facilitate memory allocation on GPU

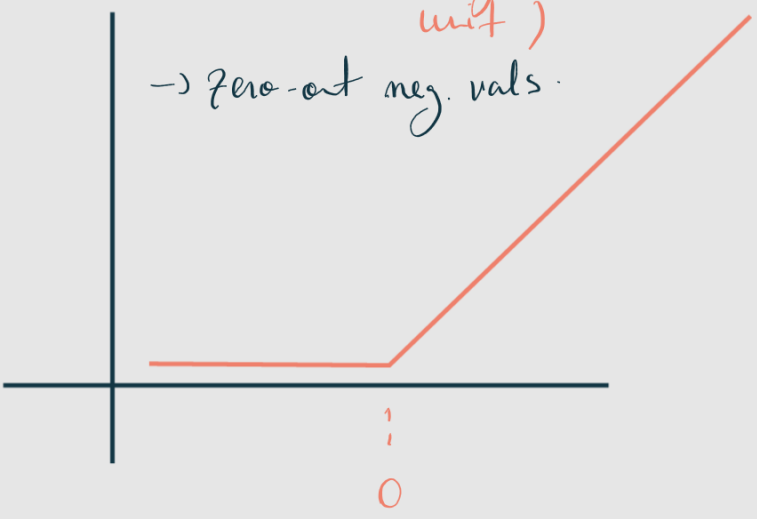
You can't directly feed lists of integers to a NN

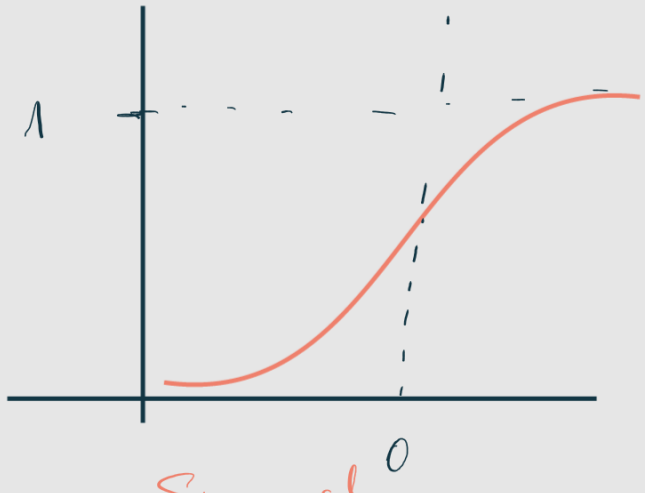
- → 1) Pad your lists so they all have the same length
- → 2) Multi-hot encode your list in a max dim vector ← 10K possibi-

[0, 1, 0, 0, 0, 1 .....]
10K

when creating a model through the layer size you define the representation space => how many dimensions??
higher dimensional => more complex representations

Relu (rectified linear unit)
-> zero-out neg. vals.





Sigmoid
=> squashes arbitrary values into [0, 1]

One-hot-encoding
<-> all 0s vector with one value of 1 in the place of the label index.

Categorical_crossentropy => measures the distance between 2 probability distributions

One-hot encoding -> categorical cross entropy
integer encoding -> sparse categorical cross entropy

Regression

training values in different ranges => normalisation
less values => smaller

=> smaller model in order to overcome overfitting.

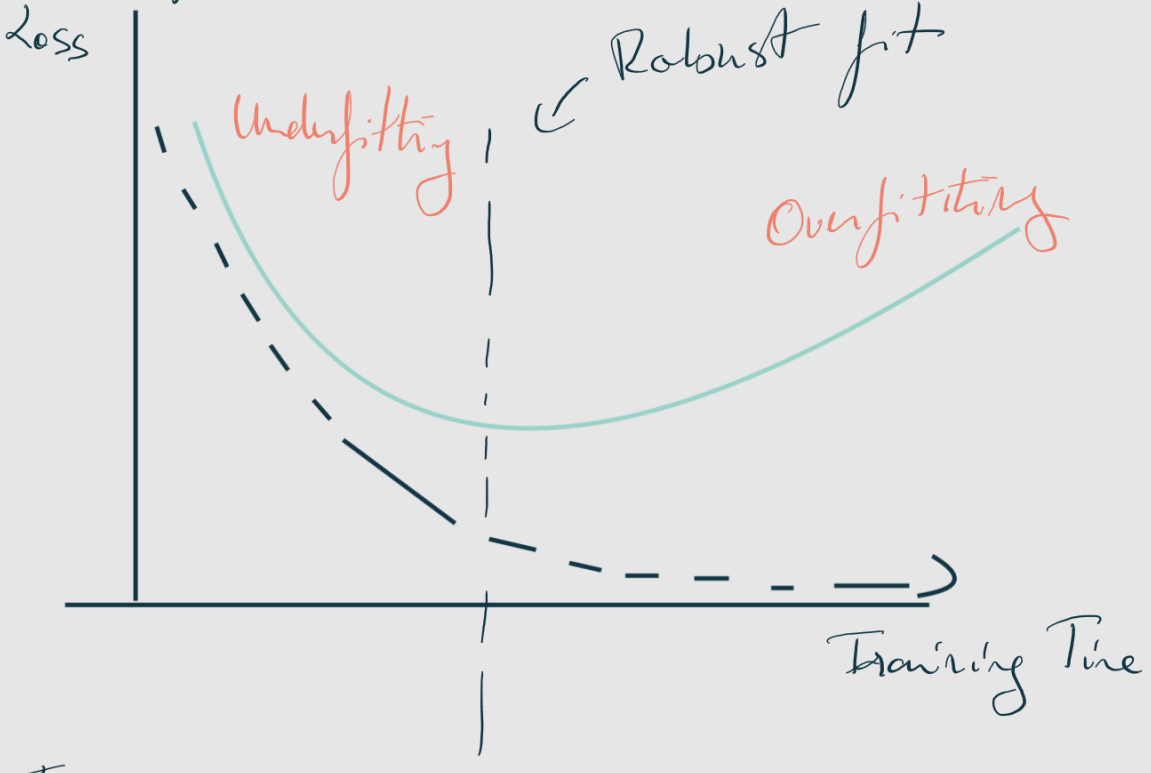
=> No activation on the last layer which is typicall for regression

mse => Mean Squared Error : Widely used for regression

Small dataset => use a k-fold cross validation
Split the available data into k partitions

Chapter 5: Fundamentals of Machine Learning

Optimisation vs Generalisation



There may be some weird training data and your models twists itself in order to improve the loss.
=> the generalisation will decrease

Models can be trained to fit and find correlations on any data, and still have a good accuracy

MNIST → $28 \times 28 \Rightarrow 256^{784}$ possibilities

but handwritten digits only occupy a tiny subspace of the huge parent space

⇒ the subspace is not random, it is highly structured

1) The subspace is continuous: take one sample and alter it, it is still a digit

⇒ all samples are connected by smooth paths that run through the subspace

⇒ The handwritten digits form a manifold within the space of the possible arrays

manifold = lower-dimensional subspace of a parent that is locally similar to a

linear (Euclidean) space.

Ex: a curve in the plane is a 1D manifold in 2D

Manifold hypothesis => all natural data lies on a low-dimensional manifold within the high dimensional space

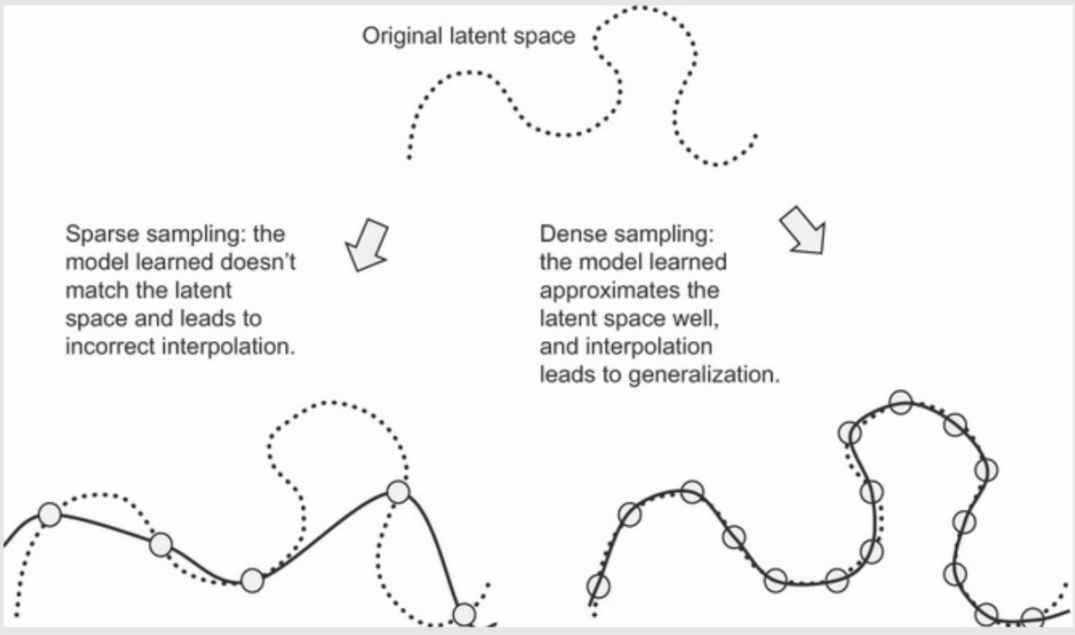
1) ML models have to fit relative simple low-dimensional, highly structured subspaces
2) within these manifolds its always possible to interpolate between two inputs => morph one into another via a continuous path.

Interpolation can help you make sense of things that are close to what you have seen before => local generalisation

ML -> smooth continuous mapping from input to output.
This smoothness helps approximate latent manifolds

generalisation -> more a consequence of the natural structure of your data; less noisy & more informative features
=> better generalisation

You need a dense sampling of the input space



When getting more data is not an option

modulate the quantity of info that your model is allowed to store

small nr. of patterns => force to focus on prominent patterns with a

Evaluating your model

better chance of generalizing well

- Training
- Validating
- Testing

Why not train & test ⇒ because you fine on test ⇒ this may give twisted results that perform artificially well on testing

1) Simple holdout validation
   set aside some validation
   - if little data is available the validation & test might be very few samples
2) K-fold validation
   Split in k partitions. Validate each i-th partition and than average
3) Iterated K-fold validation with shuffling
   Applying k fold multiple times after shuffling the data

Beating a common sense baseline.

↳ you need to start from a trivial way of creating a baseline: >50% for binary classification
>0.1 for MNIST
if the dataset is imbalanced and predicts always 1 it can have a 90% accuracy.

Having a common-sense baseline to refer to is essential when you are getting started on a problem.

Keep in mind

1) Data representativeness
(What labels and frequencies has the data both in test & training)
→ shuffle data

2) The arrow of time
→ don't shuffle if it is timeseries.

3) Redundancy in data
If data is duplicated you may test on training data and this will falsely improve the accuracy

Problems that can appear in the creation of a ML algo.

1) Training just doesn't get started : your loss is stuck.
=> This can always be overcomed (you can train on 0 sense data and you should still be able to train something)

Always a problem with the gradient descent
- optimiser
- initial weights
- learning rate
- batch size : bigger batches means less noise in the back prop. of the algo

2) You can't beat the base-line

The validation metrics don't improve. It stops somewhere near a random classifier

=> This indicates that something is fundamentaly wrong with your approach

For ex: dense architecture for timeseries => can't do nothing.

3) You can't overfit

Increase the capacity and the representation space of the model. => The model should learn quickly and than start to overfit.

Once your model has shown sme generalisation & overfitting you can start to focus on maximising generalisation.

Generalisation <= latent structure of data

1) More data
2) Minimize labeling errors
3) Clean data and deal with missing values
4) Do feature selection

Feature engineering = using your own knowledge about the data and the ML algo to make the algo work better by applying hardcoded (non-learned) transformations to the data before it goes into the model

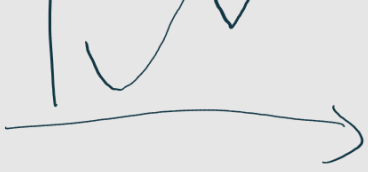
Regularisation techniques

1) Reducing the network size

a model that is to small will not overfit

test a model with multiple configurations

the model is too large if it overfits right



away and the validation loss curve looks choppy with high variance

2) Adding weight regularisation

Occam's razor => given 2 explanations, the explanation most likely to be correct is the simplest one, the one that makes fewer assumptions

Simpler models => less likely to overfit

= a model where the distribution of parameter values has less entropy (or a model with fewer parameters)

=> put constraints on the complexity of a model by forcing its weights to take only small values

= weight regularisation

↳ adding a cost associated with having large weights

- L1 regularisation = cost = absolute value of weights
- L2 regularisation = cost is the ^{2} of weights
- Adding a dropout -> one of the most effective and commonly used regularisations

= randomly dropping (setting to 0) some outputs from a layer.

dropout rate = the "forced out" rate

"At the bank they would randomly switch employees in order to avoid conspiracies. For the model not to learn from some conspiracies (false patterns) the dropout regularise the learning."