Machine learning workshop: introduction

Alex Richardson

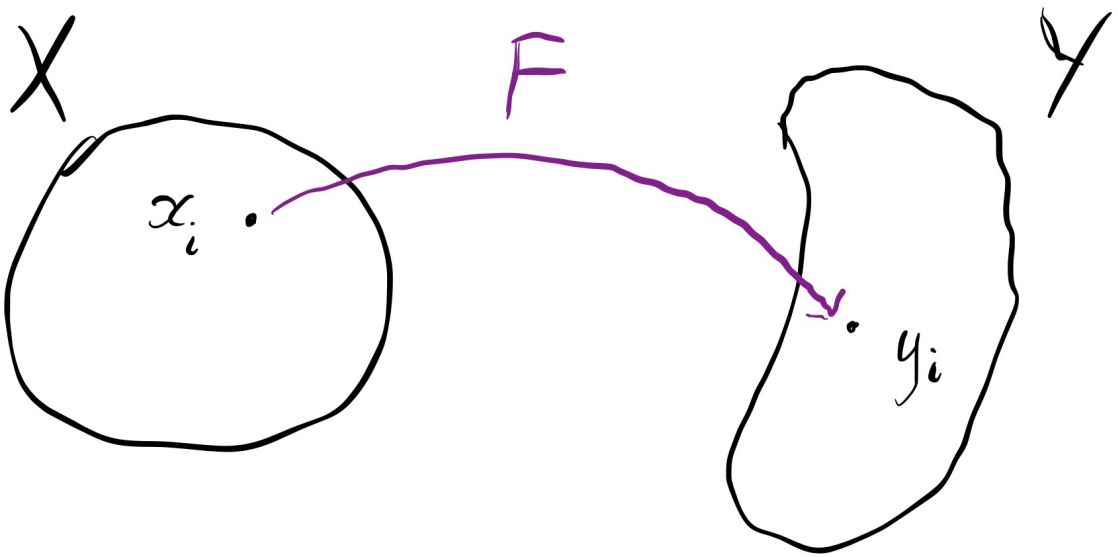
February 23, 2024

Structure

- What is machine learning anyway?
- Neural networks
- How to actually train models
- Python implementation
- First jupyter-notebook

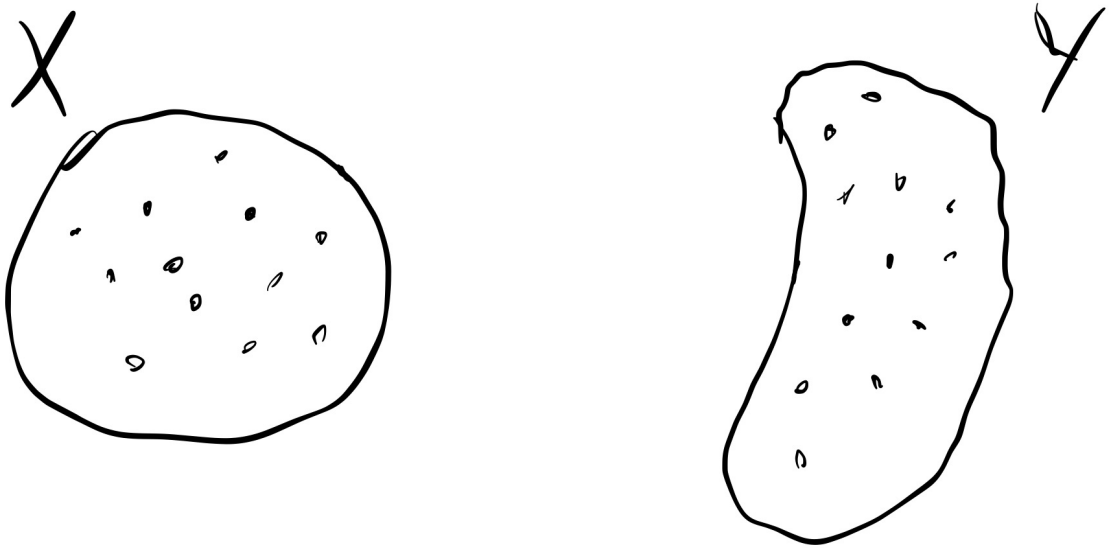
Parameterised function approximation

- Machine learning is just a set of techniques for finding **approximations** to a function that we don't know.
- Suppose we have some sets X and Y, we can assume there is some (unknown) function F such that:
  - $y_i = F(x_i)$ for all $x_i \in X$ and $y_i \in Y$



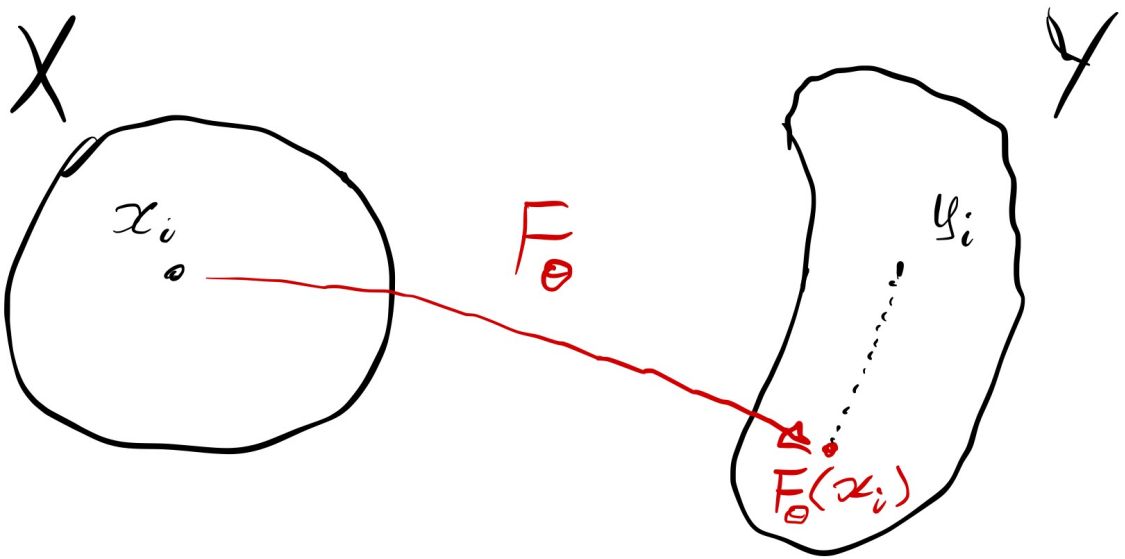
Parameterised function approximation

- In machine learning we typically have pairs of (x_i, y_i) (data), but F is unknown



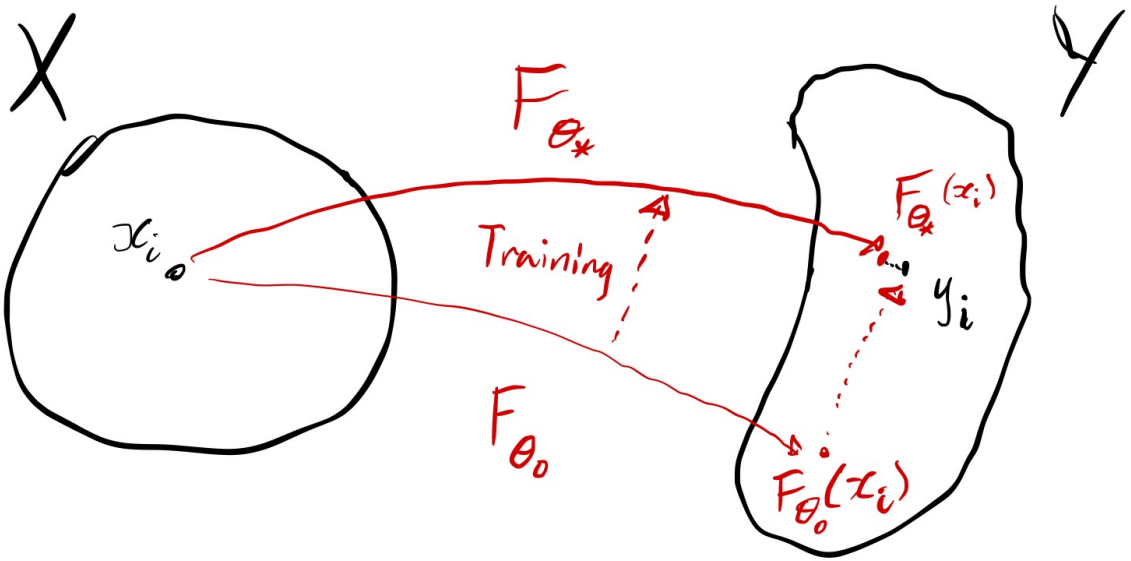
Parameterised function approximation

- If we **parameterise** F as F_θ (with parameters θ), we can tweak θ such that $F_\theta(x_i)$ approaches y_i
- We often refer to F_θ as the **model**



Parameterised function approximation

- Tweaking θ to fit $F_\theta(x_i)$ to y_i is called **training** the model F_θ

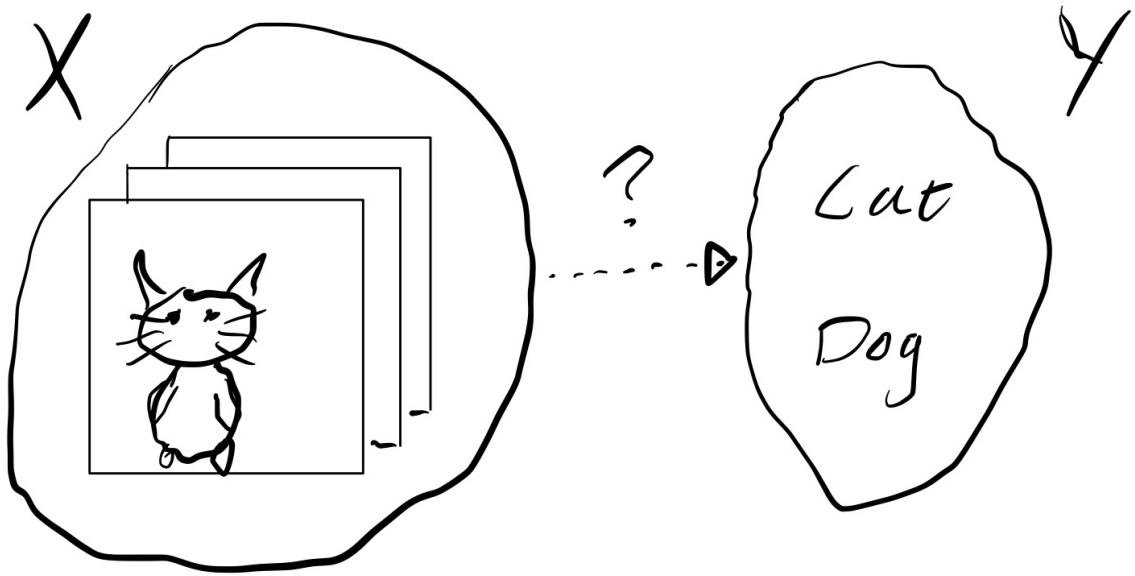


Making things less abstract...

- What can X look like?
  - Colour images (3 matrices $\mathbb{R}^{m \times m \times 3}$)
  - Sentences (how is this represented...?)
  - Time-series (of numbers, images...)
- What can Y look like?
  - Images
  - Text description
  - Categorical label
  - A number
- What does F look like?
  - Depends on what X and Y are...

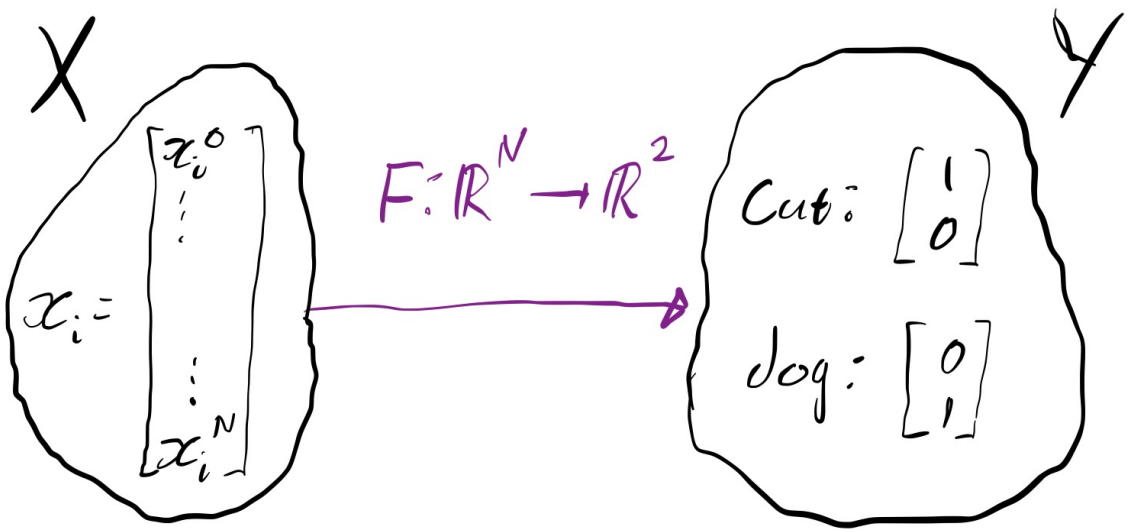
Example: classification

- Given pairs (x, y)
  - Suppose each x is a colour image of a cat or dog
  - And each y is a text label of 'cat' or 'dog'



Example: classification

- We need to represent our sets X and Y in **vector spaces**:
  - Each 512×512 colour (3 channel) image is already a vector: $x_i \in \mathbb{R}^{786432}$
  - For categorical variables, use **one hot encoding**: represent each category as a different unit basis vector



Example: classification

- A really simple (linear) model is a matrix multiplication
  - $F_\theta(x) = A_\theta x$
  - Where $A_\theta \in \mathbb{R}^{2 \times 786432}$
- We can find the entries of A_θ to get outputs of $F(x)$ as close to y as possible

Example: classification

- A simple linear model would work if the data (x, y) are linearly related
- However images of animals $\rightarrow$ text labels is highly nonlinear
- Depending on how **complex** we expect the relationship between data to be, we must parameterise F_θ appropriately.
- 2 important questions:
  - How do we choose a structure for F_θ?
  - How do we find good parameters for F_θ?

Neural networks

- Neural networks are a good choice of F_θ
- They are a very broad class of functions
- Importantly they are **nonlinear**
- Historically they are based on simplified models of neurons, but they have diverged quite far

Neural networks

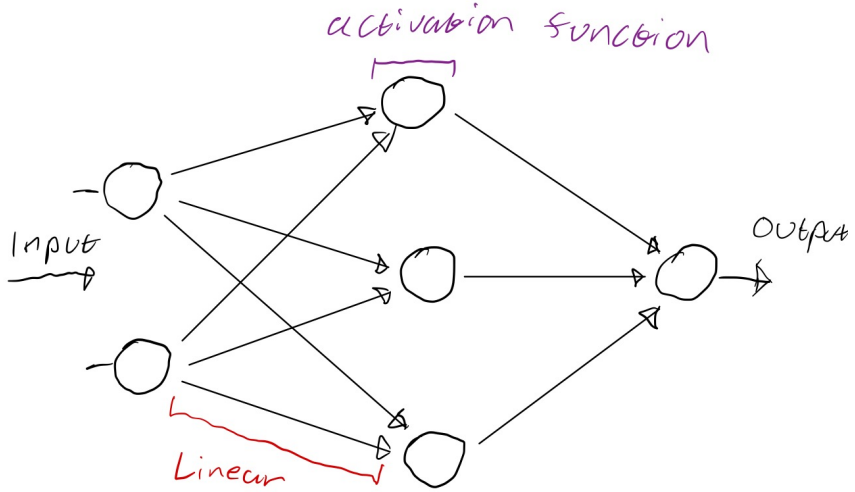
- (Feed-forward) Neural networks are typically of the form:

$$F(x) = \sigma_n \circ L_n \circ \sigma_{n-1} \circ L_{n-1} \circ \cdots \circ \sigma_0 \circ L_0(x)$$

- Where L_i are parameterised linear transformations
  - Matrix multiplications
  - Convolutions
- σ_i are **element-wise** nonlinear functions, referred to as **activation functions**
  - $\sigma(x) = \max(x, 0)$
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - $\sigma(x) = \tanh x$

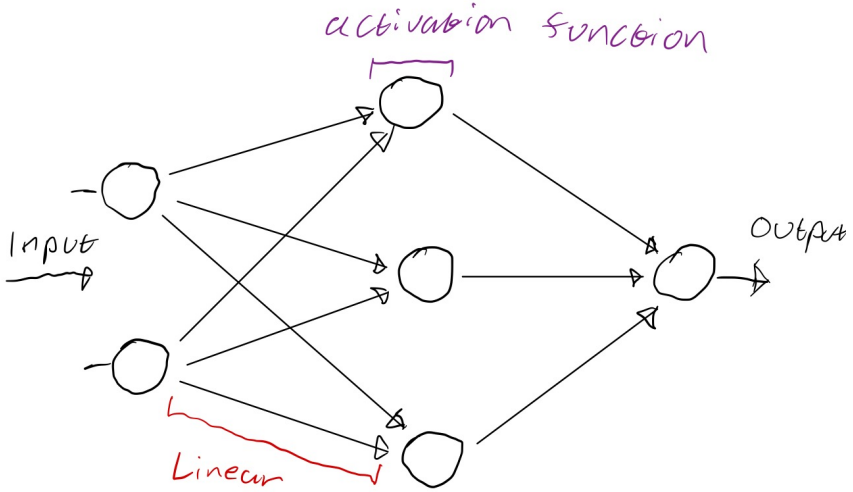
Neural Networks

- The L_i and σ_i are referred to as **layers**
- Feed-forward neural networks are ones where layers are connected simply in series



- Each node takes a weighted sum of it's left inputs, applies an activation function and feeds that value forward
- This network is **fully connected**, as every node in one layer connects to every node of the next

Neural Networks



- For the above diagram, the corresponding F_θ is:

$$F_\theta(x) = \sigma_1(A_1\sigma_0(A_0x))$$

- Where $A_0 \in \mathbb{R}^{3\times 2}$ and $A_1 \in \mathbb{R}^{1\times 3}$ are matrices of parameters

Neural networks

- In python code, neural networks typically look like:

```
        def F(x):
            for i in range(n):
                x = layers[i](x)
            return x
```

- Where `layers` is a list containing the various linear and nonlinear functions

Universal Approximation Theorem

- **Neural networks can approximate any continuous function**
- If we have a set of neural networks $\mathcal{F}$ of the form $F(x) = A\sigma(Bx + c)$, and the input space X is a compact/closed subset of $\mathbb{R}^N$, then:
- For **every** continuous function $g : X \to Y$, and any $\varepsilon > 0$, there exists a neural network F that is arbitrarily close to g: $|F(x) - g(x)| < \varepsilon$
  - Also required: sufficiently big hidden layer (B), and correct type of activation function
- The proof requires some functional analysis, so we'll skip it

Neural networks

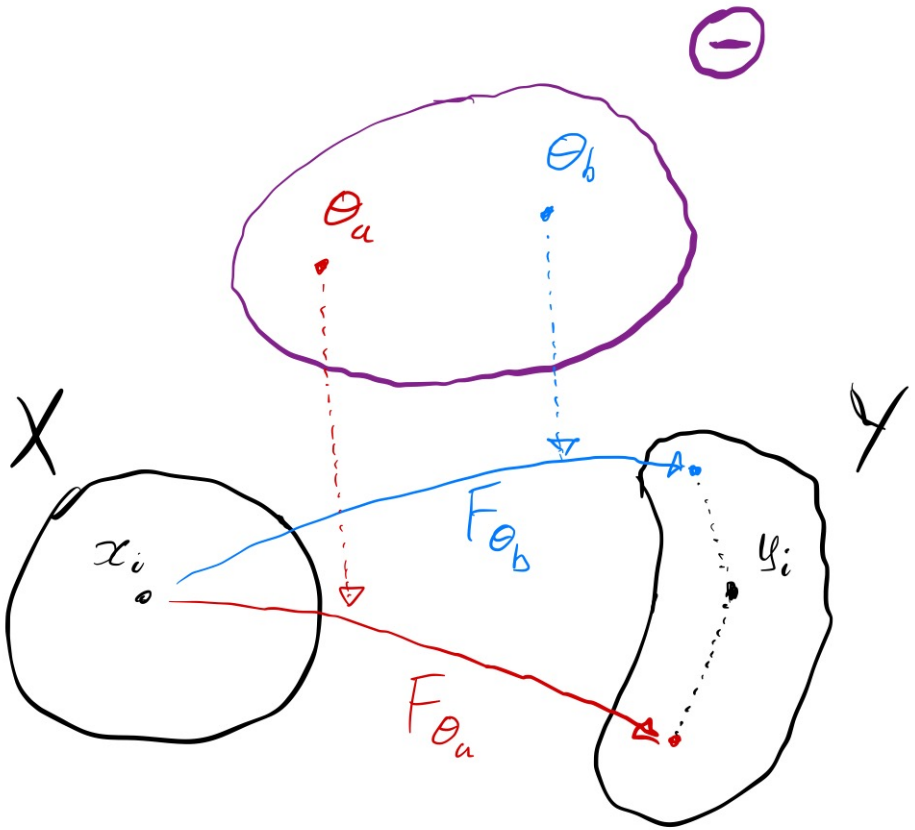
- In theory fully connected feed forward neural networks can approximate any function
- In practice they don't work that well
- Why?
  - Getting good performance on interesting data requires a lot of parameters
  - By exploiting symmetries or structures of the data we already know, we can build models with less parameters that perform just as well
  - Crucially: models with less parameters are easier to train
- We will look at fancier model structures later, but first how do we train models?

How to train models

- The basic idea of training models is easy
  - Begin by randomly choosing parameters (θ_0)
  - Repeatedly tweak the parameters in such a way that $F_\theta(x_i)$ gets closer to y_i
- There are 2 clear issues:
  - How do we measure 'closeness' (or distance) between $F_\theta(x_i)$ and y_i?
  - How do we tweak θ in a way that reduces the distance between $F_\theta(x_i)$ and y_i?

How to train models

- It is worth thinking of the space of all parameters a model can have: $\theta \in \Theta$



Loss functions

- **Loss functions** are functions that measure some meaningful sense of distance between $F_\theta(x_i)$ and y_i
- If $F_\theta(x_i), y_i \in \mathbb{R}^N$

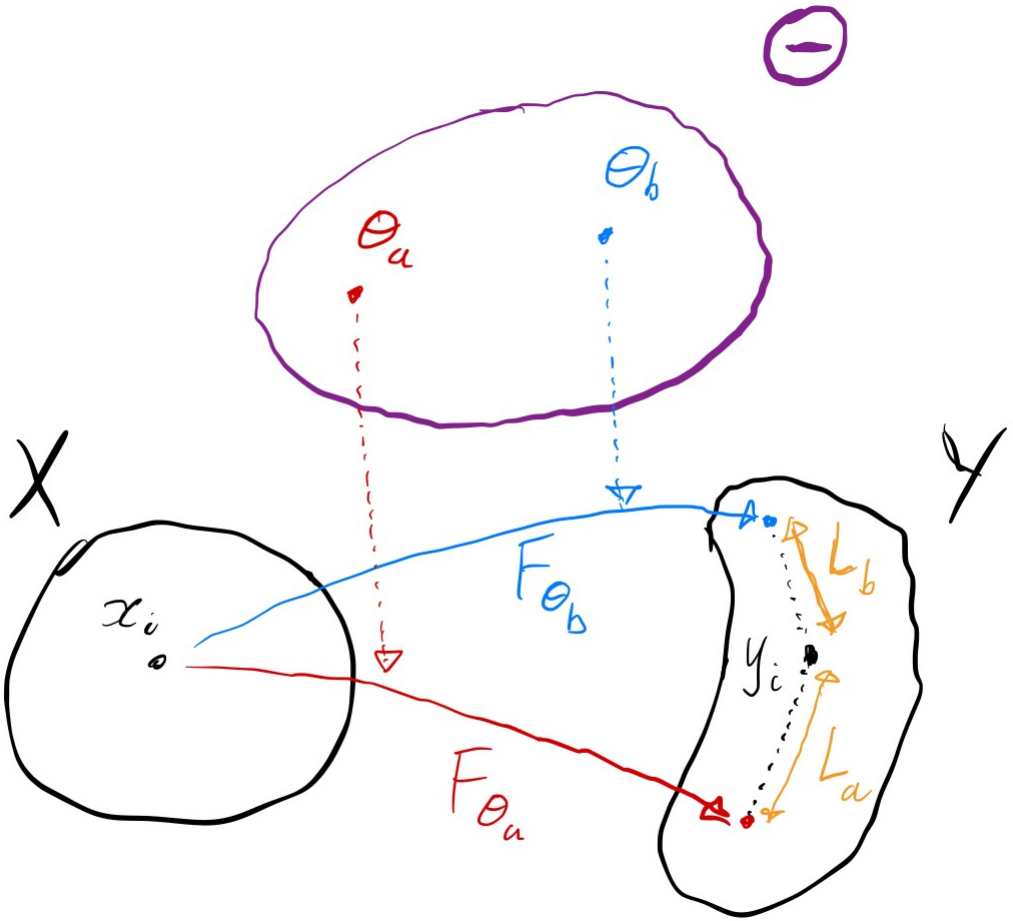
$$L : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$$

- A good choice of loss function is a vector space norm, e.g. the euclidean norm:

$$L_i(F_\theta(x_i), y_i) = \sqrt{(F_\theta(x_i) - y_i) \cdot (F_\theta(x_i) - y_i)}$$

- L_i will be zero iff $F_\theta(x_i) = y_i$

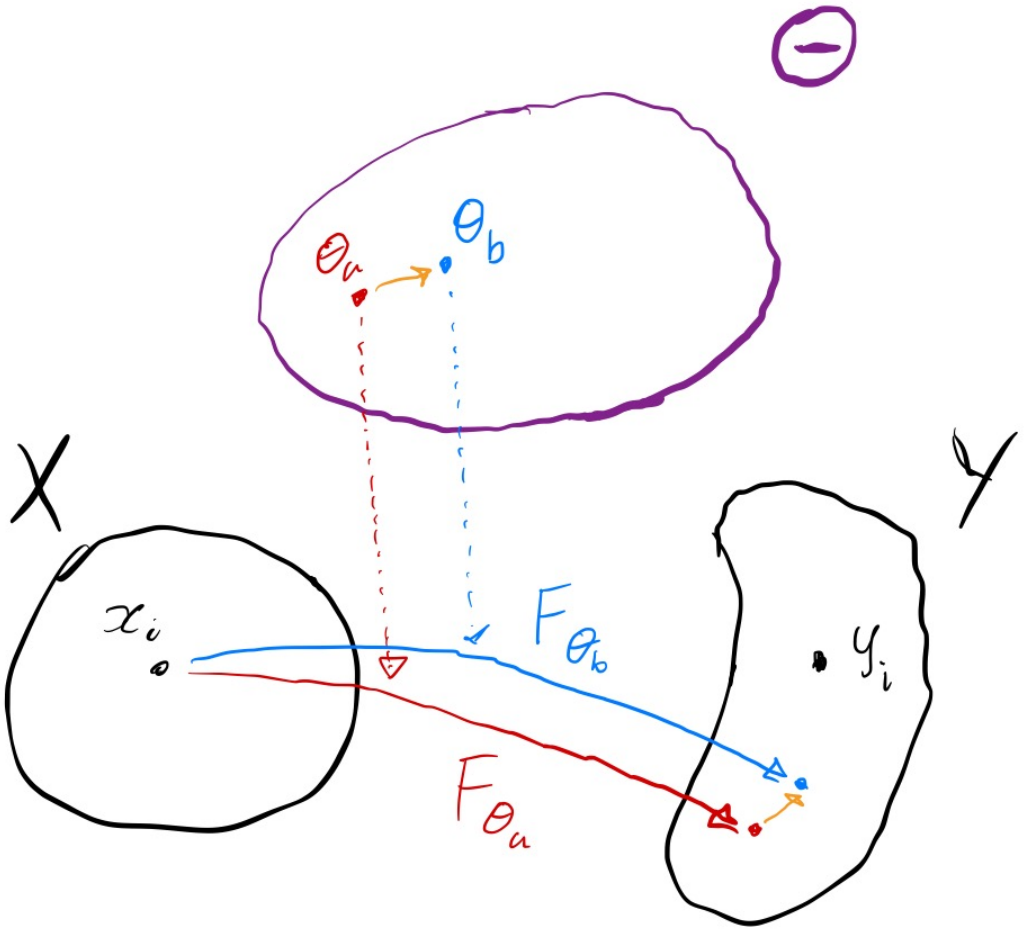
Loss functions



Loss functions

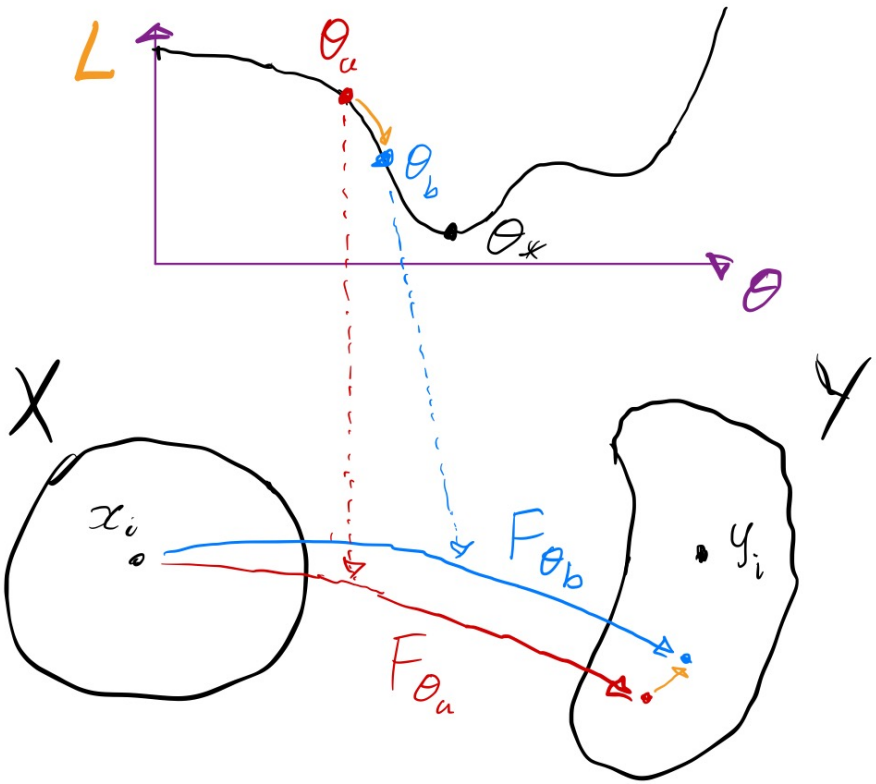
- With a loss function defined, the training process is now tweaking θ to minimise $L_i(F_\theta(x_i), y_i)$
  - We actually want to minimise $\mathcal{L} = \sum_i L_i(F_\theta(x_i), y_i)$
- To efficiently minimise $\mathcal{L}$, we would like $\mathcal{L}$ to be **differentiable** with respect to θ
  - This means F_θ depends continuously on θ
  - We must also choose $\mathcal{L}$ carefully

F must be continuous in θ



Gradient descent

- By moving in the direction of steepest gradient of $\mathcal{L}$ with respect to θ, we get a new θ with a smaller loss:



Optimisers

- A simple training loop would look like:
  - Randomly pick some initial values of θ
  - Compute the gradient of $\mathcal{L}$
  - Update θ based on this gradient
- A simple gradient descent method is:

$$\theta_{i+1} = \theta_i - \varepsilon \nabla_\theta \mathcal{L}$$

- Where ε is a small number referred to as the **learning rate**
- There are many more sophisticated ways of doing this...

But how does this actually work...

- Using **Automatic Differentiation**, gradients of the code which defines your loss function and model can be computed
  - This is where the widely used **backpropagation** algorithm comes in
- Typically computing these gradients is as quick as computing the function itself!

Summary

- Machine learning is a set of techniques that allow us to:
  - Construct general parameterised functions called **models**
  - **Train** (or **learn**) these models to a dataset of inputs and outputs
- Important details to remember:
  - The model must be **differentiable** with respect to its parameters
  - We must represent the dataset in some **vector space**
  - We need a meaningful **loss function** to measure how good our model is

What python libraries?

- There are several popular python libraries for machine learning. Some examples are:
  - Pytorch
  - Jax
  - Tensorflow
  - Scikit learn
  - Keras
- We will explore using Jax, because it is my current favourite

Jax

- Jax is **not** a machine learning library
- It is instead just a re-implementation of Numpy, with some very nice extra features:
  - Everything is differentiable
  - Everything automatically parallelises to your hardware (i.e. GPUs) using the XLA compiler
  - There's a decent Just In Time (JIT) compiler, that removes the issue of python being slow
  - There are good methods for explicitly vectorising code, even vectorising across specific hardware

Jax

- There are several smaller libraries building on Jax, and (mostly) they are all mutually compatible:
  - Equinox: neural network library
  - Flax: another neural network library
  - Optax: gradient based optimisers for training anything
  - RLax: reinforcement learning
  - jax-md: molecular dynamics
  - BRAX: differentiable physics simulator for robotics
  - Diffrax: differentiable numerical differential equation solvers

Jax fundamentals

- If you replace `import numpy as np` with `import jax.numpy as np`, your code will probably still work[1]
- There are three important functions in Jax:
  - `grad`
  - `jit`
  - `vmap`

[1] except updating individual array entries requires new syntax, and random number generation is different

grad

```
1          def f(x,y):
2              return (x-y)**2
3          df = jax.grad(f) # defines a new function
4          print(df(5.0,2.0)) # returns 6.0
5
```

- Here `df` is a new function that returns the gradient of `f` **with respect to it's first argument** x
- Important: `f` must be a pure function (i.e. no changing of global variables)
  - Because of this, random numbers are implemented a bit differently in Jax...

jit

```
        def f(x,y):
            return (x-y)**2
        f_jit = jax.jit(f) # defines a new function
        for i in range(1000):
            a += f_jit(i,2.0)

```

- Here `f_jit` will be slightly slower the first time it's called, but much faster on subsequent calls
- This is especially noticeable if `f` is complicated or contains a lot of loops

vmap

```
1        def f(x,A):
2            return A@x
3        vf = jax.vmap(f,in_axes=(0,None),out_axes=0)
4
```

- Here we have explicitly vectorised the `x` input, and output of `f`, but have kept `A` the same
  - `f`: $\mathbb{R}^n \times \mathbb{R}^{m \times n} \to \mathbb{R}^m$
  - `vf`: $\mathbb{R}^{b \times n} \times \mathbb{R}^{m \times n} \to \mathbb{R}^{b \times m}$
- This is a lot more flexible and precise than the numpy vectorize method
- vmapped code will parallelise automatically, and generally run faster than iterating through a loop

Combined together...

```
        def f(x,A):
            return np.sum(A@x)
        df = jax.grad(jax.grad(f))
        vdf= jax.vmap(df,in_axes=(0,None),out_axes=0)
        vdf= jax.jit(vdf)
```

- Now `vdf` is a JIT compiled function that returns an array of second derivatives of `f` with respect to rows of a matrix `x`

Notebook 1

- Work through the first jupyter-notebook
  - It contains a very minimal implementation of a fully connected feed forward neural network, a loss function and a gradient optimiser, in pure Jax
  - We train it to classify images from the MNIST hand-written digit dataset
  - Run the code and try to understand it:
    - Modify it and see what breaks it
    - The network doesn't perform that well - can you try to improve the code?

Notebook 2

- This notebook is the same as the previous one, except we make use of the Equinox and Optax libraries for defining our network and optimiser
- Go through this notebook
  - Notice how much cleaner the code is
  - The trained network appears to perform better - why?