EngSci 205: Engineering-Centric Machine Learning

Oliver Maclaren (oliver.maclaren@auckland.ac.nz)

Preface

The following notes were prepared by me (Oliver Maclaren) and Ruanui Nicholson for ENGSCI 205: Engineering-Centric Machine Learning, based on discussions with Piaras Kelly, Justin Fernandez, Cameron Walker, Andreas Kempa-Liehr, Ivan Koptev, and others.

They assume some basic familiarity with fitting simple statistical machine learning models using e.g. scikit-learn. The focus is on high-level ideas about *what* we put into these models, as well as on developing practical familiarity working with multi-dimensional arrays called *tensors* in PyTorch in order to construct basic linear and nonlinear (neural) models for oneself. After initial introductory material, we focus on incorporate engineering or scientific insights via penalty or *regularisation* terms in linear and nonlinear models.

The plan is roughly

- Week 1: Key ideas
- Week 2: Vectors, matrices, and tensors in PyTorch
- Week 3: Making linear data models yourself
- Week 4: Incorporating scientific knowledge in penalty terms
- Week 5: Making nonlinear data models (neural networks)
- Week 6: Incorporating scientific knowledge in neural networks

Part I

Key ideas

Introduction

What is **machine learning**? What is **engineering**? How are they related? These questions are too difficult to answer properly but we can give some brief ideas. You will have seen some of these ideas already in the first part of the course, but we recap them here as they will inform our approach in what follows.

The first key ingredient in machine learning is **data**. This can be in the form of **numerical measurements** but could also be **text**, **images**, **videos** and so on. Of course scientists, engineers, and statisticians have been using data long before 'machine learning' was a thing (although the term 'machine learning' and related terms like artificial intelligence and machine intelligence date back to around the 1950s). Sometimes people say machine learning is nothing but statistics! So, what distinguishes machine learning from other fields that use data? One of the key aspects of machine learning is a focus on **prediction** or, more generally, **performance on a well-defined task in some specific domain**, **without** necessarily being concerned with **understanding the domain** in the traditional 'scientific' sense. Consider for example the following quote from the 2008 Wired article on 'The End of Theory' (emphasis added):

> Google's founding philosophy is that we don't know why this page is better than that one: If the statistics of incoming links say it is, that's good enough. No semantic or causal analysis is required. That's why Google can **translate languages without actually 'knowing' them** (given equal corpus data, Google can translate Klingon into Farsi as easily as it can translate French into German). And why it can **match ads to content without any knowledge** or assumptions about the ads or the content.

Note the above piece highlights the performance of a **clearly-defined task** in a specific domain – translate given text from one language to another, choose an ad to show when someone visits a website – **without explicit 'knowledge'** of the broader domain (language, advertising psychology). The author of the above piece makes a **controverisal comparison** to the scientific approach:

> The **scientific method** is built around **testable hypotheses**. These **models**, for the most part, are systems visualized in the **minds of scientists**. The models are then **tested**, and experiments **confirm or falsify theoretical models of how the world works**. This is the way science has worked for hundreds of years.
>
> Scientists are trained to recognize that **correlation is not causation**, that no conclusions should be drawn simply on the basis of correlation between X and Y (it could just be a coincidence). Instead, **you must understand the underlying mechanisms that connect the two**. Once you have a model, you can connect the data sets with confidence. **Data without a model is just noise**.
>
> But faced with **massive data**, this approach to science — hypothesize, model, test — is becoming **obsolete** ...
>
> There is now a better way. [lots of data means we can] say: **"Correlation is enough." We can stop looking for models**. We can analyze the data without hypotheses about what it might show. We can throw the numbers into the biggest computing clusters the world has ever seen and **let statistical algorithms find patterns** where science cannot."

These are strong claims! **How justified are these?** And where do engineers fit in?

Is data enough?

The strong claim above that 'data are all you need' is provocative but oversold. One way to illustrate this is to distinguish between **passive** observation and prediction of a system under given conditions and **intervention** on a system that changes the conditions.

Example: suppose that you observe the weather every day and record whether it rains or not. You also place a bucket outside and record how full the bucket gets. Clearly there will be a 'statistical' or 'correlational' pattern/relationship between whether the bucket has water in it and whether it has rained. This means, for example, that if you forgot to look outside during the day but still collected the bucket at the end of the day you could successfully 'predict' (or 'retrodict') whether it had rained or not. However, suppose then that you wake up and want to go outside for a picnic. If you take the statistical pattern too seriously and have no notion of cause and effect *you might think that to prevent it raining you should make sure the bucket doesn't get full – So you simply cover the bucket and head out for your picnic!*

Now, the machine learner may reply that the problem is we haven't actually collected data on what happens when the bucket is artifically covered. But this is part of the point – one of the reasons we develop scientific theories and mechanistic understanding is to **generalise** to new contexts and scenarios: our bucket – rain relationship is stable within a limited domain but the laws of physics (presumably) hold throughout the entire universe. Our understanding of cause and effect, accumulated throughout human history, allows us to say that rain causes the bucket to fill but the bucket filling does not cause the rain, without needing to collect additional data.

Despite this warning, the machine learner raises a weaker but still interesting point: in many real-world contexts we are interested in **the environment is stable enough, the available data is representative enough, and the performance goals well-defined enough that we don't necessarily need a deep understanding** of all the mechanistic processes and causes and effects involved in order to carry out a task well.

Aside: Another example

If the previous example was not convincing, consider the following example from the current (2024) version of ChatGPT. ChatGPT has been trained on many, many sources of information including, presumably, this version of a well-known puzzle...

A man has to get a fox, a chicken, and a sack of corn across a river. He has a rowboat, and it can only carry him and one other thing. If the fox and the chicken are left together, the fox will eat the chicken. If the chicken and the corn are left together, the chicken will eat the corn.

How does the man do it?

To solve this classic river crossing puzzle, the man can follow these steps:

1. **Take the chicken across the river** and leave it on the other side.
2. **Go back alone** to the original side.
3. **Take the fox across the river** to the other side.
4. **Bring the chicken back** to the original side.
5. **Take the corn across the river** to the other side.
6. **Go back alone** to the original side.
7. **Take the chicken across the river** again.

Very impressive! Now, consider a much simpler puzzle ...

Aside: Another example continued

A farmer needs to cross a river with two chickens. He has a boat with room for one human and two animals. How can he cross with his chickens using the smallest number of trips?

The farmer can cross the river with his two chickens in the smallest number of trips using the following sequence:

1. The farmer takes both chickens across the river (1 trip).
2. The farmer leaves the chickens on the far side and returns alone (1 trip).
3. The farmer then crosses the river again with both chickens (1 trip).

In total, the farmer makes 3 trips to get himself and the two chickens across the river.

A terrible failure! Here the lack of 'understanding' of the domain has become apparent and ChatGPT fails to generalise to a new scenario, even though it is a much simpler scenario!

Basic setting

In the standard setting of **supervised** machine learning, we have access to **training data**, consisting of a series of **examples**:

$$(x, y)_1, (x, y)_2, \dots, (x, y)_m,$$

i.e.

$$(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m),$$

composed of **features** (or inputs/covariates) x_i, $i = 1, \dots, m$ and **targets** (or outcomes/repsonses/labels) y_i, $i = 1, \dots, m$. Both the x_i and y_i may be numbers, vectors, functions, images, etc.

The goal is then, e.g., to **predict** the target y_j for a new set of features x_j, coming from an unseen or future **test dataset** or test scenario.

In machine learning, as opposed to statistics or scientific modelling, the aim is typically to make as **minimal assumptions as possible on the underlying mechanisms** generating the data. However, many standard methods and theoretical frameworks implicitly or explicitly assume that the data is independent and identically distributed (or at least 'exchangeable'). This is actually a fairly strong assumption, especially for engineering applications!

More generally, the basic conceptual assumption is that **the training data is representative of the test data** in some sense. Making this precise, without just assuming iid data, is a difficult and often ignored task! However, in scientific and engineering applications, ensuring the training data is representative of the scenarios in which the model will be used is often even more important than usual.

Let's briefly discuss some basic assumptions on how the data is generated.

Basic types of assumptions

A common assumption is that there is an underlying **function** f that maps features to targets:

$$y = f(x).$$

More generally we might have a **noisy** mapping such as

$$y = f(x) + e$$

or, more generally,

$$y = f(x, e).$$

where e is an (unobserved) error term.

Even this can be an approximation though! For example, if x represents **time** and y represents the **value** of some dynamically varying quantity, the above model assumes **deterministic dynamics** with only observation error.

Such deterministic models are common in engineering applications (e.g. systems governed by ordinary or partial differential equations), but more generally we may wish to use a **stochastic process** model, in which the dynamics themselves are stochastic.

Stochastic process models

For models in which the dynamics themselves are stochastic or noisy, the models are typically represented (assuming discrete time!) in forms such as

$$y_t = f(y_{t-1}, y_{t-2}, y_{t-3}, \dots, e_t)$$

i.e. the **present (or future) is a function of the past (or past up until the present) and some error term**. This is often called an **autoregressive** model in statistics and machine learning.

This form is itself **inspired by scientific modelling**: years of experience have taught us that such models are useful ways to capture ideas of 'cause and effect' in dynamic systems. E.g. Newton's laws can be written, once discretised, as a noise-free version of the above.

We can massage this back into our (x, y) format by taking x to be the **'history'** $\mathcal{H}_{t-1}$ of y up to $t-1$, i.e. the **past (or lagged) values** of y:

$$(x, y)_t = (\mathcal{H}_{t-1}, y_t) = ((y_{t-1}, y_{t-2}, ...), y_t).$$

Now we again have a model in the standard form

$$y_t \approx f(x_t) = f(\mathcal{H}_{t-1})$$

where the approximation may be represented by additive or non-additive error terms giving stochastic dynamics.

What if we want to repeat the procedure to predict y_{t+1}? We need to update the history to include both the new y_t and the past history! This can be thought of as a model of **history updating**:

$$\begin{aligned}\mathcal{H}_t = (y_t, \mathcal{H}_{t-1}) &= (f(\mathcal{H}_{t-1}), \mathcal{H}_{t-1}) \\ &= g(\mathcal{H}_{t-1}).\end{aligned}$$

State space models

The 'history updating' idea leads naturally to the 'systems dynamics' concept of **system state**: we define the concept of the state of a dynamic system as the 'relevant history' required for predicting the next y_t value. We can define the state vector at time $t-1$ as

$$z_{t-1} = \mathcal{H}_{t-n:t-1} = (y_{t-1}, \dots, y_{t-n})$$

for $n \geq 1$, and give the **state update** or **process model** equation as

$$z_t = g(z_{t-1}) = (f(\mathcal{H}_{t-n:t-1}), \mathcal{H}_{t-n+1:t-1}),$$

where $\mathcal{H}_{t_1:t_2} = \emptyset$ if $t_1 > t_2$. This represents a setting where we only need to remember up to a finite number of past values in order to predict the future, and will hence forget the rest. This is a type of **Markovian** assumption.

Higher-order differential equations can always be converted to **state space** or first order form (see MM2/3/4?); for example **Newton's second law** is a **second-order** (in time) differential equation that becomes a **system of two first order** differential equations for position and momentum when in state space form.

More generally, observation and process noise can be represented in terms of a **state space** or **hidden state** model such as:

$$\begin{aligned}y_t &= f(z_t) + e_t \\ z_t &= g(z_{t-1}, \epsilon_t)\end{aligned}$$

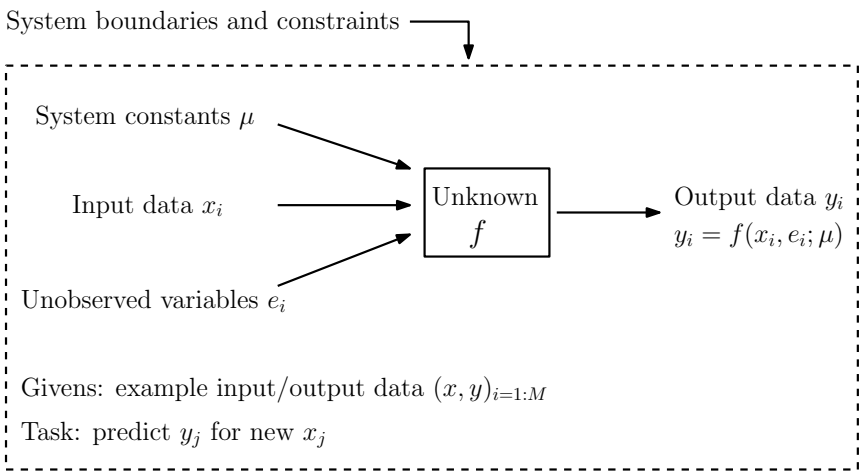
where e_t is (additive) observation error, ϵ_t is (possibly non-additive) 'process' error and typically only (t, y) i.e. y_t is observed. The 'state space evolution' describes the dependence of the present on the past and is unobserved. The observation equation describes instantaneous but noisy or indirect observations of the current state.

A deterministic model with observation noise can also be thought of as a special case of the above where there is no process error.

Stepping back: conceptual picture

As we have seen there are many types of dependence assumptions we can make. In general though, we are trying to apply our system in a relatively **stable environment** that we can imagine is characterised in an engineering sense by a **system boundary** and fixed **system constants** denoted here by μ (note: these could be e.g. fixed mean, variance of a random environment!).

This gives a rough conceptual picture as below:



where, as discussed, we may need to carefully define our x, y, e, μ etc variables depending on what sort of minimal assumptions we are willing to make.

Enough generalities! Let's consider a **specific physical system** and how the above ideas can help us use machine learning to approximate and predict it.

The nonlinear pendulum

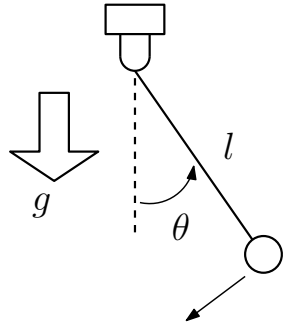
Many problems in engineering and science are governed by differential equations. For example, the angular motion $\theta(t)$ of a pendulum with length l subject to Newton's laws and no friction or air resistance, is given by

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = 0$$

with initial conditions

$$\theta(0) = \theta_0, \quad \frac{d\theta}{dt}(0) = v_0$$

A schematic is shown below.



Derivation: in class! See also MM2 or Greenberg (1998) 'Advanced Engineering Mathematics'.

Note that this is a **nonlinear**, **second-order** ordinary differential equation in **time**. The solution, $\theta(t)$, is a **deterministic** function of time that cannot be given in terms of so-called elementary functions, though it can be solved in terms of elliptic integrals. More complex models, e.g. including forcing or damping, can generate chaotic solutions, and must be solved by numerical simulation.

The linear pendulum

We can also make the **small angle approximation** $\sin\theta \approx \theta$ to obtain the **linear pendulum** model:

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\theta = 0$$

with initial conditions

$$\theta(0) = \theta_0, \quad \frac{d\theta}{dt} = v_0$$

This is now a **linear**, **second-order** ordinary differential equation in **time**. The solution, $\theta(t)$, is again a **deterministic** function of time.

This has an explicit solution:

$$\theta(t) = \theta_{\max}\cos\left(\sqrt{\frac{g}{l}}t + \phi\right)$$

where ϕ is a phase shift term and $\theta_{\max}$ is the maximum angular displacement, and both these terms depend on the initial conditions. If the initial velocity is zero, this simplifies to

$$\theta(t) = \theta_0\cos\left(\sqrt{\frac{g}{l}}t\right)$$

i.e. $\theta_{\max} = \theta_0$ and $\phi = 0$.

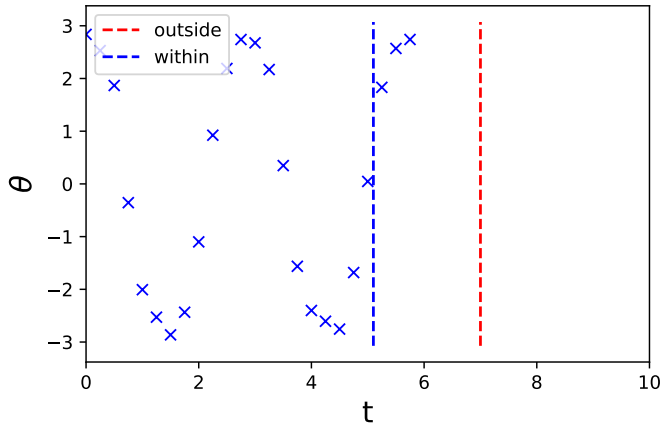
We will assume zero initial velocity for both the linear and nonlinear pendulumns from now on for simplicity.

Predicting the pendulum

Now, suppose we have M, (possibly noisy) observations of the form

$$(t_i, \theta_i),\ i = 1, \ldots, M$$

from a pendulum (possibly nonlinear!) and wish to predict θ_j for a new, given t_j. We consider two cases of t_j, one that lies 'within' the range of observed data, and one that lies 'outside' the observed range. These corresponds to **interpolation** and **extrapolation** tasks, respectively.



What can we do? The classical scientific/engineering approach is to **directly use our knowledge of the pendulum equations**. For example, if we know g and l, as well as the initial conditions, we can simply **solve or simulate** our model to obtain $\theta(t; g, l, \theta_0, v_0) = \theta(t; \mu)$, where $\mu = (g, l, \theta_0, v_0)$ represents our system parameters (assumed constant here).

If we didn't know some of our parameters, the classical approach is then extended to first carry out **parameter estimation** for the parameters given data and then proceed as above with these estimates. This sort of approach is covered in other courses. What does a 'machine learner' do instead?

Predicting the pendulum: empirical approach

Here, instead, we want to explore the possibility of **working as directly with the data as possible**, and see what minimal additional assumptions we can make in order to carry out the **prediction task**. This can be thought of as **empirical modelling**, rather than scientific or engineering modelling.

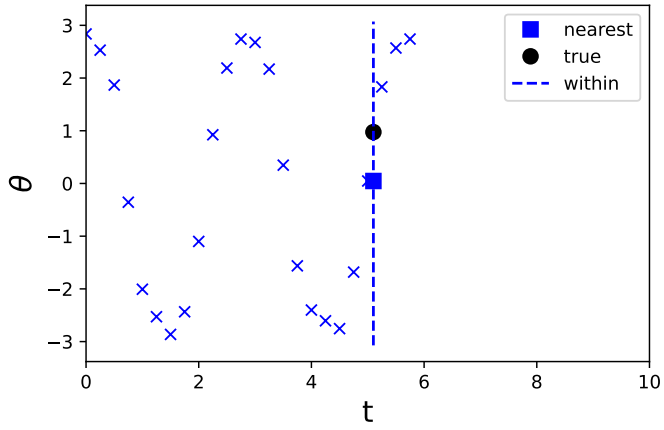
Here are three simple approaches to predicting at the 'within domain' time point, i.e. the **interpolation** problem:

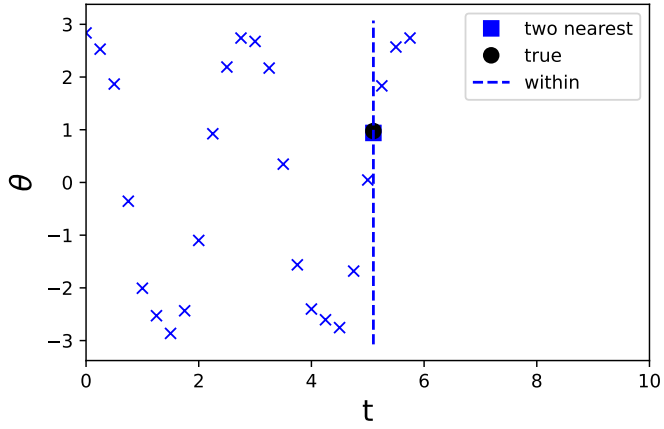
- Find the time point closest to the desired prediction point and use the associated θ value.
- More generally, take the average of values associated with the K closest time points (K-Nearest Neighbours!).
- Fit a machine learning/statistical regression function to the given training points and predict at the new time point using this.

How well does this work? Let's see!

Predicting the pendulum: interpolation problem

Simple nearest neighbour approaches:

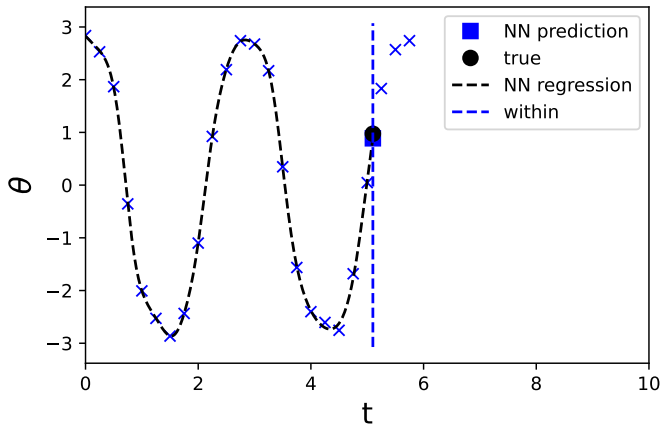




Predicting the pendulum: interpolation problem

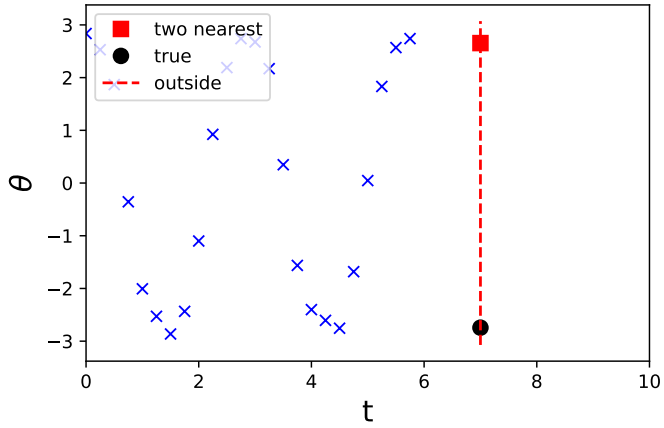
Using a simple neural net:

```
from sklearn import neural_network
regr =neural_network.MLPRegressor(max_iter=5000,
                        activation='logistic',solver='lbfgs')
X = t_data.reshape(-1,1)
y = y_data
regr.fit(X, y)
```



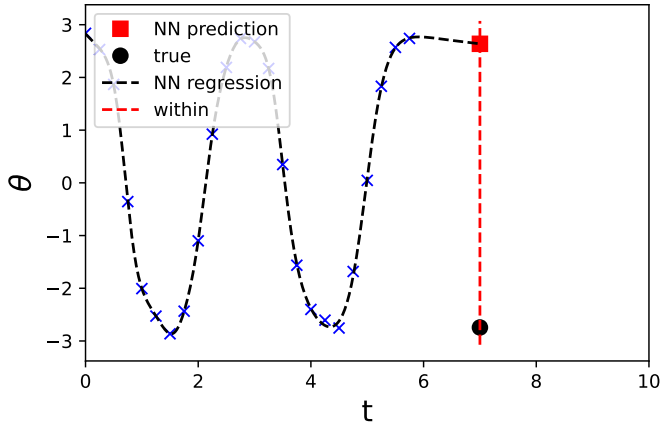
Predicting the pendulum: extrapolation problem

What about the extrapolation problem? Let's naively try our best previous approaches. First, using two closest time points:



Predicting the pendulum: extrapolation problem

Next, a neural network:



Not particularly good or reliable in either case! The problem is that the time point (or feature) we wish to predict is *a priori* quite different to the previous points that we've seen...

However, it *does* perhaps seem like we might be able to predict better than we are currently! Why? One answer is that **the data appear to exhibit some form of regularity**, e.g. approximate periodicity etc.

Can we do better?

Extrapolation problem: simple period-based method

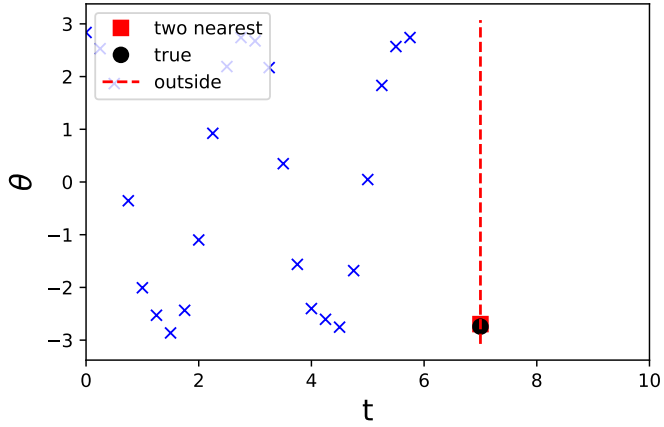
A simple observation is that our data appear somewhat periodic, i.e.

$$\theta(t+T) \approx \theta(t)$$

for some period T. This is also obvious from the solution to the ODE model, i.e. our engineering/scientific knowledge!

The time taken for the pendulum to return to its initial value is approximately 2.75 seconds, which is our estimate of T.

In this case, rather than find the closest absolute time point, we could find the closest point 'within the cycle', i.e. match to t *modulo* T, written 't mod T'. Recall that e.g. 7 mod 5 = 2, 14 mod 6 = 2 etc, i.e. we get the *remainder* when dividing the first number by the second. Our target time, here 7. If our period is about 2.75, then we should take the y values with t values closest to about 7 mod 2.75 = 1.5. Let's try!



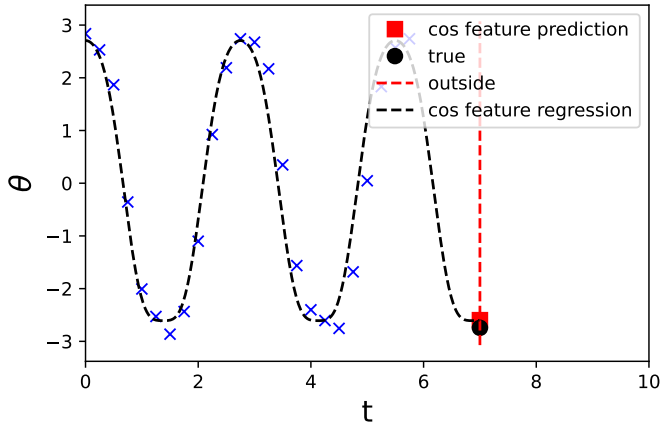
Extrapolation problem: basis functions and features method

The above approach works but requires e.g. a periodic assumption which may not generalise very easily. A related but more general approach is to represent our solution as a linear or nonlinear combination of periodic functions. In the linear case this is a **Fourier series**!

Approaches to time series based on Fourier methods are called **spectral** methods; more generally, this is a **basis function** approach, assuming our function can be easily constructed from linear combinations of a simpler set of basis functions.

Machine learners also call such basis functions **features** and may consider prediction based linear or *nonlinear* combinations of these (functional) features.

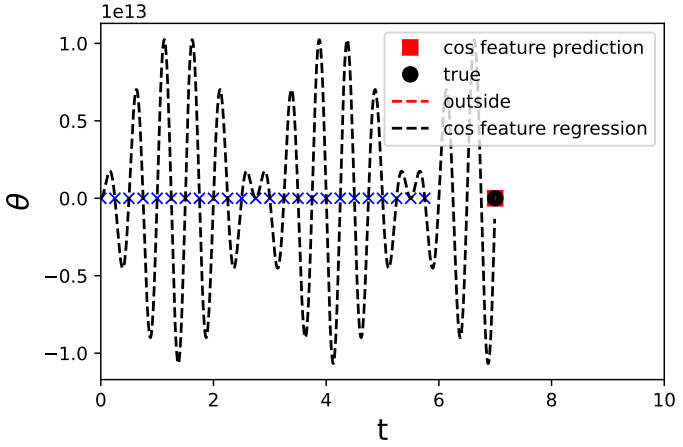
Motivated by a Fourier cosine series, let's try using a small collection of features (basis functions) of the form $\cos \frac{2n\pi t}{T}$, for $n = 1, \ldots, 4$ (say) and $T = 2.75$ (our estimated period), as features in a **linear regression**.



Instability

This works reasonably for these particular choices, but depends on the number and type of terms included.

Here's the same process for $n = 1, \ldots, 6$. Now we are close to interpolating the training data and our predictions are unstable in general (though by luck happen to not be too bad for our actual point of interest, probably because it is about 2.5 periods).



While there are various ways to use and improve spectral methods, let's consider another alternative.

Extrapolation problem: simple autoregressive model

Suppose we wish to use our knowledge that a pendulum can be modelled by a second order differential equation, but don't want to derive or solve the equation as such. We can argue as follows. A first order derivative of $y(t)$ can be approximated (using the 'past', i.e. backward differences) by

$$\frac{dy}{dt} \approx \frac{y(t) - y(t-h)}{h}$$

Given a differential equation of the form

$$\frac{dy}{dt} = f(y)$$

we can thus write this as a simple update rule

$$y(t) \approx y(t-h) + f(y(t-h))h = F(y(t-h))$$

where the present (or future) is given by a an update function F of the past (or present).

A second order time derivative can be discretised approximately as

$$\frac{d^2y}{dt^2} \approx \frac{\frac{y(t)-y(t-h)}{h} - \frac{y(t-h)-y(t-2h)}{h}}{h} = \frac{y(t) - 2y(t-h) + y(t-2h)}{h^2}$$

If we take $h = 1$ for simplicitly, we might expect that a second order system can be approximated by an update rule of the form

$$y(t) \approx F(y(t-1), y(t-2))$$

This motivates us to try a second-order **autoregression**: build a model to predict the current y values from its past two values.

Extrapolation problem: simple autoregressive model

In an autoregressive approach of order k with m realisations of our primary variable, we effectively have a smaple size of $m - k$, i.e. the number of '(past, present)' pairs in the data. E.g. for $m = 6$ and $k = 2$ we get four pairs:

$$(y_1, y_2, y_3, y_4, y_5, y_6)$$
$$(y_1, y_2, y_3)$$
$$(y_2, y_3, y_4)$$
$$(y_3, y_4, y_5)$$
$$(y_4, y_5, y_6)$$

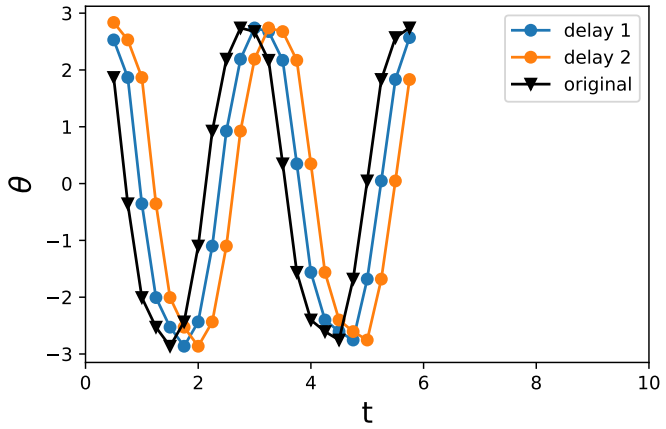
To carry out autoregression, we simply construct a target vector of present values, dropping the first k, and **delay feature vectors**, each lagging the target vector by one additional increment.

E.g. ...

Manual lagged features

We can e.g. use `np.roll` to created lagged features; see below. Note that the future is carrying out its oscillation *before* the lagged variables, as we want!

```
k = 2
y_data_target = np.roll(y_data,0)[k:]
X_lagged = np.zeros((len(y_data_target),k))
for i in range(0,k):
    X_lagged[:,i] = np.roll(y_data,i+1)[k:]
```



Extrapolation problem: simple autoregressive model

Using `X_lagged` and `y_data_target` from above in a *linear regression* model (we can also do nonlinear...) gives us a model of the form

$$\theta_{t+1} \approx F(\theta_t, \theta_{t-1}) = a\theta_t + b\theta_{t-1} + c$$

i.e. a *one-step-ahead* prediction model given the two-lag (relative to the prediction) *history*. How do we predict multiple steps ahead? We **iterate** the one-step prediction model! (Technically we are iterating the *mean* to get the predictive mean.)

This works best if we use the **state space** form to keep track of the current two-lag history required for predicting the next variable and updating to the next relevant history (state):

$$\begin{bmatrix} \theta_{t+1} \\ \theta_t \end{bmatrix} \approx \begin{bmatrix} F(\theta_t, \theta_{t-1}) \\ \theta_t \end{bmatrix} = G\left(\begin{bmatrix} \theta_t \\ \theta_{t-1} \end{bmatrix}\right)$$

In our linear case, assuming $c = 0$ for simplicity, this could be written in matrix form as

$$\begin{bmatrix} \theta_{t+1} \\ \theta_t \end{bmatrix} \approx \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \theta_t \\ \theta_{t-1} \end{bmatrix}$$

Then, applying the predictions sequentially (or recursively), we get the n-step-ahead prediction as the first component of:

$$\begin{bmatrix} \theta_{t+n} \\ \theta_{t+n-1} \end{bmatrix} \approx G(G(\ldots G(\begin{bmatrix} \theta_t \\ \theta_{t-1} \end{bmatrix}))) = G^n\left(\begin{bmatrix} \theta_t \\ \theta_{t-1} \end{bmatrix}\right)$$

i.e. for our linear case

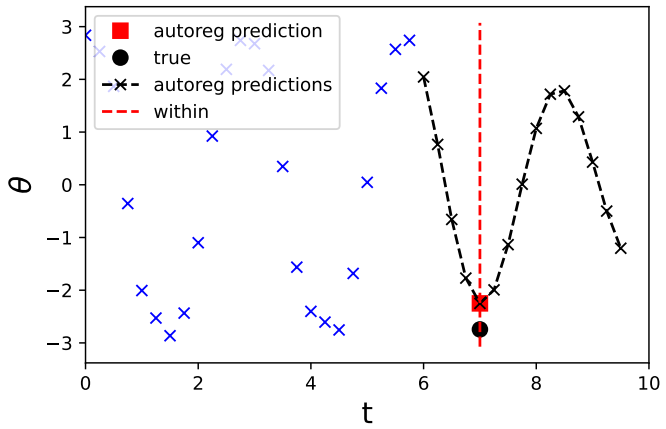
$$\begin{bmatrix} \theta_{t+n} \\ \theta_{t+n-1} \end{bmatrix} \approx A^n \begin{bmatrix} \theta_t \\ \theta_{t-1} \end{bmatrix}$$

with

$$A = \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}.$$

Extrapolation problem: simple autoregressive model

Let's predict both the point of interest and a few extra time steps beyond that:



Not bad! It captures the general oscillatory behaviour, though it does seem to be more damped than the true motion. This is not uncommon as we are emphasising fit to the **dynamics**, sometimes called **gradient matching**, rather than fit to the **trajectory**, sometimes called **trajectory** or **integral matching** (terms used when fitting ODE models in the classical style).

More sophisticated approaches along these lines include more complex **linear** models at both the dynamics and trajectory level, as well as **nonlinear time series** versions. **State space** approaches are particularly powerful, and lead to **Kalman filtering** and nonlinear extensions for smoothing and predicting time series.

In addition, the basic concept of **autoregression** is a key component of modern LLM neural nets like ChatGPT! It essentially predicts the next word based on the input prompt (history).

Assumptions make data representative!

Let's regroup. What have we learned? We have seen that in order to predict new data we have to **make assumptions that go beyond the data**.

This is a general lesson that has been known since at least **Hume** (1739) introduced the **problem of induction** (likely much much earlier)!

In *A Treatise of Human Nature*, Hume states:

> there can be no demonstrative arguments to prove, that those instances, of which we have had no experience, resemble those, of which we have had experience

The physicist Max Born in *Natural Philosophy of Cause and Chance* (1949) summarised the problem as:

> no observation or experiment, however extended, can give more than a finite number of repetitions; therefore, the statement of a law – B depends on A – always transcends experience. Yet this kind of statement is made everywhere and all the time, and sometimes from scanty material.

There have been a range of proposed resolutions of the problem of induction, from probabilistic proposals to Popper's argument that science does not need induction, summarised in *Conjectures and Refutations* (1962):

> Hume, I felt, was perfectly right in pointing out that induction cannot be logically justified...Induction, i.e. inference based on many observations, is a myth. It is neither a psychological fact, nor a fact of ordinary life, nor one of scientific procedure...The actual procedure of science is to operate with conjectures...Repeated observations and experiments function in science as *tests* of our conjectures or hypotheses, i.e. as attempted refutations.

Regardless of one's preferred approach to the problem of induction, **all involve making assumptions (or at least conjectures) to go beyond the available data**, whether explicit or implicit.

Features vs architecture vs regularisation

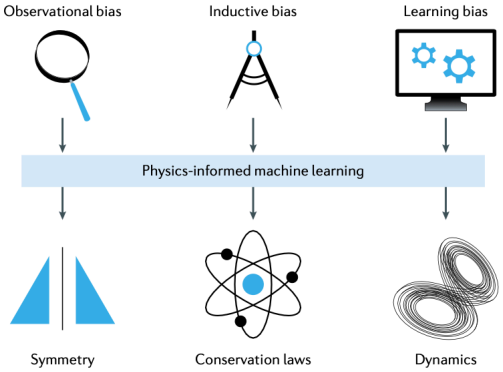
In machine learning, people often talk about 'inductive bias' and related types of **good**, or at least **necessary biases**. E.g., from Karniadakis et. al (2021):

Box 2 | **Principles of physics-informed learning**

Making a learning algorithm physics-informed amounts to introducing appropriate observational, inductive or learning biases that can steer the learning process towards identifying physically consistent solutions (see the figure).

- Observational biases can be introduced directly through data that embody the underlying physics or carefully crafted data augmentation procedures. Training a machine learning (ML) system on such data allows it to learn functions, vector fields and operators that reflect the physical structure of the data.
- Inductive biases correspond to prior assumptions that can be incorporated by tailored interventions to an ML model architecture, such that the predictions sought are guaranteed to implicitly satisfy a set of given physical laws, typically expressed in the form of certain mathematical constraints. One would argue that this is the most principled way of making a learning algorithm physics-informed, as it allows for the underlying physical constraints to be strictly satisfied. However, such approaches can be limited to accounting for relatively simple symmetry groups (such as translations, permutations, reflections, rotations and so on) that are known a priori, and may often lead to complex implementations that are difficult to scale.
- Learning biases can be introduced by appropriate choice of loss functions, constraints and inference algorithms that can modulate the training phase of an ML model to explicitly favour convergence towards solutions that adhere to the underlying physics. By using and tuning such soft penalty constraints, the underlying physical laws can only be approximately satisfied; however, this provides a very flexible platform for introducing a broad class of physics-based biases that can be expressed in the form of integral, differential or even fractional equations.

These different modes of biasing a learning algorithm towards physically consistent solutions are not mutually exclusive and can be effectively combined to yield a very broad class of hybrid approaches for building physics-informed learning machines.



Regularisation and hybrid models

We have given a taste of different ways to make assumptions and improve predictions beyond our immediate data.

For the remainder, we will look in more detail at simple ways to build models and incoporate engineering knowledge. In particular, we are going to focus on what the previous quoted article calls **learning bias**, which amounts to including physical or engineering constraints in a 'soft' (vs. exactly enforced) manner, via penalty or so-called **regularisation** terms.

Because such an approach includes both a flexible data model and an approximately enforced physical model, we will call these **hybrid models** (though this term is used by many authors in many ways).

Note: Regularisation and related topics are covered in much more detail in e.g. **ENGSCI 721**.

Part II

Basic tools

Vectors, matrices, arrays, tensors

In engineering and machine learning it's important to be able to store and manipulate **data** effectively and efficiently, as well as represent **models** effectively and efficiently. Here we will focus on **Python**-based tools, though other languages such as **R**, **Julia**, **MATLAB** and even **C/C++** and **Fortran** could also be used.

One of the central concepts in both scientific computing and machine learning is the **multi-dimensional array**. The main library for working with such arrays in Python is **NumPy**, which you would have seen. However, as we will look at, many machine learning libraries are based on their own improved NumPy-like libraries, usually using the term **tensor** instead of array.

Note: While highly structured (e.g. feature rich) data is typically organised into **tabular** format using **data table** data structures like those provided by Pandas, datatable and Polars, data provided in less structured form (e.g. explicit features not given) such as **text**, **image**, **video**, **audio**, **network** etc. data are often stored in lower-level data structures, usually multi-dimensional arrays. These are the basic inputs into **neural networks**, which excel in particular with non-tabular data and automatically learning features. Similarly, when solving physical simulation models, e.g. involving **differential equations** describing information evolving in **space and time**, the variables are usually stored in multi-dimensional ararys rather than data tables.

Tensors

Machine learners like to call multi-dimensional arrays **tensors**. Be warned that 'tensor', much like 'vector', is a term with many different definitions and meanings across mathematics, physics, computing, etc. Here we think of a number as a zeroth order array, a vector as a one-dimensional array, a matrix as a two-dimensional array/tensor, and a tensor in general as an n-dimensional array for any n. Thus a number is a tensor, a vector is a tensor, a matrix is a tensor, a three-dimensional array is a tensor etc. For us, **tensor $\equiv$ n-dimensional array**.

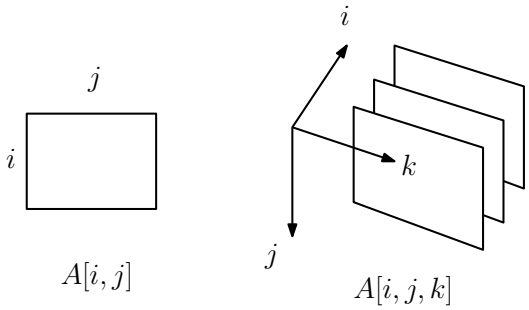
Just like a two-dimensional matrix might just represent plain data stored in a two dimensional array but could also represent a **linear mapping** of one-dimensional arrays (vectors) to one-dimensional arrays (vectors), a tensor can represent e.g. just n-dimensional data arrays but also a **linear mapping of lower dimensional tensors to lower dimensional tensors**. For example, **linear image filters** can be thought of as four-dimensional tensors that linearly map two-dimensional images to two-dimensional images. *For now, though,* just remember **tensor $\equiv$ n-dimensional array**.

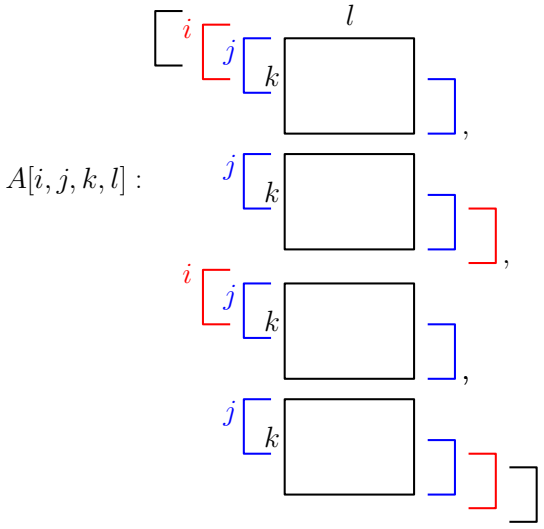
To carry out machine learning tasks, we will consider a library that duplicates much of what NumPy offers, while adding additional features important for machine learning: **PyTorch**. While this is designed specially for **neural networks**, which we will look at later, for now we can think of it as a direct NumPy replacement for working with arrays and doing basic linear algebra (typically with the same syntax and function names). As you might expect, though, **what NumPy calls an array PyTorch calls a tensor!** Similar libraries exist, e.g. **JAX**, which again closely follows NumPy while adding additional features useful for machine learning.

A nice reference for PyTorch, that we draw on a lot here, is *'Deep Learning with PyTorch'* by Stevens, Antiga, and Viehmann (2020).

Arrays/tensors

Tensors are hard to visualise in high-dimensions. At some point it becomes easier to just think *abstractly in terms of indices*. However, you can also imagine *nested indexing of two-dimensional arrays* (third figure):





Arrays in NumPy

Here are some simple multi-dimensional arrays in NumPy:

```
import numpy as np
A = np.array([[1,2,3],[4,5,6]])
print('Array and dimensions:')
print(A)
print(A.shape)
```

```
Array and dimensions:
[[1 2 3]
 [4 5 6]]
(2, 3)
```

```
A = np.array([[[1,2,3],[4,5,6]],
              [[7,8,9],[10,11,12]],
              [[13,14,15],[16,17,18]]
             ])
print('Array and dimensions:')
print(A)
print(A.shape)
```

```
Array and dimensions:
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]

 [[13 14 15]
  [16 17 18]]]
(3, 2, 3)
```


Tensors in PyTorch

Tensors work the same as arrays in NumPy. E.g.

```
import torch
A = torch.tensor([[1,2,3],[4,5,6]])
print('Tensor and dimensions:')
print(A)
print(A.shape)
```

```
Tensor and dimensions:
tensor([[1, 2, 3],
        [4, 5, 6]])
torch.Size([2, 3])
```

```
A = torch.tensor([[[1,2,3],[4,5,6]],
                  [[7,8,9],[10,11,12]],
                  [[13,14,15],[16,17,18]]
                  ])
print('Tensor and dimensions:')
print(A)
print(A.shape)
```

```
Tensor and dimensions:
tensor([[[ 1,  2,  3],
         [ 4,  5,  6]],

        [[ 7,  8,  9],
         [10, 11, 12]],

        [[13, 14, 15],
         [16, 17, 18]]])
torch.Size([3, 2, 3])
```

Why? GPUs! Derivatives!

While NumPy provides a range of highly efficient array operations and linear algebra, PyTorch and related libraries for deep learning bring (at least) two key extra ingredients: ability to use **GPUs** (Graphical Processing Units) for computation and ability to automatically compute **derivatives** of arbitrarily composed functions of tensors.

We will return to the derivative capability shortly. Firstly, what about GPUs? In the 2000s, Graphical Processing Units (GPUs) were significantly developed to support the needs of increasingly realistic computer games. This turned out to make them especially useful for highly **parallel** computation in general. Neural networks, like much of scientific computing, involves large numbers of highly parallelisable computation and machine learning researchers in the 2010s began using GPUs to train large neural networks. In addition, there have been recent efforts to develop specialised TPUs – Tensor Processing Units!

PyTorch and related frameworks are designed to easily move computations to GPUs (and TPUs). PyTorch tensors have an attribute `device`, which is where the tensor data is stored and operated on. The standard is to use `device=cuda`, where CUDA is NVIDIA's programming interface to their GPUs. Google Colab (which we will use in labs) provides access to GPUs for free, and we will look at using these via PyTorch during labs. For simple operations and models, however, we can just use normal CPUs.

But first: basic tensor usage in PyTorch

See http://pytorch.org/docs for full details! Roughly, we have

- Tensor creation
- Indexing, slicing, joining, mutating operations
- Mathematical operations
  - Pointwise functions applying to each element
  - Aggregation operations (like mean or standard deviation) over multiple elements
  - Linear algebra operations for scalar, vector and matrix operations
  - Random sampling operations
- Data saving and loading (serialisation)
- Etc!

As mentioned, many of these follow the same syntax as NumPy, when applicable.

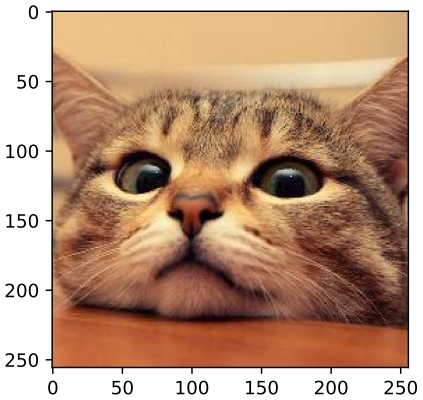
Here we will mainly look at some simple manipulations of **image** data for illustration, which comes up in many engineering applications e.g. medical imaging, as well as when dealing with **spatial** data in general, like measuring a temperature field. Instead of medical or physical field data, though, we will use a cute cat for illustration!

Bonus: There is also a nice package `einops`, compatable with NumPy, PyTorch, TensorFlow, JAX etc, that encodes various tensor operations in a nice 'Einstein-inspired' tensor index notation.

Loading data

We can load images with many libraries, including PyTorch. Here, though, we will first use the `imageio` library to load in an image of a cute cat (from the dataset associated with the book *'Deep Learning with PyTorch'* referenced above) as a NumPy array.

```
import imageio
import matplotlib.pyplot as plt
img_arr = imageio.imread('./data/cat3.png')
plt.figure()
plt.imshow(img_arr)
plt.show()
print('image array shape:')
print(img_arr.shape)
```




```
image array shape:
(256, 256, 3)
```

Let's turn this into a PyTorch tensor.

Converting to a tensor

```
# create a pytorch tensor
img_tensor = torch.from_numpy(img_arr)
img_tensor = img_tensor.float()/255 # ensure float32 dtype
```

Note though (see above) the shape is Height x Width x Channels, where Channels holds the Red Green Blue (RGB) color information. While this is standard for NumPy (and Matplotlib), **PyTorch typically assumes a Channels x Height x Width format** for images.

This is a chance to use the `permute` operation (which, as described below, actually constructs a *view* of the original data rather than a copy):

```
# create a pytorch tensor as a view
# new index ordering (0,1,2) -> (2,0,1)
cat = torch.permute(img_tensor, (2, 0, 1))
print('Tensor shape (size):')
print(cat.shape)
```

```
Tensor shape (size):
torch.Size([3, 256, 256])
```

We can also, e.g., **slice** tensors just like with NumPy arrays.

```
# create a pytorch tensor as a view
# new index ordering (0,1,2) -> (2,0,1)
cat_red = cat[0,:,:]
print('Tensor slice shape (size):')
print(cat_red.shape)
```

```
Tensor slice shape (size):
torch.Size([256, 256])
```

Arrays/tensors: storage vs access

Although we interact with tensors as multi-dimensional arrays, for efficient computation, tensor values are actually *stored* in memory in linear, contiguous chunks (i.e. continuous sequence of memory locations). When working with tensors, we typically just change how we *access* or *index* the underlying chunk of memory rather than copying the stored values, saving memory. This idea is represented by a **view**, a 'virtual' tensor that accesses the same memory but with a different shape or indexing.

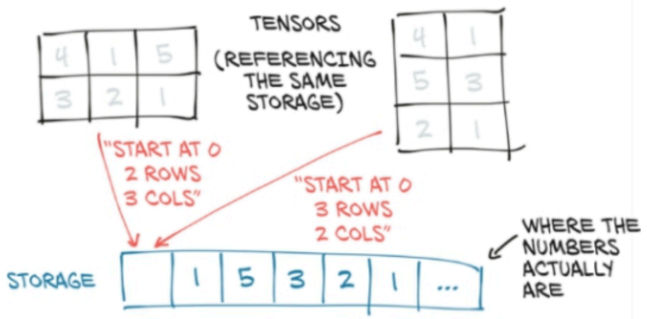
Some operations, like slicing, may generate tensors with non-contiguous memory layouts, which can cause issues for certain operations. In such cases, the `contiguous` tensor method can be used to return a new tensor with a contiguous memory layout. The `reshape` operation is an example of a method that tries to create a view if the memory is contiguous, but if the memory is non-contiguous or the new shape can't be represented as a view, creates a contiguous copy instead. The `view` function explicitly creates a view, enforcing that no data is copied, and will give an error if this can't be done.

It can important to know whether a tensor operation copies the underlying data or just returns a new view of the same data; e.g. modifying values via a view modifies the results of all other views of the same data, while modifying a copy doesn't. However, making a copy means twice the memory! In general, check the documentation of the given operation if you need to know what's happening! You can manually check by using `.storage().data_ptr()` on two tensors to check if they point to the same memory or not.

```
# same memory location (same pointer)!
print(cat_red.storage().data_ptr())
print(cat.storage().data_ptr())
```

```
140395253039104
140395253039104
```


Arrays/tensors: storage vs access (views)



From *'Deep Learning with PyTorch'* by Stevens, Antiga, and Viehmann (2020)!

We won't focus too much on this here, but it can be important to understand how views and storage in more advanced use cases. In brief:

- A 'storage', managed by `torch.Storage`, is a one-dimensional array of numerical data contained in a block of contiguous memory.
- A tensor can be thought of as a **view** of some storage, i.e. a way of *indexing* or *accessing* some fixed storage. Views are defined by three attributes: a **size** (shape) tuple, a **storage offset** number, and a **stride** tuple. The stride tuple gives the number of elements for *each dimension* that need to be skipped *in storage* between consecutive elements along the *given dimension in the tensor*.

For more, see the book *'Deep Learning with PyTorch'*, from where the above picture is taken.

Common shapes

When working with data we usually have many samples. E.g. rather than only having a single image, we usually have many example images in our training set. Because neural networks and other machine learning algorithms typically work with very large datasets, datasets are usually divided into **batches**: smaller subsets of the full dataset. This reduces the memory load and allows for efficient parallel processing using GPUs. For this reason, the 'sample number' dimension in tensor datasets is often called the **batch** dimension, with indices ranging over the number of samples *in the batch*.

Typical tensor indexing layouts for various types of data are:

- (Batch, Features) for **tabular**/structured datasets
- (Batch, Timesteps, Features) for **time-series** datasets
- (Batch, Channel, Height, Width) for **image**/2D spatial datasets
- (Batch, Channel, Depth, Height, Width) for **volumetric**/3D spatial datasets
- (Batch, Frame, Channel, Height, Width) for **video**/spatiotemporal datasets

etc.

Note that some libraries, e.g. matplotlib for plotting, TensorFlow for neural networks, adopt a *channels last* convention for image data, while PyTorch adopts a *channels first* (relative to image dimensions) convention. This is because it can be more efficient for particular image operations like **convolutions**. We can always quickly switch the indexing by calling `permute`, or related operations, which reorders the indices without copying the data, making it memory-efficient.

Using 'unfold' to generate 'sliding window views' along a dimension

PyTorch tensors have an `unfold(dimension, size, step)` method that returns a *view* of the original tensor which contains *all slices* of size `size` from the tensor along the dimension given by `dimension`, taking steps of length `step`.

You can think of this as using a *sliding window* to generate a sequence of *views* along a given dimension.

The returned tensor (view) has the same shape as the original except that it replaces the *unfolded dimension* by a new dimension of size `floor((original_size - size) / step) + 1` at the same location, *and* adds an *additional* dimension of size `size` at the *end* of the tensor.

If we imagine indexing into $A[i, j, k, ...]$ in the order $i, j, ...$ then we:

- Index as usual in the same order *until the unfolded index* is reached
- The unfolded index is then replaced at that point by a *new* index corresponding to the *window number* of the sliding window that slides along that dimension
- An additional *within window* index is added to the *very end* of the indexing chain
- We continue with the rest of the indexing until we reach the within window index we added
- We *finally* index within the window.

Using ‘unfold’ to generate ‘sliding window views’ along a dimension

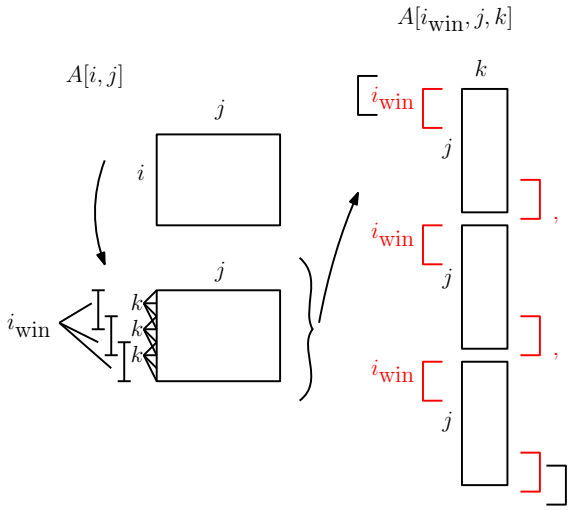
In terms of indices, we get e.g.

$$[i, j, k] \overset{\text{unfold(dim=0)}}{\mapsto} [i_{\text{window}}, j, k, l_{\text{within } i_{\text{window}}}]$$

$$[i, j, k] \overset{\text{unfold(dim=1)}}{\mapsto} [i, j_{\text{window}}, k, l_{\text{within } j_{\text{window}}}]$$

where the subcript ‘window’ refers to the ‘window number’ index and subscript ‘within’ to the element number (starting from 0 again) ‘within’ the window given by the window number index.

Note that **the other indices are left alone** and the **‘within window’** index is added to the **end** of the index list, hence becoming the **column** index for when displayed on screen. Visually:



Example

```
x = torch.arange(1., 7)
print(x)
print(x.shape)
y = x.unfold(0, 2, 1)
print('')
print(y)
print(y.shape)
z = x.unfold(0, 2, 2)
print('')
print(z)
print(z.shape)
```

```
tensor([1., 2., 3., 4., 5., 6.])
torch.Size([6])

tensor([[1., 2.],
        [2., 3.],
        [3., 4.],
        [4., 5.],
        [5., 6.]])
torch.Size([5, 2])

tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
torch.Size([3, 2])
```

Note: because the ‘within window’ index is added to the end, this will **index the columns** when displayed as a nested collection of two-dimensional arrays.

Example

```
x = torch.arange(1., 7).view(2,3) #like reshape
print(x)
print(x.shape)
y = x.unfold(0, 2, 1)
print('')
print(y)
print(y.shape)
z = x.unfold(1, 2, 1)
print('')
print(z)
print(z.shape)
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
torch.Size([2, 3])

tensor([[[1., 4.],
         [2., 5.],
         [3., 6.]]])
torch.Size([1, 3, 2])

tensor([[[1., 2.],
         [2., 3.]],

        [[4., 5.],
         [5., 6.]]])
torch.Size([2, 2, 2])
```


Example

Given a matrix we can unfold along each dimension sequentially to generate two-dimensional sliding windows. That is,

$$[i, j] \overset{\text{unfold(dim=0)}}{\mapsto} [i_{\text{window}}, j, k_{\text{within } i_{\text{window}}}]$$

$$[i_{\text{window}}, j, k_{\text{within } i_{\text{window}}}] \overset{\text{unfold(dim=1)}}{\mapsto} [i_{\text{window}}, j_{\text{window}}, k_{\text{within } i_{\text{window}}}, l_{\text{within } j_{\text{window}}}]$$

```
x = torch.arange(1., 9).view(2,4) #like reshape
print(x)
print(x.shape)
y = x.unfold(0, 2, 1).unfold(1,2,1)
print('')
print(y)
print(y.shape)
```

```
tensor([[1., 2., 3., 4.],
        [5., 6., 7., 8.]])
torch.Size([2, 4])

tensor([[[[1., 2.],
          [5., 6.]],

         [[2., 3.],
          [6., 7.]],

         [[3., 4.],
          [7., 8.]]]])
torch.Size([1, 3, 2, 2])
```

Using unfold to generate overlapping patches of an image

Unfolding in multiple dimensions can be a little confusing but you'll get the hang of it! Let's try partition a 4-by-4 2D tensor into a 4D tensor made up of four non-overlapping 2-by-2 tensors:

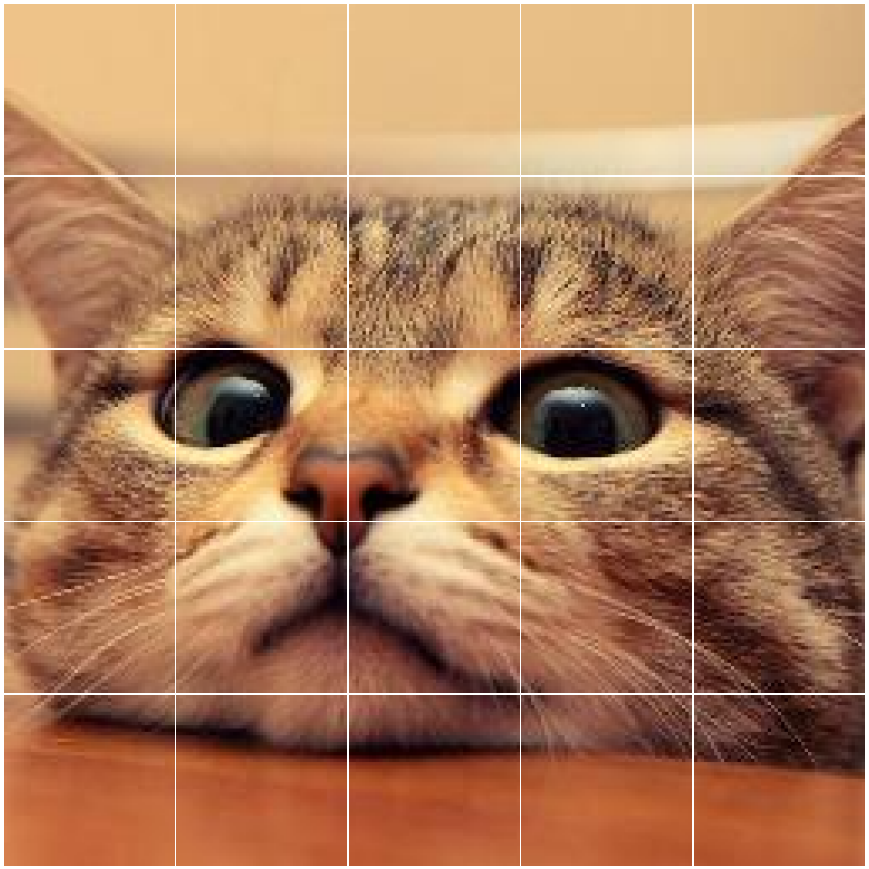
```
A = torch.arange(16).view(4,4)
print(A)
print('')
B = (A.unfold(0,2,2)).unfold(1,2,2)
print(B)
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]])

tensor([[[[ 0,  1],
          [ 4,  5]],

         [[ 2,  3],
          [ 6,  7]]],


        [[[ 8,  9],
          [12, 13]],

         [[10, 11],
          [14, 15]]]])
```


Cat split into patches

Now let's split our cat picture into *non-overlapping patches*, setting `step = size`, and plot our patched cat!

```
# Move channels to last dimension for plotting
patches = cat.unfold(1, 51, 51).unfold(2, 51, 51)
patches = patches.permute(1, 2, 3, 4, 0)
# Number of patches along each dimension
num_patches_x, num_patches_y = patches.shape[:2]

# Initialize a plot
fig, axes = plt.subplots(num_patches_x, num_patches_y,
                         figsize=(51, 51))

# Plot each patch
for i in range(num_patches_x):
    for j in range(num_patches_y):
        patch = patches[i, j]
        axes[i, j].imshow(patch)
        axes[i, j].axis('off')

plt.tight_layout()
plt.show()
```



Zero padding

We can also first *zero pad* around the edges of the image to ensure potentially *overlapping* windows can fit at *each pixel*. PyTorch has a special function `pad` for this in the `torch.nn.functional` module. Let's make cat patches of size 51-by-51, one for each pixel, so pad by 25.

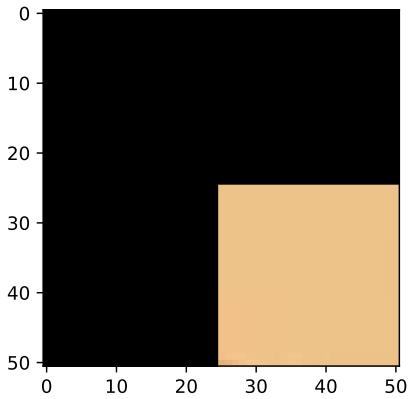
```
# zero padding
import torch.nn.functional as F
padded_cat = F.pad(cat, pad=(25, 25, 25, 25),
                   mode='constant', value=0)
```

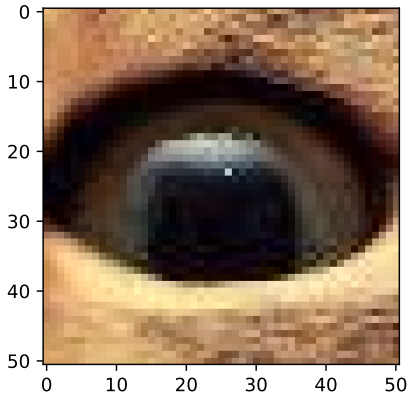
We first unfold the first image dimension (after channels) into length 51 slices, one for each pixel. Then unfold into length 51 slices along the other image dimension similarly. This creates 51 x 51 size patches, one for each pixel, and each containing 3 colour channels.

```
patches = padded_cat.unfold(1, 51, 1).unfold(2, 51, 1)
patches.shape
```

```
torch.Size([3, 256, 256, 51, 51])
```

```
# Visualise some patches.
# Need to move the colour channel last for matplotlib.
patches_disp = patches.permute(1,2,3,4,0)
plt.figure()
plt.imshow(patches_disp[0,0,:,:]) # corner patch
plt.show()
plt.figure()
plt.imshow(patches_disp[120,170,:,:]) # ...eye patch
plt.show()
```

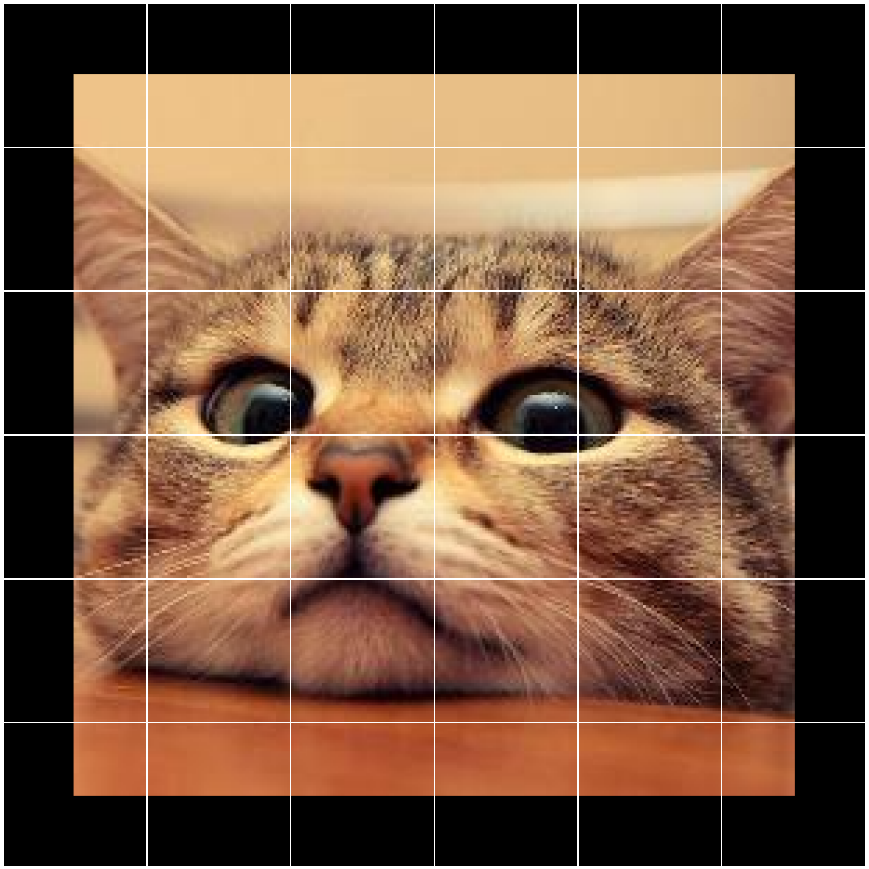




Cat split into patches: zero padding

Now let's add zero padding to see the effect. For simplicity we will use non-overlapping windows so we don't have to try visualise 256*256 = 65536 windows!

```
# Move channels to last dimension for plotting
patches = padded_cat.unfold(1, 51, 51).unfold(2, 51, 51)
patches = patches.permute(1, 2, 3, 4, 0)
# Number of patches along each dimension
num_patches_x, num_patches_y = patches.shape[:2]

# Initialize a plot
fig, axes = plt.subplots(num_patches_x, num_patches_y,
                         figsize=(51, 51))

# Plot each patch
for i in range(num_patches_x):
    for j in range(num_patches_y):
        patch = patches[i, j]
        axes[i, j].imshow(patch)
        axes[i, j].axis('off')

plt.tight_layout()
plt.show()
```



Why patches?

Note that a patch is a **local** region in a spatial object like an image. Just like we typically model dynamic physical systems as evolving according to local-in-time dynamics, with the update step depending on the recent past, **spatial** and **spatiotemporal** dependencies are also typically **local** in nature.

What does physics say about why this is? When Netwon proposed his famous Law of Gravitation, it involved, mathematically, action at a distance. However, Newton was not happy about this, saying it was

> it is inconceivable that inanimate brute matter should, without the mediation of something else which is not material, operate upon and affect other matter without mutual contact

Newton's discomfort was later resolved by the development of **field theories** in physics, where interactions occur through the exchange of field quanta (e.g., bosons), meaning that forces propagate through space-time locally.

In machine learning for image analysis, Yann LeCun and others noted that the individual neurons of the visual system in animals each respond only to small, local, overlapping regions of the overall visual field of view. Patches! This led to **convolutional neural networks**, that were until recently the gold standard machine learning approach to image analysis. These focus initially on detecting edges and texture, before aggregating these at higher and higher levels to reconstruct the overall image.

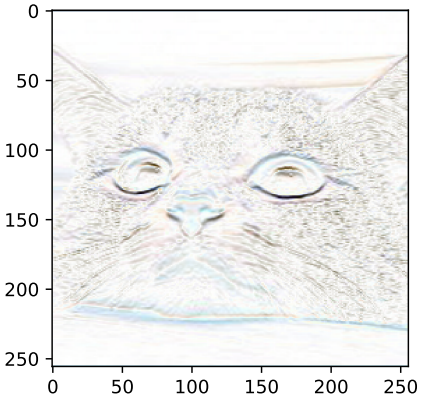
More recently, people have adapted so-called **transformer** models, traditionally used for language processing, to apply to **image patches**, where these are essentially treated as the 'words' of an image! Again, these local features are 'aggregated' into a global picture of the image, like making sentences from words.

So, in physics, biology and image analysis, we have good motivation to focus on **local regions in space and time**, and this is motivated by our scientific and engineering understanding of the world!

Cat edges

We will look at some of these ideas in more detail later. For now, here is the output of applying a **spatially local** (here patch-wise for patches centred at each pixel) **vertical edge detection** filter to our cat! This involves applying a vertical finite difference operator to each patch.

I did this manually by creating the patches using `torch.unfold` and then mutiplying each patch by a filter **kernel** using something called `torch.einsum`. There are, of course, many built-in filters in many libraries for doing this sort of thing.



Back to derivatives

Once we have data and a machine learning model, we need to carry out **parameter fitting**, or **weight learning** for neural networks, where we typically aim to **minimise some loss function**. The loss function measures the machine learning model's fit to data for any given set of model parameters (e.g. network weights). This is called **training** the model, and is what a statistician or engineer typically calls **fitting** a model or **estimating parameters**.

Neural networks are usually trained by some form of **gradient descent**, where we iteratively update our model weights (i.e. parameter estimates) by moving in the direction of **greatest local decrease** in the loss function. This is not guaranteed to find a global minimum for complex functions, but will typically find at least a local minimum, and other tricks (e.g. Stochastic Gradient Descent) can help the algorithm find 'better minima' that we hope **generalise** well.

This leads to the other important extra ingredient we mentioned that PyTorch brings to the table: the ability to **automatically compute derivatives** of (fairly) arbitrary expressions made up of tensors and functions of tensors. If we think of our loss as

$$\text{loss}(w, d) = \text{loss}(y(w), d)$$

for parameters (weights) w, data d and $y(w) = \text{model}(w)$. We usually want to minimise this with respect to the parameters w while holding the (training) data fixed. To compute the derivative with respect to w we can apply the chain rule. When the loss, model output, and w are all scalars, we get

$$\frac{\partial}{\partial w}\text{loss}(y(w), d) = \frac{\partial \text{loss}}{\partial y}\frac{\partial y}{\partial w}$$

Neural network models are typically made up of a series of **composed functions** (giving 'layers'), essentially $y(w) = f(g(h(k(\ldots(w)))))$, and applying the chain rule gives a long series of multiplications of derivatives.

In principle we can compute derivatives of many expressions by hand. However, PyTorch can do it for us, automatically and efficiently.

Computation graphs

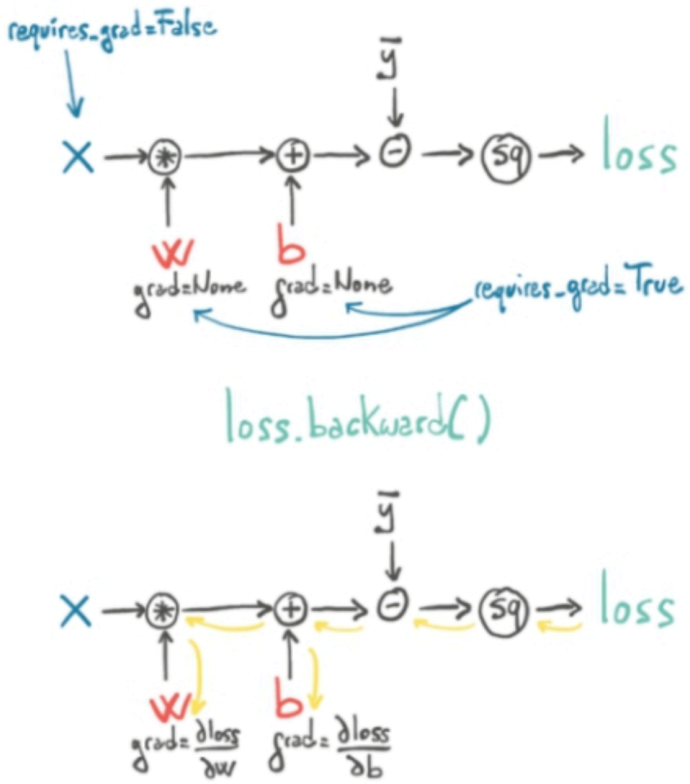
We will look at neural nets and gradient descent in more detail later. For now, the important thing is that during computation, we can imagine our tensors undergoing a series of basic operations, like multiplication, addition, squaring etc. This defines a **computation graph**. Normally, we just track the **values** going in and coming out of the functions. However, **PyTorch can also track the derivatives** of the functions along the computation graph and multiply them together to obtain the overall derivative (i.e. apply the chain rule!).

It turns out that, while we compute the values and individual derivatives as we move **forward** through the computation graph, it is usually more efficient to multiply the derivatives in the **reverse** or **backward** direction -— starting from the loss and working back toward the parameters. This efficiency generally becomes more pronounced the more parameters we have. We won't go into too much detail on this now, but this is the essence of the famous **backpropagation** approach to computing derivatives and training neural networks.

An illustration, again from *'Deep Learning with PyTorch'* by Stevens, Antiga, and Viehmann (2020), is shown below.

There's a lot going on here, so let's look at a very simple, non-neural example!

Computation graphs



From *'Deep Learning with PyTorch'* by Stevens, Antiga, and Viehmann (2020)!

Example

The main requirement required to track derivatives through a computation graph is to set the attribute `require_grad` of the **input** tensor to `True`. This means that the derivatives are tracked during the forward computation step, and can *then* be multiplied backwards to compute the **overall derivative of the output with respect to that input** (we *can* multiply forwards in principle too!).

Let's solve

$$\min_x (x-10)^2$$

```python
# Define the tensor variable to optimise and
# keep track of gradients of anything that depends on it.
x = torch.tensor([0.0], requires_grad=True)

# Define the optimizer. Only requires param not loss.
# Stochastic Gradient Descent with learning rate 0.1
optimizer = torch.optim.SGD([x], lr=0.1)

# Define the function to minimize
def func(x):
    return (x - 10) ** 2

# Optimization loop. Take 30 steps
for i in range(30):
    optimizer.zero_grad() # Clear gradients from the last step.
    loss = func(x)        # Compute the loss (objective function)
    loss.backward()       # Backpropagate to compute gradients
    optimizer.step()      # Adjust x based on gradients

    print(f'Step {i}: x = {x.item()}, loss = {loss.item()}')

# Final result
print(f'Optimized x: {x.item()}')
```



```
Step 0: x = 2.0, loss = 100.0
Step 1: x = 3.5999999046325684, loss = 64.0
Step 2: x = 4.880000114440918, loss = 40.96000289916992
Step 3: x = 5.904000282287598, loss = 26.214399337768555
Step 4: x = 6.72320032119751, loss = 16.77721405029297
Step 5: x = 7.3785600662231445, loss = 10.73741626739502
Step 6: x = 7.902848243713379, loss = 6.871947288513184
Step 7: x = 8.32227897644043, loss = 4.398045539855957
Step 8: x = 8.65782356262207, loss = 2.8147478103637695
Step 9: x = 8.92625904083252, loss = 1.8014376163482666
Step 10: x = 9.141007423400879, loss = 1.1529196500778198
Step 11: x = 9.312806129455566, loss = 0.7378682494163513
Step 12: x = 9.450244903564453, loss = 0.47223541140556335
Step 13: x = 9.560195922851562, loss = 0.3022306561470032
Step 14: x = 9.648157119750977, loss = 0.19342762231826782
Step 15: x = 9.718525886535645, loss = 0.12379341572523117
Step 16: x = 9.774820327758789, loss = 0.07922767847776413
Step 17: x = 9.819856643676758, loss = 0.05070588365197182
Step 18: x = 9.85588550567627, loss = 0.032451629638671875
Step 19: x = 9.884708404541016, loss = 0.020768987014889717
Step 20: x = 9.907766342163086, loss = 0.013292152434587479
Step 21: x = 9.926213264465332, loss = 0.008507047779858112
Step 22: x = 9.940970420837402, loss = 0.005444482434540987
Step 23: x = 9.952775955200195, loss = 0.0034844912588596344
Step 24: x = 9.962221145629883, loss = 0.0022301103454083204
Step 25: x = 9.96977710723877, loss = 0.0014272418338805437
Step 26: x = 9.975821495056152, loss = 0.0009134232532233
Step 27: x = 9.980657577514648, loss = 0.0005846000858582556
Step 28: x = 9.984525680541992, loss = 0.0003741293039638549
Step 29: x = 9.98762035369873, loss = 0.0002394545590505004
Optimized x: 9.98762035369873
```

Notes

There's a lot here that we haven't gone into detail on. Briefly though,

- Setting `requires_grad=True` on a tensor tells PyTorch to track all 'elementary' derivatives for operations involving that tensor.

- Calling `loss.backward()` computes the overall derivative (gradient) of loss with respect to all tensors that have `requires_grad=True` or are related to such tensors via the computation graph.

- When we call `.backward()` on one or more functions (e.g. multiple loss functions), PyTorch computes and accumulates the gradients of these functions with respect to all variables that have `requires_grad=True`. Note: multiple `.backward()` calls require `retain_graph=True` for all but the last call, otherwise PyTorch wipes the graph

- Backpropagation (`.backward()`) applies the chain rule, multiplying derivative terms from output to input (i.e., backward through the computation graph).

- We use an `optimizer` to update specific tensors. The optimizer adjusts the specified tensors based on the gradients accumulated by `.backward()`.

- The optimizer accumulates gradients over multiple steps unless we explicitly clear them by calling `optimizer.zero_grad()`. Clearing gradients is generally good practice as most of the time we don't want accumulated gradients.

We will return to this topic when we look at training neural networks! First, a detour to a simpler, linear world!

Part III

Simple data models

Linear models

Currently (as of 2024 ...), the most exciting class of model in machine learning is undoubtedly the **neural network**. However, it is worth first spending time on a simpler class of model, **linear models**. Why? Firstly, they are more than sufficient for many practical applications, particularly with well-structured, 'tabular' data. Linear models are simple yet powerful, well-understood, and easy to 'train'. While neural networks can provide exceptional performance, particularly on 'unstructured' image, video, language, etc. data, they can be challenging and computationally intensive to train. For many engineering and scientific applications, linear models suffice. There are at least two other reasons to consider linear models: 1) neural networks themselves can be thought of as, for common choices of activation function, essentially (very large) *piecewise* linear models, and understanding the linear components aids in understanding the overall model, and 2) they serve as an easy setting to explore the incorporation of engineering and scientific information in the form of additional **regularisation** terms.

What is a linear model?

What does it mean for a model to be linear? What is it linear in terms of? Generally speaking, the concept of a **linear operation** (or linear *operator*) is that it maps **linear combinations of vectors** to **linear combinations of the operation applied to each vector**:

$$\mathcal{L}(av_1 + bv_2) = a\mathcal{L}(v_1) + b\mathcal{L}(v_2)$$

for some linear operation $\mathcal{L}$, any scalars a, b, and any two vectors v_1, v_2 to which $\mathcal{L}$ can be applied (note: I do *not* bold my vectors!).

This is quite abstract! More concretely, any (finite-dimensional) linear operator can be **represented by matrix multiplication**. Note, by the rules of linear (matrix) algebra:

$$A(av_1 + bv_2) = aAv_1 + bAv_2$$

so clearly matrix multiplication is linear in this sense. Importantly, in **statistics and machine learning**, the term **linear** usually means **linear in the unknown parameters (weights, coefficients)** of the model.

This can be shown to mean our **target** y is expressed as a **linear combination** of (potentially nonlinear) **feature vectors** $F_i = f_i(x_i)$, each of which may be a function of some **raw feature vectors** x_i:

$$y \approx F_1 w_1 + F_2 w_2 + \cdots + F_n w_n$$

where $w_i, i = 1, \ldots, n$ are the (scalar) model parameters (or weights in neural network terminology).

To see how this can put this into matrix form we ...

Linear models: matrix form

... first identify the $m \times n$ feature matrix F as the matrix having n columns, each given an m-dimensional F_i, i.e.:

$$F = [F_1 \mid \ F_2 \mid \ ... \mid F_n]$$

and collect the weights into a vector w where

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = w_1 e_1 + w_2 e_2 + ... w_n e_n$$

where the e_i are unit vectors with zeros in all entries except the ith, where they are one.

$$Fw = w_1 F e_1 + w_2 F e_2 + \cdots + w_n F e_n = w_1 F_1 + w_2 F_2 + \cdots + w_n F_n$$

where the first equality follows from linearity and the second since $Fe_i = F_i =$ the ith column of F (exercise: verify!).

This actually demonstrates a **very useful alternative (but equivalent) perspective on matrix multiplication in linear algebra:** ***a matrix times a vector is equal to a linear combination of the matrix columns with coefficients given by the vector entries***.

This is all a long way of writing that our linear model takes the form

$$y \approx Fw$$

for target y, feature matrix F, and vector of parameters (weights) w. The matrix F is often written as X and called the **design matrix**, giving

$$y \approx Xw.$$

Example

Consider the following physics model of a **projectile** launched from a height h_0 at velocity v_0 subject to gravity but neglecting air resistance:

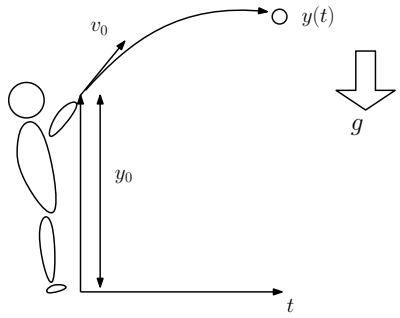
$$\frac{d^2y}{dt^2} = -g$$

with initial conditions

$$y(0) = h_0, \ \ \frac{dy}{dt}(0) = v_0.$$

This has solution

$$y = h_0 + v_0 t - \frac{g}{2}t^2$$



Now, suppose we observe (potentially noisy) data $(t,y)_1, (t,y)_2, ..., (t,y)_m = (t_1, y_1), (t_2, y_2), ..., (t_m, y_m)$ but **do not know** y_0**,** v_0**, or** g.

To **predict** (using our physics knowledge!) the (unobserved) height y_{m+1} at a new (given) time point t_{m+1} we can do the following:

- Determine the parameters y_0 and v_0 in the physics-based model above that best fit the given 'training' data.
- Use these and $t = t_{m+1}$ to predict y_{m+1}.

Example continued

We can consider our observations as approximate (due to noise) instances of our model:

$$\begin{aligned} y_1 &\approx h_0 + v_0 t_1 - \frac{g}{2} t_1^2 \\ y_2 &\approx h_0 + v_0 t_2 - \frac{g}{2} t_2^2 \\ &\vdots \\ y_m &\approx h_0 + v_0 t_m - \frac{g}{2} t_m^2. \end{aligned}$$

This can be written in matrix form

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \approx \begin{bmatrix} 1 & t_1 & -\frac{1}{2}t_1^2 \\ 1 & t_2 & -\frac{1}{2}t_2^2 \\ \vdots & \vdots & \vdots \\ 1 & t_m & -\frac{1}{2}t_m^2 \end{bmatrix} \begin{bmatrix} h_0 \\ v_0 \\ g \end{bmatrix}$$

which can be verified either by the usual 'rows of matrix times column vector' or 'linear combination of matrix columns' interpretations of matrix multiplication.

Thus **this model is linear in the unknown parameters** h_0, v_0, g and the feature/design matrix has columns that are (potentially) nonlinear functions of the raw feature t.

The shape of linear models: well-, over- and under-determined linear systems

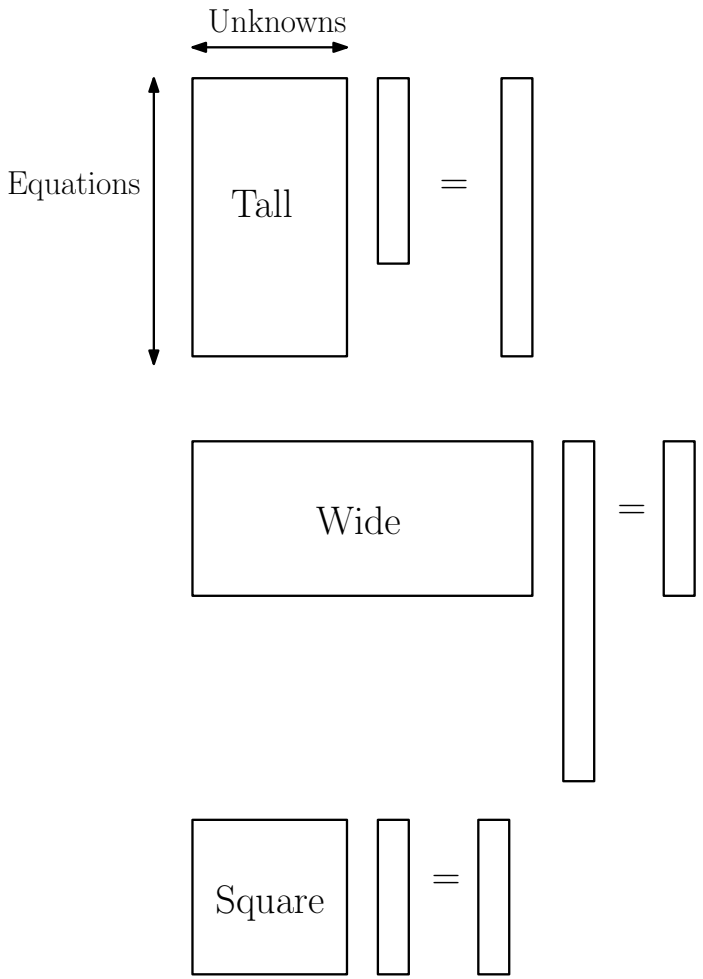
In our projectile case we might collect many more than four data points. However, our approximate physics-based solution (neglecting noise) is a quadratic, which only requires three data points to determine. If we have more or less than three points our matrix equation will involve a **non-square** matrix! How do we solve this??

Firstly, note that because we expect there to be some noise in our solution, we only expect the matrix equation to hold approximately. Secondly, there is a well-defined approach to solving non-square systems of equations: **least squares**! Let's first introduce some terminology, to be discussed in more detail:

- **Tall** systems (with linearly independent columns…) correspond to the **over-determined**, more-equations-than-unknowns case; a standard solution will not typically exist, but a **least squares** solution will.
- **Wide** systems (with linearly independent rows…) correspond to the **under-determined**, less-equations-than-unknowns case; a standard solution may exist but there will typically be many of them, i.e. the solution is **not-unique**. There is, however, a unique **least norm** solution.
- **Square** systems (with linearly independent rows and columns) corresponds to a **well-determined** system, same-equations-as-unknowns case; this is **the exception not the norm** in statistics and machine learning!

Graphically, we get ...

The shape of linear models: well-, over- and under-determined linear systems



Solving tall linear systems

The idea of solving a **tall** or over-determined system is, rather than solve it exactly, consider the **residual**

$$r = y - Xw$$

and minimise the size of this. We have used X to denote the feature/design matrix. Typically we minimise the **sum of squared residuals**, i.e. the Euclidean norm (also called the L_2-norm) of the r vector

$$\min_w ||y - Xw||^2.$$

You may have seen that the solution to this, assuming X has linearly independent columns can be given explicitly by

$$w = X^+ y = (X^T X)^{-1} X^T y$$

where X^+ is called the (Moore-Penrose) pseudo-inverse. This gives us a **best approximate solution** in the least-squares sense. If a true inverse exists it will reduce to this (exercise: show this!). Note that **this particular explicit expression for the pseudo-inverse only holds for the tall/square case**, not, e.g., the wide case.

Packages like NumPy and PyTorch have in-built functions for computing this solution. These are based either on calling a **least squares solver**:

```
w = torch.linalg.lstsq(X,y)
```

or **explicitly computing and then applying pseduo-inverse**:

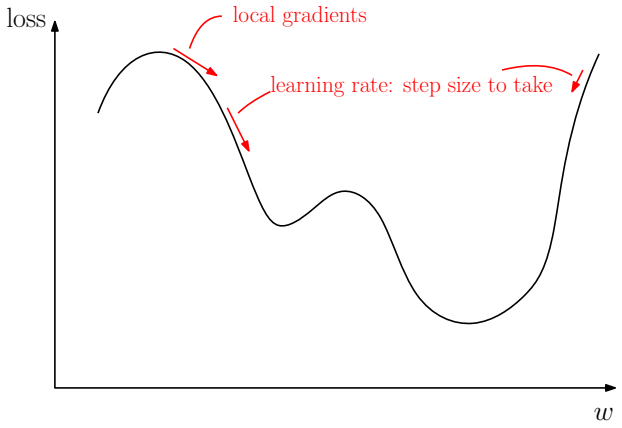
```
w = torch.linalg.pinv(X)y
```

.

Both should give the same answer up to rounding errors, but the former is usually the most efficient and stable!

Solving tall linear systems: gradient descent

In PyTorch we can also solve this directly, *though much less efficiently compared to specialised linear solvers*, by using **gradient descent** optimisation. We've mentioned this in the last chapter, and will go into more detail when we discuss neural networks properly, but the figure below gives the general idea:



('weight/parameter space')

Each time we call `optimizer.step()` we update the parameters by (for gradient descent) moving in the direction of the local gradient of the loss, by an amout proportional to the **learning rate** (i.e. step size!). PyTorch typically uses a form of **stochastic** gradient descent (SGD), which we will discuss properly later.

Solving tall linear systems: example

Here compare specialised solver solutions and gradient descent.

```
import torch
# True parameters
w_true = torch.tensor([10,10,9.81], dtype=torch.float32)

# Design matrix X (m x n)
X = torch.tensor([
    [1.0, 0.1, -0.5*0.1**2],
    [1.0, 0.5, -0.5*0.5**2],
    [1.0, 0.8, -0.5*0.8**2],
    [1.0, 1.0, -0.5*1.0**2],
    [1.0, 1.1, -0.5*1.1**2],
    [1.0, 2.0, -0.5*2.0**2],
], dtype=torch.float32)

# Observation vector y (m-dimensional)
y = X@w_true

# Add normal noise with mean 0 and standard deviation 1.0
torch.manual_seed(42)
noise = torch.randn(len(y)) * 1.0
y_noisy = y + noise

# Solution using lstsq
lstsq_result = torch.linalg.lstsq(X, y_noisy)
w_lstsq = lstsq_result.solution

# Solution using pseudoinverse
w_pinv = torch.linalg.pinv(X) @ y_noisy

# Solution using gradient descent
```

```
w = torch.zeros(X.shape[1], requires_grad=True)

# Loss function, sum square residuals
def loss_fn(w):
    return torch.norm(y_noisy - X @ w) ** 2

# Optimizer
learning_rate = 0.005 #had to play with
optimizer = torch.optim.SGD([w], lr=learning_rate)

# Gradient Descent Loop
num_iterations = 10000 #quite large!
for i in range(num_iterations):
    optimizer.zero_grad()
    loss = loss_fn(w)
    loss.backward()
    optimizer.step()

# Create copy of x that doesn't track gradients anymore
w_gd = w.detach()

# Comparison
print("\nUsing torch.linalg.lstsq:")
print(w_lstsq)
print("\nUsing torch.linalg.pinv:")
print(w_pinv)
print("\nUsing Gradient Descent:")
print(w_gd)
print("\nTrue:")
print(w_true)
```

```
Using torch.linalg.lstsq:
tensor([10.5745,  8.7996,  9.0290])
```



```
Using torch.linalg.pinv:
tensor([10.5745,  8.7996,  9.0290])

Using Gradient Descent:
tensor([10.5747,  8.7990,  9.0285])

True:
tensor([10.0000, 10.0000,  9.8100])
```

Note: the gradient descent solution is very dependent on the learning rate and number of iterations (which needs to be quite large here!), even for such a simple problem. It is not generally a good idea to use this for such a simple problem, but turns out it is a more generalisable approach for more complex problems.

There are also many sophisticated tricks and different optimisers we can use in PyTorch. Here we have used the simplest naive approach.

Solving wide linear systems

In the **wide** or under-determined case there will typically be many solutions to our linear system. One approach to obtaining a unique solution is to take the 'smallest' solution in the least squares sense, i.e. solve

$$\min_w ||w||^2$$

subject to

$$w \text{ is a solution to } y = Xw$$

The idea is **among the possible solutions** to the main equations, we choose the **smallest** one, i.e. the one closest to zero in terms of Euclidean distance (L_2 norm).

When the rows of X are linearly independent we can obtain the explicit solution

$$w = X^+ y = X^T (XX^T)^{-1} y.$$

Note that we again have the (Moore-Penrose) pseudo-inverse, but that this now has a different expression for wide systems.

Regardless of the conditions, we can again call `torch.linalg.lstsq` or `torch.linalg.pinv` and they will give the correct pseudo-inverse. More **generally** (regardless of whether tall or wide or conditions on linear independence), these solve the **two stage optimisation problem**

$$\min_w ||w||^2$$

subject to

$$w \text{ is a solution to } \min_w ||y - Xw||^2$$

Now the idea is that **among the (possibly approximate!) least squares solution, we choose the one with minimum norm.**

This idea is **automatically** implemented in `torch.linalg.lstsq` and `torch.linalg.pinv`. In contrast, formulating this ideas as a simple gradient descent problem is **harder** in general. Let's look at an approximate approach!

Solving linear systems in general

As mentioned, the Moore-Penrose pseudo-inverse can be used to solve linear systems in general, whether tall, wide, or square. This pseudo-inverse can be characterised either in terms of a two-stage optimisation problem or a set of four algebraic equations.

Though beyond the scope of this course, the algebraic characterisation is that the pseudo-inverse of X, X^+ should satisfy

$$\begin{aligned} XX^+X &= X \\ X^+XX^+ &= X^+ \\ (X^+X)^T &= X^+X \\ (XX^+)^T &= XX^+ \end{aligned}$$

Alternatively, to implement the two-stage optimisation idea approximately, we can instead solve

$$\min_w ||y - Xw||^2 + \lambda ||w||^2$$

for a *small* **regularisation parameter** λ. Making this sufficiently small ensures

- The least squares data fit term dominates the weight fitting
- Once this is approximately optimised, the residual will be small and the small penalty (regularisation) term will also be minimised, leading to a small weight solution.

This approach gives the pseudo-inverse solution $w = X^+ y$ as $\lambda \to 0$ for $\lambda > 0$. However, non-zero λ can in general have a non-trivial – and even good! – effect on the solution.

Example wide system

```
import torch
# True parameters
w_true = torch.tensor([10,10,9.81], dtype=torch.float32)

# Design matrix X (m x n), dropping some data points
X = torch.tensor([
    [1.0, 0.1, -0.5*0.1**2],
    [1.0, 2.0, -0.5*2.0**2],
], dtype=torch.float32)

# Observation vector y (m-dimensional)
y = X@w_true

# Add normal noise with mean 0 and standard deviation 1.0
torch.manual_seed(42)
noise = torch.randn(len(y)) * 1.0
y_noisy = y + noise

# Solution using lstsq
lstsq_result = torch.linalg.lstsq(X, y_noisy)
w_lstsq = lstsq_result.solution

# Solution using pseudoinverse
w_pinv = torch.linalg.pinv(X) @ y_noisy

# Solution using gradient descent
w = torch.zeros(X.shape[1], requires_grad=True)

# Loss function, sum square residuals + penalty
lam = 0.00001 #lambda param: made it up!
def loss_fn(w):
```



```
    return (torch.norm(y_noisy - X @ w) ** 2 +
                       lam*torch.norm(w) ** 2)

# Optimizer
learning_rate = 0.0005 #had to play with
optimizer = torch.optim.SGD([w], lr=learning_rate)

# Gradient Descent Loop
num_iterations = 10000 #quite large!
for i in range(num_iterations):
    optimizer.zero_grad()
    loss = loss_fn(w)
    loss.backward()
    optimizer.step()

# Create copy of x that doesn't track gradients anymore
w_gd = w.detach()

# Comparison
print("\nUsing torch.linalg.lstsq:")
print(w_lstsq)
print("\nUsing torch.linalg.pinv:")
print(w_pinv)
print("\nUsing Gradient Descent:")
print(w_gd)
print("\nTrue:")
print(w_true)
```

```
Using torch.linalg.lstsq:
tensor([11.2546,  0.3671,  0.7400])

Using torch.linalg.pinv:
tensor([11.2546,  0.3671,  0.7400])
```

```
Using Gradient Descent:
tensor([11.2520,  0.3677,  0.7391])

True:
tensor([10.0000, 10.0000,  9.8100])
```

The solutions are quite a lot worse for the wide system than for the tall system (relative to the true values), as you might expect from having less data to inform our estimates, though they are very close to each other. This latter fact means we have (apparently) successfully implemented the pseudo-inverse using gradient descent (for this case and this choice of parameters…)

Side note: *It turns out that in many cases we can get 'simple' or small solutions from the nature of the* ***optimiser*** *itself, rather than the loss function,* ***without explicit regularisation****. That is, we can simply use e.g. (stochastic) gradient descent to minimise the sum of square errors for the data and hope it 'naturally' finds a 'simple' but non-unique minima during its iterative solution procedure, usually using special rules for stopping the optimisation process ('early stopping'). Try it here! However, in our case we are interested in how to* ***introduce explicit regularisation*** *and* ***control*** *the types of bias introduced, so here we include it.*

But how well have we fit the (two!) data points?

Estimation and prediction

As mentioned, in order to **predict** for a new time point using our linear model, we *first* solve for the estimated parameters (weights), and *then* propagate these forward to our desired prediction. For a linear model this corresponds to evaluating the value of the new row

$$y_{m+1} = x_{m+1}^T \hat{w}$$

where x^T is a new row vector of feature values at the time point for the prediction, generated according to the same recipe used to construct X at the training points, and now we plug-in the estimate $\hat{w} = X^+ y$. For example, for the projectile we have

$$[y_{m+1}] \approx \begin{bmatrix} 1 & t_{m+1} & -\frac{1}{2}t_{m+1}^2 \end{bmatrix} \begin{bmatrix} \hat{h}_0 \\ \hat{v}_0 \\ \hat{g} \end{bmatrix}$$

We can also consider the 'fitted' (within sample prediction) values of y. For a tall system this gives

$$\hat{y} = X\hat{w} = XX^+ y = Hy$$

Here the matrix $H = XX^+$ is called the 'hat' matrix and it is a 'projection' matrix in data space. Projection matrices are defined by $HH = H$, i.e. applying them twice does the same thing as applying them once. It is called the 'hat matrix' because it 'puts the hat on' y. You can think of is as a 'smoothing' matrix that maps noisy data to the mean. If we include 'in-between points' in X as well as the original observed data we can get our 'interpolation'-style predictions (similarly for 'out-of-sample' data points). The form of X is the same, we just plug-in the time points we want to predict at to get the rows.

Let's consider an example! We will use the weight vectors computed using linear least squares in both the tall and wide cases.

Example: constructing X for prediction

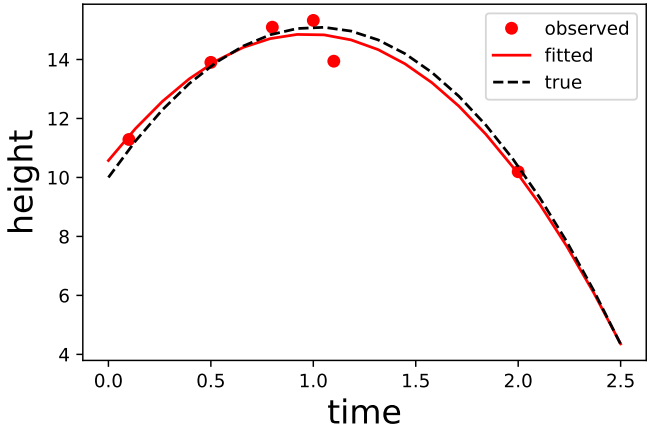
The following function allows us to construct a design matrix to make predictions at any point of interest:

```
def prediction_design_matrix(t_rows):
    # make columns separately
    one_col = torch.ones(t_rows.shape[0], 1)
    time_col = t_rows.view(-1, 1)
    quad_col = -0.5 * (t_rows ** 2).view(-1, 1)

    # Concatenate the columns to form the design matrix
    X = torch.cat([one_col, time_col, quad_col], dim=1)
    return X
```

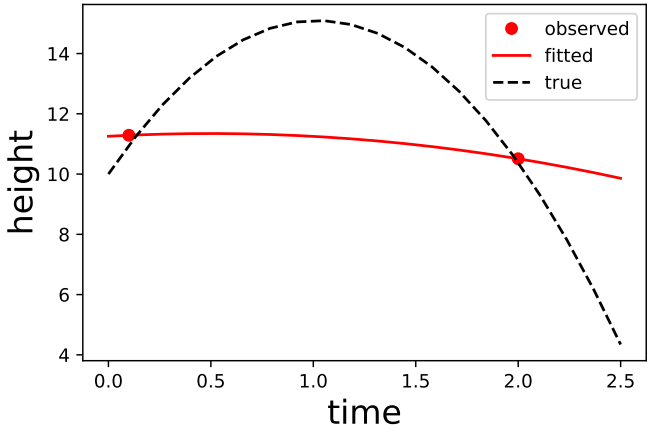
Using this in the tall and wide cases, respectively, gives ...

Example

Tall:



Wide:



Issues

It seems we have a general approach to estimating and predicting with linear models! Unfortunately, however, we have an issue: in the **wide-system** case, our parameters estimates, and often our predictions to, can be **very biased and also unstable** due to the small amount of data and large number of possible approximate solutions that fit the data.

Importantly, large, **deep neural networks** and related modern machine learning models are **analogous to wide systems**: we have **very many (sometimes billions!) of parameters/weights** to estimate from (typically) comparably less data.

As discussed, some form of 'bias' in the general sense is essentially **necessary** to make predictions from the data, particularly in the 'wide' setting.

However, here our bias is of one particular form – could we do better? And does the amount of bias required relate to the amount of data we have? And can we better choose the bias to get more stable solutions?

Regularisation revisited

An interesting approach to further stabilising solutions is to **not** target the $\lambda \to 0$ limit (at least for finite amounts of data), and instead choose a non-zero value that provides **additional penalisation** towards $w = 0$, i.e. we **sacrifice some fit to the data** in order to get less variable solution. In statistics this idea is sometimes called the 'bias-variance' trade-off: often, we can deliberately introduce bias in order to 'reduce the variance' of our solutions, i.e. increase their stability (see ENGSCI 760 and ENGSCI 721).

Thus we solve

$$\min_w ||y - Xw||^2 + \lambda||w||^2$$

for some other choice of λ, not necessarily as small as possible.

This particular form of a linear model plus penalty of this type is often called **ridge regression** in statistics and machine learning. It's referred to as (a type of) **Tikhonov regularisation** in **inverse problems** (see ENGSCI 721).

We will look at methods for choosing λ shortly. First, let's look at how, for *linear* models, we can implement this efficiently when we want to use `torch.linalg.lstsq` or `torch.linalg.pinv` rather than direct gradient descent.

Stacking

Consider the squared Euclidean norm (L_2 norm) of some vector x:

$$||x||^2 = \left|\left| \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \right|\right|^2 = x_1^2 + x_2^2 + \cdots + x_n^2.$$

Note that we can partition x into subvectors y, z and write x in terms of **stacked** vectors

$$||x||^2 = \left|\left| \begin{bmatrix} y \\ z \end{bmatrix} \right|\right|^2 = ||y||^2 + ||z||^2$$

In our case, this means we can write

$$||y - Xw||^2 + \lambda||w||^2 = \left|\left| \begin{bmatrix} y - Xw \\ \sqrt{\lambda} w \end{bmatrix} \right|\right|^2$$

We can further write the right-hand side as the **augmented** system

$$\left|\left| \begin{bmatrix} y - Xw \\ \sqrt{\lambda} w \end{bmatrix} \right|\right|^2 = \left|\left| \begin{bmatrix} y \\ 0 \end{bmatrix} - \begin{bmatrix} X \\ \sqrt{\lambda} I \end{bmatrix} w \right|\right|^2$$

i.e.

$$\left|\left| \tilde{y} - \tilde{X} w \right|\right|^2$$

where

$$\tilde{y} = \begin{bmatrix} y \\ 0 \end{bmatrix}, \quad \tilde{X} = \begin{bmatrix} X \\ \sqrt{\lambda} I \end{bmatrix}$$

So to solve the penalised linear least squares problem we can solve the standard linear least squares problem in **augmented** form:

$$\min_w \left|\left| \tilde{y} - \tilde{X} w \right|\right|^2$$

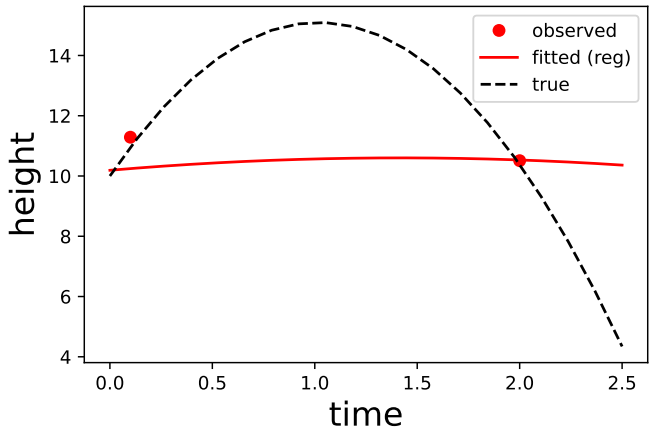
Example

```python
import torch

# True parameters
w_true = torch.tensor([10, 10, 9.81], dtype=torch.float32)

# Design matrix X (m x n), with some data points omitted
X = torch.tensor([
    [1.0, 0.1, -0.5 * 0.1**2],
    [1.0, 2.0, -0.5 * 2.0**2],
], dtype=torch.float32)

# Observation vector y (m-dimensional)
y = X @ w_true

# Add normal noise with mean 0 and standard deviation 1.0
torch.manual_seed(42)
noise = torch.randn(len(y)) * 1.0
y_noisy = y + noise

# Regularisation parameter (lambda)
lambda_reg = torch.tensor(0.1, dtype=torch.float32)

# Augment the design matrix and observation vector
X_aug = (torch.cat([X, torch.sqrt(lambda_reg) *
            torch.eye(X.shape[1])], dim=0))
y_aug = torch.cat([y_noisy, torch.zeros(X.shape[1])])

# Solution using least squares with regularisation
lstsq_result_reg = torch.linalg.lstsq(X_aug, y_aug)
w_reg = lstsq_result_reg.solution
```

```
# Output the regularised solution
print(f"Reg. least squares solution: {w_reg}")
```

```
Reg. least squares solution: tensor([10.1896,  0.5805,  0.4094])
```

Fitted:



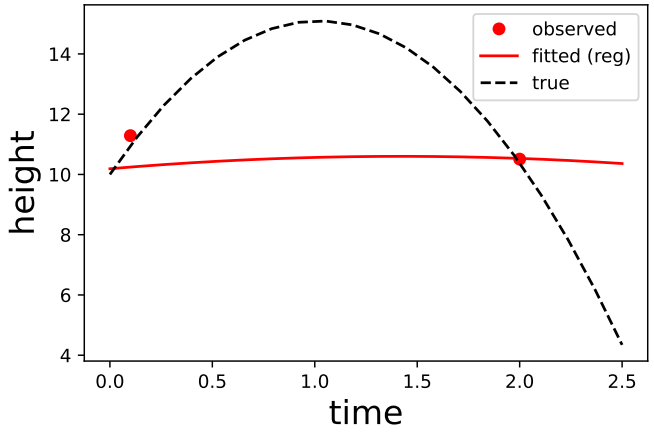
Note that we have sacrificed some data fit for a 'simpler solution'. Here the result is not really so good, but the idea of engineering/physics-informed machine learning (that we will see next week) is to **penalise towards physical/engineering models at the possible expense of some pure data fitting performance**. The idea is that we can generalise **beyond** the training data in this way.

Solving in PyTorch with gradient descent

Nothing much is changed in our PyTorch implementation – we just use a different λ.

```
# Solution using gradient descent
w = torch.zeros(X.shape[1], requires_grad=True)

# Loss function, sum square residuals + penalty
lam = 0.1 #lambda param: not as small anymore
def loss_fn(w):
    return (torch.norm(y_noisy - X @ w) ** 2 +
                       lam*torch.norm(w) ** 2)

# Optimizer
learning_rate = 0.0005 #had to play with
optimizer = torch.optim.SGD([w], lr=learning_rate)

# Gradient Descent Loop
num_iterations = 10000 #quite large!
for i in range(num_iterations):
    optimizer.zero_grad()
    loss = loss_fn(w)
    loss.backward()
    optimizer.step()

# Create copy of x that doesn't track gradients anymore
w_gd = w.detach()

# Print
print("\nUsing Gradient Descent:")
print(w_gd)
print("\nTrue:")
print(w_true)
```

```
Using Gradient Descent:
tensor([10.1887,  0.5807,  0.4091])

True:
tensor([10.0000, 10.0000,  9.8100])
```



The result is very similar to what we found with the specialised solver.

Choosing regularisation parameter

Coming soon!

Part IV

Hybrid linear models

Hybrid models

Hybrid models combine **data-driven** models with **theory-based** (e.g., scientific/engineering) models to improve overall model performance and generalisation beyond the training data. Here we consider hybrid models with losses of the form:

$$\text{overall loss} = \text{data loss} + \lambda \times \text{model loss}.$$

The 'data loss' term is a standard (training) data fit term such as sum of squared errors. The 'model loss' term measures the distance of a proposed model from some theoretical (engineering, physics, biology etc.) model. The **regularisation parameter** λ, a type of **hyperparameter**, determines the balance between agreement with the data and agreement with the theoretical model. The term **'regularisation'** in this context essentially refers to a form of **soft constraint** that (deliberately) *biases* the model towards desirable solutions (stable, physical, interesting etc.) in a way that **goes beyond the available data**. Hence here it contributes to the overall loss above and beyond the data loss. Unlike a hard constraint, which strictly enforces a condition, a soft constraint allows for some deviation but discourages larger deviations.

Note: Here we are looking at **explicit regularisation** via a penalty term. A common assumption in standard machine learning, in contrast, is that **implicit** regularisation via the training method, in particular forms of **stochastic gradient descent**, typically provides sufficient regularisation even for very large models like deep neural networks, and that more traditional penalty terms are often not required. The downside is that we have no guarantees on the types of bias we are introducing. Areas such as **scientific/engineering/physics-informed machine learning** hence still typically (at least currently) use explicit regularisation to ensure the biases are of the desired type.

Linear data models: Using basis functions

We have already considered linear models in the previous section. As discussed there, linear models represent the response variable as a **linear combination** of columns of the **design** or **feature matrix**:

$$y \approx Xw$$

for response vector y, design matrix X, and weight (parameter) vector w.

When dealing with **dynamic**, continuous-time physical systems it can be useful to use pre-chosen **smooth functions** as our features. When expanding our response as a linear combination of such functions we often refer to these as **basis functions** (as in e.g., Fourier series):

$$y(t) \approx \sum_{j=1}^{n} b_j(t) w_j.$$

Although these functions can in principle be evaluated for any time point, we typically assume we have a discrete set of observation times $t_i, i = 1, \ldots, m$. This leads to a system, defined at the observations, in the same form as our design matrix expression:

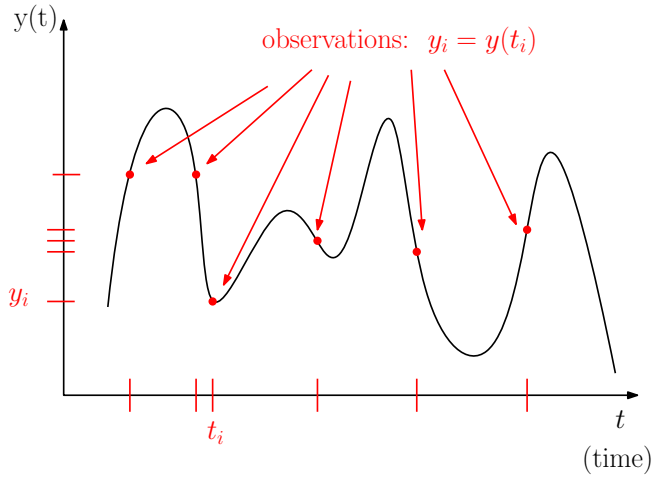
$$y \approx Bw$$

where $y_i = y(t_i)$, w is the weight vector with jth entry w_j, and the jth column of B is a vector with entires $b_j(t_i)$. That is, here our design matrix is $X = B$.

We will consider an example of such basis functions shortly. Firstly, however, we need to consider two different 'time grids' of interest.

Underlying time grid vs. observation time grid

If we are dealing with a dynamic physical system it is often reasonable to suppose the underlying dynamics are either continuous in time or at least defined for a **fine time grid** that is finer than our **observation time grid** (similar ideas apply for spatial/spatiotemporal systems). For example, the underlying dynamics might be solved with an ODE solver that discretises time into a grid of p points. In contrast, we may only have actual data **observations** of the system at $m \ll p$ points. The observation grid may or may not be regular. See the figure below.



When considering physics-based regularisation, it can be useful to introduce both grids explicitly, along with a **linear** mapping between them, represented by an **observation matrix** (or an observation operator).

Observation matrix

Consider a **standard basis vector** (not to be confused with our basis functions above!):

$$e_j = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow j\text{th entry.}$$

The effect of multiplying an m by n matrix M by e_j^T on the left is

$$e_j^T M = \text{row}_j(M)$$

i.e. it picks out the jth row of M. (Note: multiplying by e_j on the right picks out the jth column – recall the 'linear combination of columns' interpretation of matrix multiplication).

If we recall that, for design matrices, **each row corresponds to an observation**, we can model the process of taking a 'coarser' set of observations of a vector y defined on a fine time grid by

$$y_{\text{obs}} = A_{\text{obs}} y$$

where y_{obs} is the vector of observed points and A_{obs} is the 'observation matrix' that has a row e_j^T for any grid point j that we have an observation for.

Example? Example!

Observation matrix example

```
import torch
y = torch.arange(10,20, dtype=torch.float32)
obs_indices = [0,4,9]
A_obs = torch.eye(len(y),dtype=torch.float32)[obs_indices]
y_obs = A_obs@y
print('Fine-scale y:')
print(y)
print('\n')
print('Observation matrix:')
print(A_obs)
print('\n')
print('Observed y:')
print(y_obs)
```

```
Fine-scale y:
tensor([10., 11., 12., 13., 14., 15., 16., 17., 18., 19.])


Observation matrix:
tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])


Observed y:
tensor([10., 14., 19.])
```

Observation matrix and basis/feature matrix

If our underlying fine-grid variable y is modelled as a finely-discretised basis function expansion

$$y \approx Bw,$$

we can model our coarse-grid observations as

$$y_{\text{obs}} = A_{\text{obs}} y \approx A_{\text{obs}} B w = B_{\text{obs}} w$$

where $B_{\text{obs}} = A_{\text{obs}} B$ is the matrix containing the basis functions evaluated at the observation points. That is, A_{obs} picks out the rows of the *matrix* B corresponding to observations, just like it does with vectors.

Whether or not we explicitly model the two different time grids depends on the application.

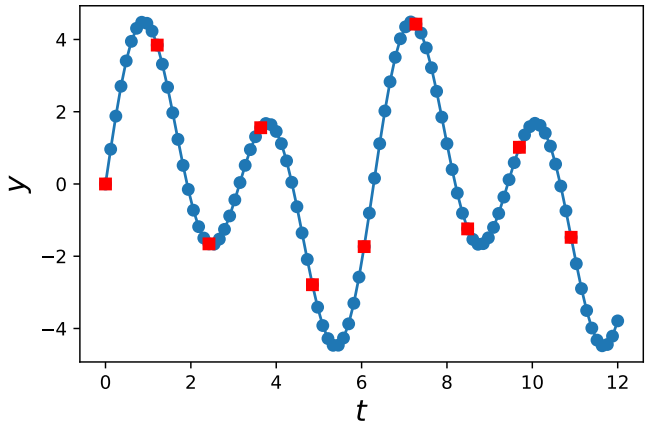
For standard, 'pure' data modelling we usually work with B_{obs} directly and don't worry about introducing an explicit observation matrix; however, **when introducing a theory/physics/engineering-based model as well, we usually want to enforce this model on the fine-time grid** rather than just the coarse observation grid.

The basic idea is that **we typically assume theory-based model holds at all time points, not just those we've observed**. This allows us to provide predictions for, or constraints on model behaviour at, points we haven't observed. Of course, sometimes we take the grids to be the same for simplicity.

Example

Suppose our B matrix consists of two columns, $\sin(t)$ and $\sin(2t)$, evaluated on a fine time grid t_i, of points between $t_1 = 0$ and $t_p = 12$, for $p = 100$. This means B has 100 rows and 2 columns. Then, suppose we take as observations every 10th point, starting from the first, giving 10 observation indices.

```
import matplotlib.pyplot as plt
t = torch.linspace(0,12,100, dtype=torch.float32)
B = torch.stack([torch.sin(t),torch.sin(2*t)],dim=1)
w = torch.tensor([2,3],dtype=torch.float32) # arbitrary choice
y = B@w
obs_indices = torch.arange(0,100,10, dtype=torch.int)
A_obs = torch.eye(len(y),dtype=torch.float32)[obs_indices]
t_obs = A_obs@t
y_obs = A_obs@y # = A_obs@B@w
```



Here red squares indicate the observed data points and blue dots (including at red square locations) indicate the 'fine' time grid points.

B-Spline basis functions

We can represent a range of smooth functions of time with **splines**, i.e. piecewise polynomials defined by various conditions such as the degree of the polynomials used and the properties at the 'join' points, where these join points are called *knots*. To construct a spline we can use basis functions; here we will use so-called B-Spline basis functions. **For this course we don't need to know much about these other than how to obtain them from a Python library**. However the following facts may be useful:

- Any spline of made up of degree d polynomials (d being the highest power of the component polynomials) can be represented as a linear combination of B-Splines of the same degree.

- Splines are defined by the degree of the polynomials and the location of the internal knots where the polynomials join. The term 'knot' refers to internal knots in general (i.e. excluding boundary points).

- At (non-repeated) knots the spline has $d - 1$ continuous derivatives. E.g. for a piecewise linear spline, i.e. $d = 1$, the first derivative is not continuous at the knots, though the value ('zeroth derivative') is ($d - 1 = 0$ if $d = 1$).

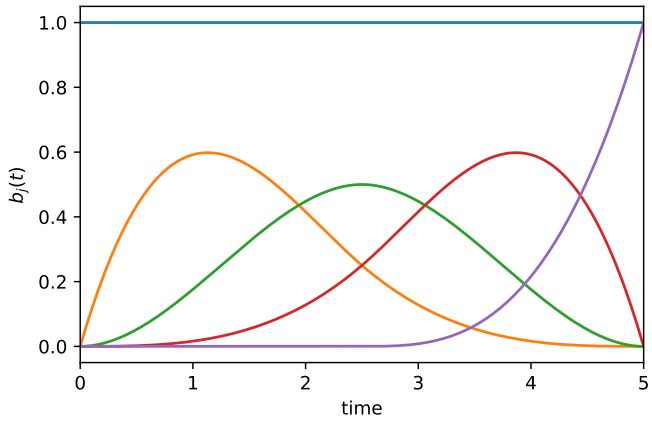
- Many libraries for constructing spline bases take the polynomial degree and either the knot *locations* or a parameter called 'degrees of freedom', ν. This parameter ν is equal to the *number of basis functions* and is given by the degree d plus the number of knots k. Hence, given a desired number of basis functions ν, the number of implied (eqully-spaced) knots will be $\nu - d$.

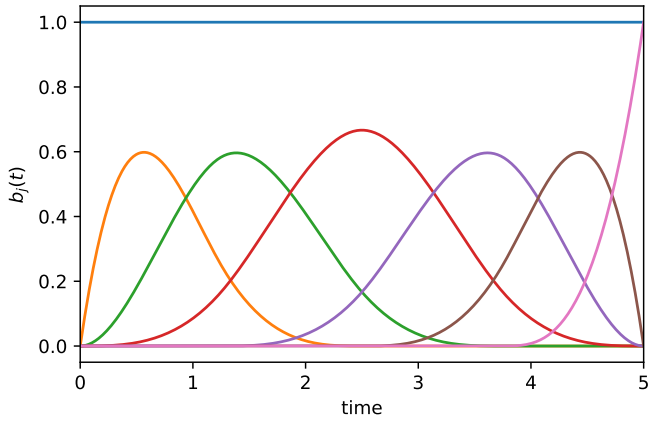
- Many libraries for constructing the 'design matrices' – matrices with basis functions as columns – automatically add a constant vector of all ones. We can explicitly add it with statistical formulae such as '1 + basis functions', or remove it with formulae like 'basis functions − 1'.

B-Spline basis functions

Many languages such as R, Python, MATLAB, Julia etc have B-Splines available. Below is an example of how to obtain B-Spline bases of different sizes, along with an intercept/constant term, using the **Python library Patsy**. This is installed in Google Colab by default.

```
# Four cubic basis functions
# number of implied knots = 4-3 = 1,
# as well as one constant function.
# Number of columns of B is 4 + 1 = 5.
from patsy import dmatrices, dmatrix
# time grid
t = torch.linspace(0., 5., 1000)
# design matrix as tensor
B1 = torch.tensor(
    dmatrix("1 + bs(t, df=4, degree=3)", {"t": t}))

# Six cubic basis functions,
# number of implied knots = 6-3 = 3,
# as well as one constant function.
# Number of columns of B is 6 + 1 = 7.
B2 = torch.tensor(
    dmatrix("1+bs(t, df=6, degree=3)", {"t": t}))
```





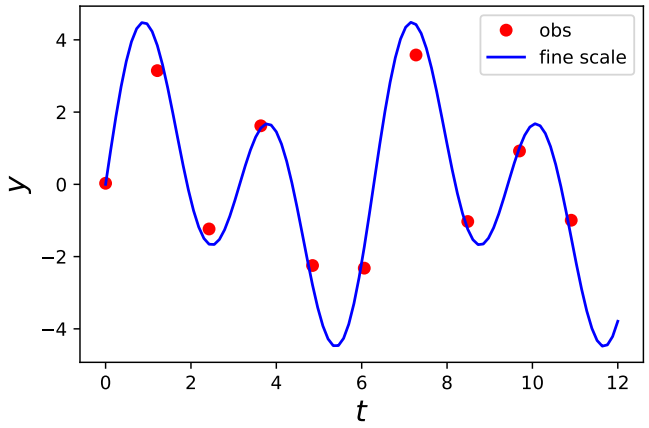

```
print(B1.shape)
print(B2.shape)
```

```
torch.Size([1000, 5])
torch.Size([1000, 7])
```


Example fit

Suppose our true data is a coarse, and noisy version of our `sin` combination used above. Let's fit a B-Spline basis to it with linear least squares. First generate data:

```
t = torch.linspace(0,12,100, dtype=torch.float32)
B = torch.stack([torch.sin(t),torch.sin(2*t)],dim=1)
w = torch.tensor([2,3],dtype=torch.float32)
y = B@w
# observation matrix for observed points
obs_indices = torch.arange(0,100,10, dtype=torch.int)
A_obs = torch.eye(len(y),dtype=torch.float32)[obs_indices]
# noise
torch.manual_seed(44)
noise = torch.randn(A_obs.shape[0]) * 0.5
# extract at observation points and add noise
y_obs = A_obs@y + noise
t_obs = A_obs@t
```



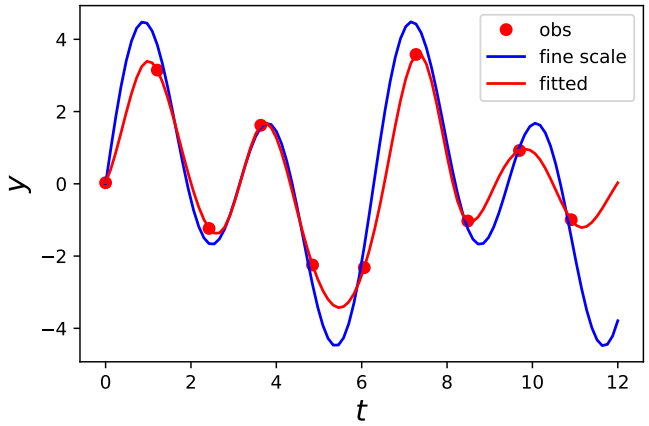
Example fit

Now, let's fit a spline basis to the data.

```
# 15 cubic basis functions,
# number of implied knots = 15-3 = 12,
# as well as one constant function.
# Number of columns of B is 12 + 1 = 7.
B = torch.tensor(
    dmatrix("1+bs(t, df=15, degree=3)", {"t": t}),
    dtype=torch.float32)

# Solution using lstsq.
# We need to include A_obs as that's how we got the data
lstsq_result = torch.linalg.lstsq(A_obs@B, y_obs)
w_lstsq = lstsq_result.solution

# We can then get the implied fit on the fine time grid
y_fitted = B@w_lstsq
```



```
plt.plot(t_obs,y_obs,'ro',label='obs')
plt.plot(t,y,'b',label='fine scale')
plt.plot(t,y_fitted,'r',label='fitted')
plt.xlabel(r'$t$',fontsize=16)
plt.ylabel(r'$y$',fontsize=16)
plt.legend()
plt.show()
```



Note the red shows both the observed data points and the implied fitted model. We typically don't observe the blue directly and the red curve is an *estimate* of the true blue curve. Here the B-Spline basis does a reasonable job given the data is both noisy and relatively sparse.

This model is a **pure data model**. There is no physics or engineering knowledge as such (other than smoothness). It also requires choosing a number of basis functions – not too many, not too few.

How can we include physical information? One way is ...

Derivative matrices

Suppose we want to include physics- or engineering-based knowledge in the form of **approximately-enforced differential equations** into a data model like a spline basis approximation. We can represent derivatives on our fine time grid by using **finite difference** matrices. These map a vector defined on a fine grid (e.g. of points in time or space) to a new vector of **derivatives** of the input vector. For example:

$$D = \frac{1}{\Delta t}\begin{bmatrix} -1 & 1 & 0 & 0 & 0 & ... \\ 0 & -1 & 1 & 0 & 0 & ... \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & 0 & -1 & 1 \end{bmatrix}$$

Note that for a vector y containing values $y_1, y_2, \dots, y_m$ we have

$$Dy = \frac{1}{\Delta t}\begin{bmatrix} y_2 - y_1 \\ y_3 - y_2 \\ \vdots \\ y_m - y_{m-1} \end{bmatrix} = \begin{bmatrix} \frac{y_2 - y_1}{\Delta t} \\ \frac{y_3 - y_2}{\Delta t} \\ \vdots \\ \frac{y_m - y_{m-1}}{\Delta t} \end{bmatrix}$$

which is a vector that contains backward difference approximations to the derivative at all time points starting from the *second* time point.

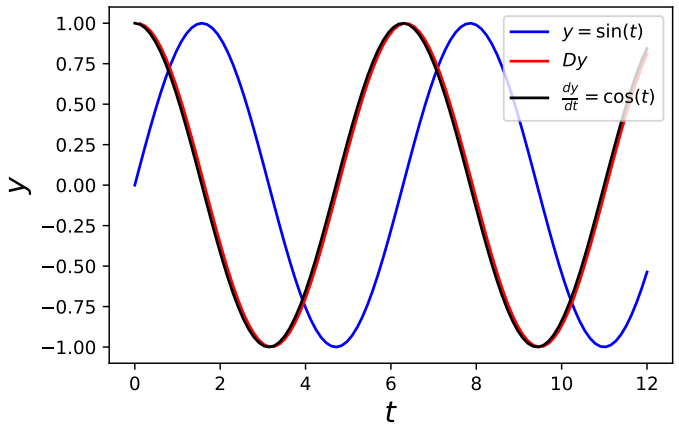
The vector here has **dimension of one less than the input vector**, which corresponds to the idea that a first order differential equation usually needs to be supplemented by an additional **initial condition**, and hence an additional entry in the vector. This is not so important here as the data will play the role of initial conditions.

We can define **second** and **third** (etc) order derivative matrices similarly, and can also obtain these by (carefully) **multiplying the first derivative matrix by itself** the corresponding number of times (we need to be careful of initial/boundary conditions and dimensions as usual). This represents repeated differentiation. We usually lose one row per application of the first derivative matrix, just as e.g. a second-order differential equation requires two initial/boundary conditions.

Example

```
t = torch.linspace(0,12,100, dtype=torch.float32)
dt = t[1]-t[0]; N = len(t)
y = torch.sin(t)
D = torch.zeros(N-1, N)
# Use 'fancy indexing' to assign values to diagonals
D[torch.arange(0, N-1), torch.arange(0, N-1)] = -1/dt
D[torch.arange(0, N-1), torch.arange(1, N)] = 1/dt
print(D*dt) #multiply by dt to get difference matrix part
```

```
tensor([[-1.,  1.,  0.,  ...,  0.,  0.,  0.],
        [ 0., -1.,  1.,  ...,  0.,  0.,  0.],
        [ 0.,  0., -1.,  ...,  0.,  0.,  0.],
        ...,
        [ 0.,  0.,  0.,  ...,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  ..., -1.,  1.,  0.],
        [ 0.,  0.,  0.,  ...,  0., -1.,  1.]])
```



Higher derivative matrices: example

```
# first derivative, backward diff
print(D*dt)
# second derivative, backward diff
print(D[1:,1:]@D*dt**2)
# third derivative, backward diff
print(D[2:,2:]@D[1:,1:]@D*dt**3)
```

```
tensor([[-1.,  1.,  0.,  ...,  0.,  0.,  0.],
        [ 0., -1.,  1.,  ...,  0.,  0.,  0.],
        [ 0.,  0., -1.,  ...,  0.,  0.,  0.],
        ...,
        [ 0.,  0.,  0.,  ...,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  ..., -1.,  1.,  0.],
        [ 0.,  0.,  0.,  ...,  0., -1.,  1.]])
tensor([[ 1., -2.,  1.,  ...,  0.,  0.,  0.],
        [ 0.,  1., -2.,  ...,  0.,  0.,  0.],
        [ 0.,  0.,  1.,  ...,  0.,  0.,  0.],
        ...,
        [ 0.,  0.,  0.,  ...,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  ..., -2.,  1.,  0.],
        [ 0.,  0.,  0.,  ...,  1., -2.,  1.]])
tensor([[-1.0000,  3.0000, -3.0000,  ...,  0.0000,  0.0000,  0.0000],
        [ 0.0000, -1.0000,  3.0000,  ...,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000, -1.0000,  ...,  0.0000,  0.0000,  0.0000],
        ...,
        [ 0.0000,  0.0000,  0.0000,  ...,  1.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  ..., -3.0000,  1.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  ...,  3.0000, -3.0000,  1.0000]])
```


Including derivatives in a linear model

Given a simple differential equation

$$\frac{dy}{dt} = 0,$$

this can be approximated by

$$Dy = DBw = 0$$

where D is the first-order derivative matrix, B is a basis matrix (e.g. B-Splines) and w is the vector of unknown parameters (weights), assuming $y \approx Bw$.

Rather than try to solve this exactly, we can now modify our least squares data loss function to take the form:

$$\min_w ||y_{\text{obs}} - A_{\text{obs}}Bw||^2 + \lambda||DBw||^2.$$

where A_{obs} is our observation matrix, y_{obs} the observed data, and λ is a regularisation (or penalty) parameter determining **how strongly to enforce the differential equation**. Higher λ means higher penalty on deviations from $Dy = 0$ and so the model is enforced more strongly; lower λ works in the opposite direction, allowing the 'pure data model' to dominate the fitting.

The same idea generalises to higher-order differential equations by simply **taking D to be a higher-order derivative matrix**.

Notes:

- We have previously solved essentially this exact problem in the last section. Here the design matrix is $X = A_{\text{obs}}B$ and there is a penalty on w as before, except now the w in the penalty term is weighted by DB. This is a minor difference that makes no real difference to implementation!
- We typically use a penalty term of the form $\hat{\lambda}||\hat{D}Bw||^2$ in practice, where $\hat{D} = Ddt$ is the simple difference matrix and the new regularisation parameter is implicitly related to the old via $\hat{\lambda} = \lambda/dt$. This helps standardise the scale of the regularisation term but is mathematically equivalent to the other form.

Let's look at how to 'stack and solve' these problems!

Stacking

Following the ideas of the previous section, we can write

$$||y_{\text{obs}} - A_{\text{obs}}Bw||^2 + \lambda||DBw||^2 = \left|\left| \begin{bmatrix} y_{\text{obs}} - A_{\text{obs}}Bw \\ \sqrt{\lambda}DBw \end{bmatrix} \right|\right|^2$$

We can further write the right-hand side as the **augmented** system

$$\left|\left| \begin{bmatrix} y_{\text{obs}} - A_{\text{obs}}Bw \\ \sqrt{\lambda}DBw \end{bmatrix} \right|\right|^2 = \left|\left| \begin{bmatrix} y_{\text{obs}} \\ 0 \end{bmatrix} - \begin{bmatrix} A_{\text{obs}}B \\ \sqrt{\lambda}DB \end{bmatrix} w \right|\right|^2$$

where 0 is a **(sub)vector of zeros** with length equal to the number of rows of D. This gives a linear system of the form:

$$\left|\left|\tilde{y} - \tilde{X}w\right|\right|^2$$

where

$$\tilde{y} = \begin{bmatrix} y_{\text{obs}} \\ 0 \end{bmatrix}, \quad \tilde{X} = \begin{bmatrix} A_{\text{obs}}B \\ \sqrt{\lambda}DB \end{bmatrix}$$

So to solve the penalised linear least squares problem we can solve the standard linear least squares problem in **augmented** form:

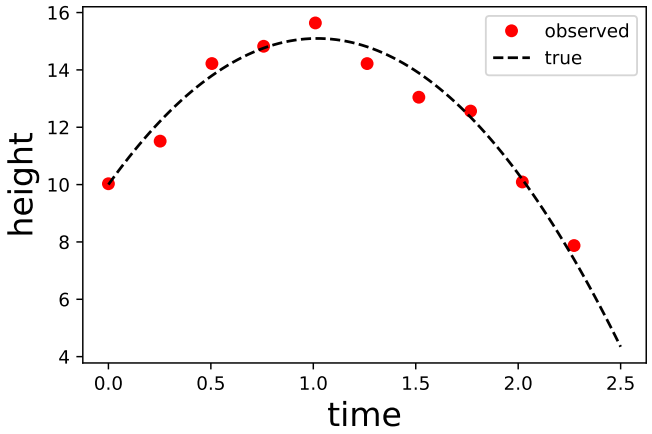
$$\min_w \left|\left|\tilde{y} - \tilde{X}w\right|\right|^2$$

just as before!

Notes:
- This approach works for differential equations giving the discrete form $Dy = 0$,, $Dy =$ constant, or $Dy = Ay$, for D an abitrary derivative matrix and A a *known* matrix.
- Differential equations of the form $Dy = f(y)$ lead to nonlinear systems of equations in this approach; if A is unknown then this is true even for $f(y) = Ay$.
- We can sometimes differentiate the system more times to get our equation in the desired form, however.

Example

Consider our projectile problem with observed data as shown:



We can fit a spline + ODE model by writing our ODE

$$\frac{d^2y}{dt^2} = -g$$

as

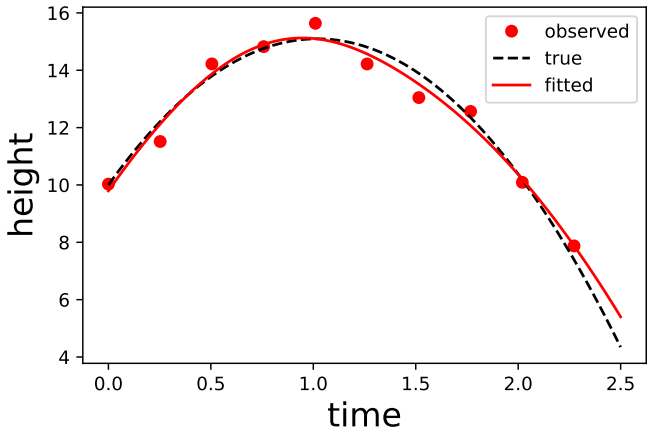
$$\frac{d^3y}{dt^3} = 0$$

which treats g as unknown. We can implement this as follows...

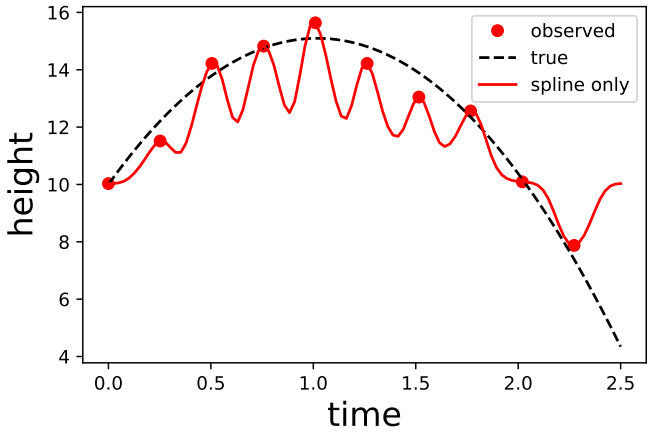
Example continued

```
t_grid = torch.linspace(0,2.5,100)
# spline basis. Can use lots when have diff. eqn!
B = torch.tensor(
    dmatrix("1+bs(t, df=30, degree=3)", {"t": t}),
    dtype=torch.float32)

# observation matrix for observed points
obs_indices = torch.arange(0,100,10, dtype=torch.int)
A_obs = torch.eye(len(t_grid),dtype=torch.float32)[obs_indices]

# first derivative matrix
D = torch.zeros(N-1, N)
# Use 'fancy indexing' to assign values to diagonals
D[torch.arange(0, N-1), torch.arange(0, N-1)] = -1
D[torch.arange(0, N-1), torch.arange(1, N)] = 1
# get third difference matrix
D3 = D[2:,2:]@D[1:,1:]@D

# Augment the design matrix and observation vector
lambda_reg = torch.tensor(1e5, dtype=torch.float32)
X_aug = (torch.cat([A_obs@B, torch.sqrt(lambda_reg) *
            D3@B], dim=0))
y_aug = torch.cat([y_data, torch.zeros(D3.shape[0])])

# solve
lstsq_result = torch.linalg.lstsq(X_aug, y_aug)
w_lstsq = lstsq_result.solution

# Get the implied fit on the fine time grid
y_fitted = B@w_lstsq
```


Example continued



Compare this to the pure spline fit with the same number of basis functions:



Notes

Note that

- The pure spline fit **fits the training data better** (it interpolates it), but **fits the (unseen) test data worse**.
- In contrast, the ODE-penalised fit **fits the training data worse but generalises to unseen data better**.

This is a common phenomenon! For this and related reasons, statisticians and modellers have **traditionally advised against exactly fitting the training data** in the name of avoiding overfitting.

However, the main concern is not so much having 'too good' of a fit to the training data, rather the concern that this **may not be representative of the ability to fit the test data**.

One of the more surprising observations of modern machine learning is there **are** models that **exactly fit the training data ('overfit') and yet generalise well to new data**. One perspective on this is that away from data points the regularisation or structural assumptions dominate, and that it is possible to both fit the training data well and generalise well, if we also have good regularisation/structural assumptions.

In the present course, however, we will typically see the traditional wisdom – of sacrificing some fit to the training data in order to get better fit to unseen data – hold up.

This relates to the topic of...

Choosing regularisation parameters

So far we have pre-specified the regularisation parameter. In both the pseudo-inverse and physics-informed cases we have general guidelines for setting this: for the pseudo-inverse we set it to be 'small but not zero'. For a physics model we set it to be 'large' if we think the model is correct. What about in general? **Can we use the data** to determine a 'good' value?

One issue is that λ controls the ability to fit **unseen** data: it involves a loss term evaluated at points **other than the training data** (e.g. the fine time scale). If we fit its value based just on the training data, and the data model component is capable of fitting the training data exactly, the optimal value will be zero (why?)!

A strategy around this, is to set regularisation/generalisation/hyperparameters like λ based on the ability of the fitted model to **predict data it wasn't trained on**. The simplest way to do this is to **split** our original training dataset into a smaller training data subset and a non-overlapping **validation** subset (you will have seen this idea already!). We fit the model on the training subset for various λ values and use these models to predict the unfitted validation set. We use the λ value that leads to the best predicting model. This **mimics the process of fitting a model to training data and evaluating on test data**, and hence provides information about 'generalisation' ability (under assumptions such as exchangeable or IID data, howevever!).

This approach falls under the umbrella of the family of techniques known as **cross-validation**. More sophisticated variations of cross-validation, such as k-fold cross-validation, repeatedly split the training data into training and validation data subsets and assess the average performance on the validation sets. Note that we typically still have a third, separate **test set** that we don't see until the final model, including hyperparameters, is developed.

Other approaches, popular in **inverse problems** (see ENGSCI 721), include **L-curves** and **discrepancy principles**.

Generalised cross-validation for linear models

Generally, the more complex the model (e.g. deep neural network vs linear model), the simpler the data splitting approach is, as it can be hard/computationally expensive to implement more sophisticated methods.

For **linear models**, however, there is a particularly simple approximation available: **generalised cross-validation**. This is somewhat of a misnomer: it is an *approximation* to what you would obtain by carrying out 'leave-one-out' cross-validation for a linear model.

You **don't need to know the details**, but for completeness, it minimises the following formula (representing generalisation error) to estimate λ:

$$\mathrm{GCV}(\lambda) = \frac{1}{n}\frac{\|(I - H_\lambda)y_{\mathrm{obs}}\|^2}{\left(1 - \frac{\mathrm{Tr}(H_\lambda)}{n}\right)^2}$$

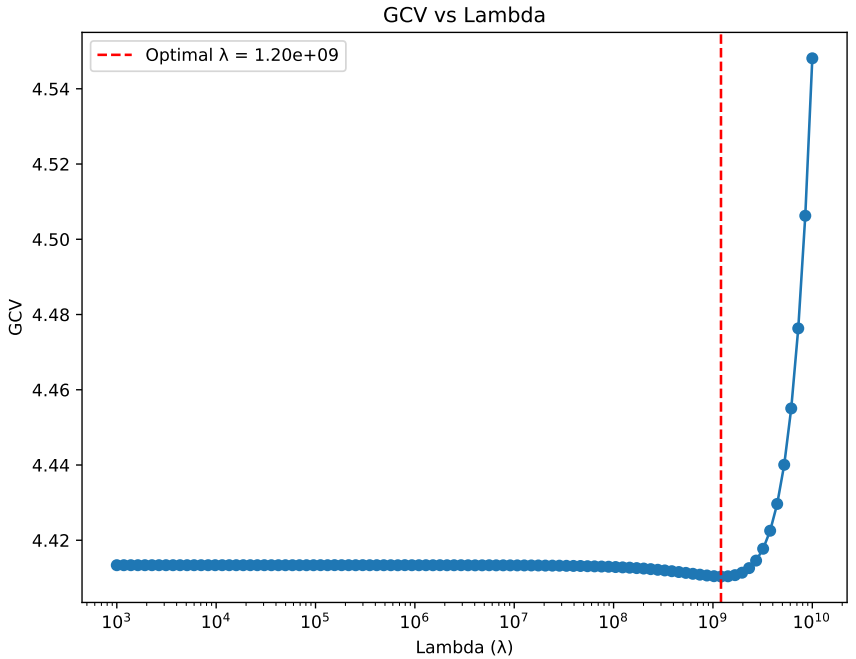
where H_λ is the **hat (smoothing) matrix** associated with the regularised estimate. In our hybrid model example, this is given by:

$$H_\lambda = B_{\mathrm{obs}}\left(B_{\mathrm{obs}}^T B_{\mathrm{obs}} + \lambda B^T D^T D B\right)^{-1} B_{\mathrm{obs}}^T$$

where $B_{\mathrm{obs}} = A_{\mathrm{obs}}B$.

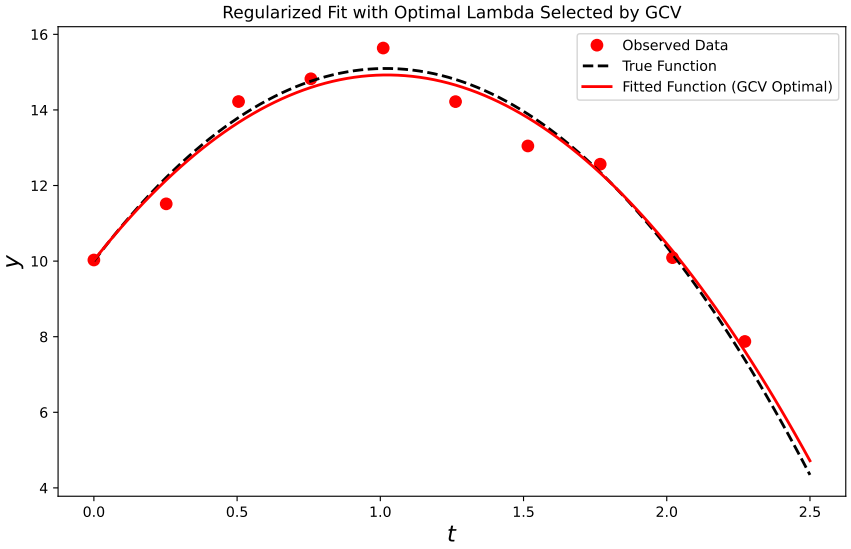
The important point is that, although it may look intimidating, **this is easy to calculate for linear models** for each λ. We can simply evaluate this for a **grid** of λ values and choose the one giving the smallest value.

Example (minus details!)




```
Optimal lambda selected by GCV: 1.2045e+09
```

Example (minus details!) continued



Notes:

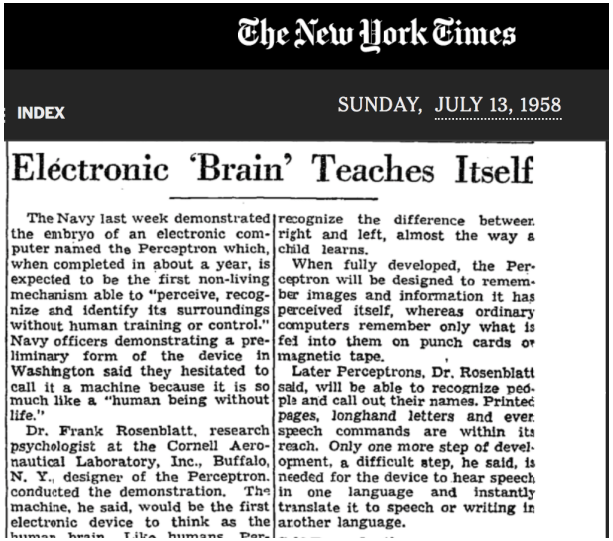
- Because the model is the **true model** here by design, our regularisation parameter enforces the model quite strongly! The GCV function also doesn't vary very much here (look at the scale).
- This relates to the idea that when working with *reliable* physics-informed models, we can typically just take the regularisation parameter to be some relatively large value and **not worry too much**.
- The more **approximate** the model is, however, the more we need to worry about things like using cross-validation to choose our regularisation parameters.

Part V

Neural models

Neural networks

Finally we reach the most hyped model class of today (2024...) – **(artificial) neural networks**! Artificial neural networks have been around since at least 1958, when Rosenblatt presented a computer implementation of the **perceptron**, a simple 'single-layer' neural network, based on earlier work by McCulloch and Pitts (1943). These were inspired by the neurons of the brain, hence the term artificial neural network. Even then the excitement was real:



The New York Times

INDEX

SUNDAY, JULY 13, 1958

Electronic 'Brain' Teaches Itself

The Navy last week demonstrated the embryo of an electronic computer named the Perceptron which, when completed in about a year, is expected to be the first non-living mechanism able to "perceive, recognize and identify its surroundings without human training or control." Navy officers demonstrating a preliminary form of the device in Washington said they hesitated to call it a machine because it is so much like a "human being without life."

Dr. Frank Rosenblatt, research psychologist at the Cornell Aeronautical Laboratory, Inc., Buffalo, N. Y., designer of the Perceptron, conducted the demonstration. The machine, he said, would be the first electronic device to think as the

recognize the difference between right and left, almost the way a child learns.

When fully developed, the Perceptron will be designed to remember images and information it has perceived itself, whereas ordinary computers remember only what is fed into them on punch cards or magnetic tape.

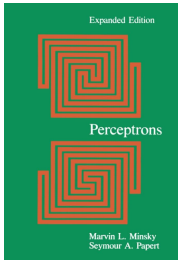
Later Perceptrons, Dr. Rosenblatt said, will be able to recognize people and call out their names. Printed pages, longhand letters and even speech commands are within its reach. Only one more step of development, a difficult step, he said, is needed for the device to hear speech in one language and instantly translate it to speech or writing in another language.

Article from the New York Times on the Rosenblatt's work on the Perceptron.

Artificial intelligence and neural networks: brief history

Artificial intelligence (AI), of which 'machine learning' is a subfield, has undergone numerous 'booms' and 'busts' since around 1950. The term **AI winter** has been used for several of the 'bust' cycles in AI, where funding and interest dropped after failures to live up to the hype. This includes 'bust' periods such as 1974-1980 and 1987-2000. The current (2024) 'boom' period roughly started around 2012. (See e.g. the Wikipedia article AI Winter) for further reading and references).

Within AI, neural networks have undergone their own boom and bust periods. For example, after their initial hype (see NY Times article!), Minsky and Papert published a book *Perceptrons* in 1969 containing several criticims of neural networks. Famously, they showed that *single-layer* neural nets are unable to compute the basic logical operation XOR (exclusive OR). It was known that multiple-layer networks (called *multilayer networks* or *multilayer perceptrons*) were not subject to this limitation, but feasible training methods for these were not known. In contrast, many AI researchers between the 1950s and 1980s focused on **expert systems** and **symbolic AI**, in which *explicit rules* are encoded rather than learned from data (note the similarity to encoding 'engineering knowledge'!)



The book that (briefly) killed neural networks??

Artificial intelligence and neural networks: brief history continued

Development of the **backpropagation** algorithm for training multilayer neural networks helped lead to renewed interest in neural networks in the 1980s. Backpropagation – essentially *efficiently and automatically* using the chain rule to compute gradients and hence carry out optimisation – was popularised in the 1980s, for example in the 1986 paper *Learning representations by back-propagating errors* by Rumelhart, Hinton, and Williams. However, it wasn't until around 2012 that the availability of cheap and powerful computer hardware like **GPUs**, the availability of **large datasets**, and neural network **architecture** improvements, such as **convolutional neural networks (CNNs)** and **recurrent neural networks (RNNs)**, enabled the current explosion of interest (currently the 'hot' architecture is the **transformer**, the 'T' in ChatGPT!). These advances have led to a resurgence in machine learning and the use of neural networks with many- (many, many-) layers, often referred to as **deep neural networks**. Training such networks is known as **deep learning**.

Machine learning is an opposing but complementary approach to rule-based AI – a *yin and yang*! The idea is to **learn** approximate, statistical, rules from the data, rather than impose them *a priori*. As we have seen, this has strengths and weaknesses in engineering applications! For now, engineers appear to need to continue to learn the 'rules' of Newton's Laws of motion alongside machine learning and statistics. Perhaps, however, we will see the machine learning approach continue to supplant traditional approaches? Or perhaps various **hybrid methods**, such as those we looked at in the previous section, will come to dominate? Exciting times!

Neural networks vs linear models

What is a neural network? First consider a simple **linear model** for observations (x_i, y_i), $i = 1, \ldots, m$,

$$y_i \approx b + x_i w$$

where w is a **weight parameter** and b is called a **bias parameter** (or *intercept* in linear regression). This can also be written as

$$y_i \approx 1b + x_i w = \begin{bmatrix} 1 & x_i \end{bmatrix} \begin{bmatrix} b \\ w \end{bmatrix}$$

where we include the 'constant feature' 1. For m observations we get the vector equation

$$y = \bar{X}\bar{w}$$

where

$$\bar{X} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix}$$

and

$$\bar{w} = \begin{bmatrix} b \\ w \end{bmatrix}.$$

We've used a 'bar' over the X and w to indicate the constant/bias terms are included. For various reasons, neural network folk often keep the weights and bias terms separate, and instead write e.g.

$$y = 1_m b + Xw$$

where here

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

and 1_m is a *ones vector* of length m.

Neural networks vs linear models

So far we have a standard linear model. This can obviously be extended to multiple input features, e.g. if we denote the ith measurement of the jth feature as x_{ij} we have a design/feature matrix of the form:

$$X = [x_{ij}]_{i=1:m, j=1:n}$$

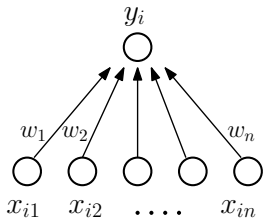
and linear model

$$y \approx b1_m + Xw$$

for feature weights

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

The neural network folk like to represent this model with diagrams like the below:



The **bias** term is sometimes depicted explicity but **often left implicit**. We almost have a simple neural network! We just need one more element that is normally implied by diagrams like the above...

Activation functions

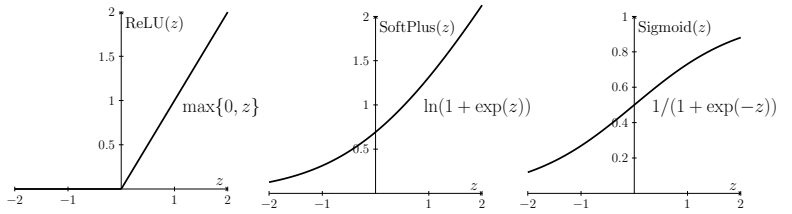
The main element distinguishing a neural network from a standard linear model is a **nonlinearity** introduced by an **activation function** applied to the output of a linear component before the final output is produced. This means our model for a single output has the general form:

$$y \approx f(Xw + 1_m b)$$

where f is a **nonlinear function** called an **activation function** that is (usually) applied **elementwise**, i.e. to each sample realisation, so that

$$y_i \approx f(\text{row}_i(X)w + b)$$

where $\text{row}_i(X)$ is the ith row of X. Activation functions were originally inspired by the behaviour of biological neurons, which typically only 'fire' (send a signal) when a threshold signal is reached. Thus, typical activation functions generally have a **threshold** or **switch-like** shape, with varying degress of smoothness and saturation properties, e.g.



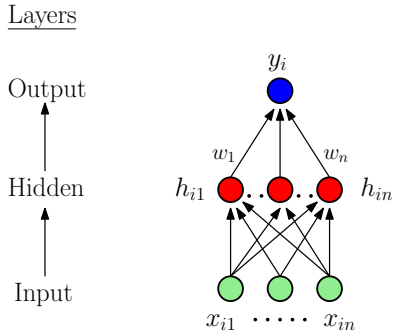
Typical activation functions, where z is the output of the linear component.

Although originally inspired by biology, **mathematically**, the important thing is that **introducing nonlinearities** allows us to represent more complex classes of functions: so-called **universal approximation theorems** state, roughly, that sufficiently complex neural networks, with nonlinear activation functions, can approximate *any* function.

Multiple neurons and multiple layers

So far we have a single 'regression' style equation that represents a single output given multiple features, a weight for each feature, a single bias term, and a nonlinearity applied to the output of the linear component. This is called an **artificial neuron**, or just **neuron** for short. Similar models occur in statistics, for example this class of models essentially corresponds to so-called **generalised linear models** in statistics. What else distinguishes neural networks from standard statistical models?

Another key element in neural networks is that we typically consider **multiple neurons**, arranged in **layers**, as in the below diagram. There are three *types* of layers: **input**, **output**, and the intermediate **hidden** layers. Note, however, that we can have **as many hidden layers as we want**!



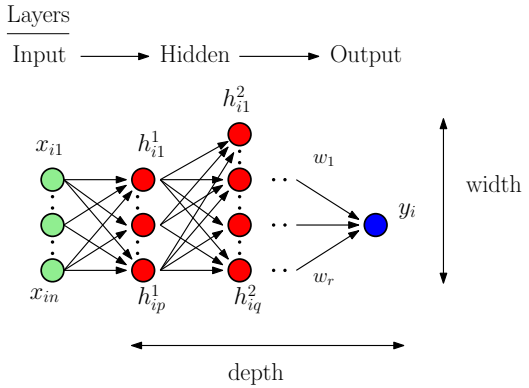
Neural network with *one* hidden layer

Thus we have a collection of regression-like equations that in turn form the features for another collection of regression-like equations, and so on, until we reach the output. Note, though: **without nonlinear activation functions this would still just be a big linear model!**

Architecture

The term **architecture** refers to the structural design of the neural network: how many **layers**, how many **neurons** – also called **nodes** or **units** – in each layer, etc. The number of layers is referred to as the **depth**, while the number of neurons in a layer is referred to as the (layer) **width**.

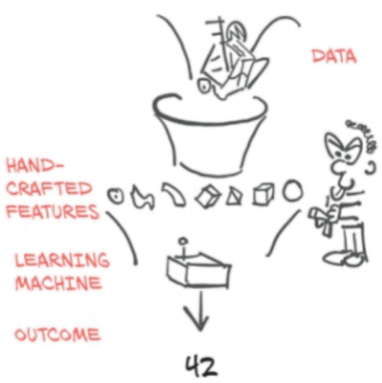
A **feedfoward neural network** is the archetypal neural network where information moves in only one direction – from the **input** nodes, through the **hidden** layers, and finally to the **output** nodes (as on the previous page). There are **no loops or feedback connections** between nodes in the network. Hence its general architecture consists of an input layer, one or more hidden layers, and an output layer:



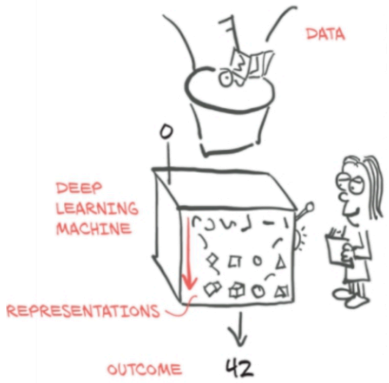
Recurrent neural networks include 'recurrent connections' or **loops** and are hence no longer feedforward neural networks. For simplicity we only focus on very simple **feedforward** neural networks here. It is relatively simple to experiment with different architectures in PyTorch on your own, however!

How many layers? Deep learning and feature engineering

Since around 2012 there has been a large shift towards increasing **deep** architectures with **many layers**. Why? What's the goal? The following image from *'Deep Learning with PyTorch'* by Stevens, Antiga, and Viehmann (2020) illustrates the conceptual shift associated with deep learning vs. more 'traditional' machine learning/data science:



Feature engineering



Deep learning

That is, in *theory*, the goal of deep learning is to **automatically learn** as much of the data-to-outcome pipeline as possible, reducing the reliance on hand-crafted components. Note, however, that this diagram just illustrates the **general conceptual orientation** of different approaches, and is not a prescription of what to do. **As engineers**, we have professional and ethical obligations in terms of safety and reliability, as well as a pragmatic orientation on timely results. **Current deep learning pipelines may not be the safest/most reliable/timely approach for many engineering applications!**

Building neural networks in PyTorch

So far we've mainly just used the tensor data structures available in PyTorch, and a little of its autodiff capabilities. *In theory, we could build neural networks from these components.* However, the main `torch` module of PyTorch also contains a submodule `torch.nn` dedicated to helping us build neural networks more easily.

There are two main ways to build neural networks in PyTorch. The *simplest* is to use the further submodule of `nn`, `torch.nn.Sequential`, which allows us to simply specify a *sequence* of 'layers' to define a neural network model. While we can use this to do familiar things like predict the motion of a pendulum or projectile, we can also use this approach to make a simple neural network for **image classification**. For example, here is **a simple neural network designed to classify 28×28 pixel grayscale images into one of five classes**:

```
import torch
import torch.nn as nn

# Define the model
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 128),
    nn.ReLU(),
    nn.Linear(128, 5),
    nn.Softmax(dim=1)
)
```

That's it! Sort of... we still need to obtain our training data and train our model. But first, let's consider what's happening in our model.

Model summary

We can use `torchsummary` (available in Colab) to summarise the model:

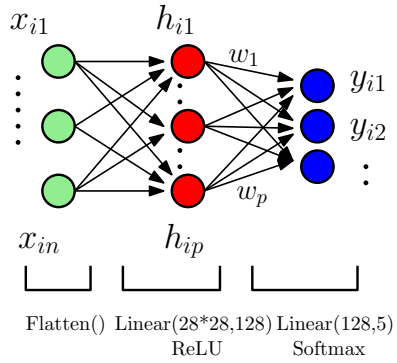
```
from torchsummary import summary
summary(model, input_size=(1, 28, 28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
          Flatten-1                  [-1, 784]               0
           Linear-2                  [-1, 128]         100,480
             ReLU-3                  [-1, 128]               0
           Linear-4                    [-1, 5]             645
          Softmax-5                    [-1, 5]               0
================================================================
Total params: 101,125
Trainable params: 101,125
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.39
Estimated Total Size (MB): 0.40
----------------------------------------------------------------
```


Terminology

Note on **terminology**: `Flatten`, `ReLU` and `Softmax` are considered 'layers' in the *PyTorch* model, but these are **not parameterised**. They instead represent **data reshaping**, **data transformations**, and **activation functions**. In **standard, mathematical neural network terminology** we usually only use the term 'layer' for what we would call **parameterised layers** in PyTorch (other than the input/data layer which is not parameterised).

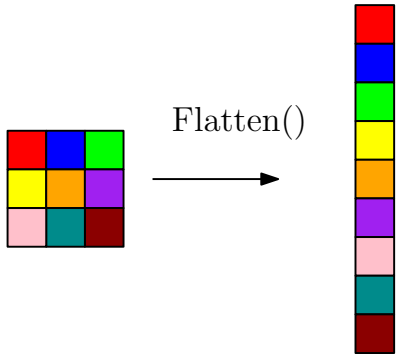
Furthermore, in a PyTorch 'layer' like `nn.Linear(28 * 28, 128)`, its mathematical/diagrammatic counterpart is associated with the *output* nodes (128) of the PyTorch layer. So the **width** of the layer in the mathematical/diagrammatic sense is 128. Here we have five PyTorch 'layers' corresponding, in standard neural network terms, to **one input layer** and **two (parameterised) layers**. That is, we have one (non-parameterised) input layer, one (parameterised) hidden layer, and one (parameterised) output layer. There are also multiple output classes.



Neural network predicting multiple class probabilities.

Input layer

The first 'layer' (in PyTorch terms) is `nn.Flatten`. This turns an **image into a column vector** by 'flattening' the image *row-wise* into a long vector:



Turning an image into a vector.

Note that a common mathematical convention (and the Fortran convention) is to flatten *column-wise*, but C and Python – and hence PyTorch – **flatten row-wise**.

This step is required for inputs into `nn.Linear` layers in PyTorch, which expect input tensors to be two-dimensional of shape `[batch_size, features]`. Here the 'features' are the pixel values for each of the $28 \times 28 = 784$ pixes. Note though, that other, more advanced image-handling layers (such as *convolutional layers*) do not require images to be flattened, and can work directly with multi-dimensional tensors.

This is a non-parameterised layer and is essentially our input layer.

Hidden layers

The `nn.Linear` PyTorch layer is the first parameterised layer of the neural network and is a **hidden** layer because its *output* nodes are not directly observed outputs.

This layer has $28 \times 28 = 784$ input nodes and 128 output nodes. That is, we are representing 128 'regressions', each taking 784 input variables with their own weights! Each regression also has a bias term, so we get $128 \times 784 + 128 = 100480$ learnable parameters. The width of this layer is 128, i.e. the number of *output* nodes in the layer.

Mathematically, the subsequent `nn.ReLU` **activation function** is part of this layer. Separating the specification of the activation function from the linear component *in code*, however, gives greater flexibility.

Output layer

The `nn.Linear(128, 5)` layer is the **output layer**. It maps the 128 hidden neurons to five output classes. Mathematically, the `nn.Softmax(dim=1)` **activation function** is also part of this layer.

What does `nn.Softmax` do? It normalises, in a particular way, an input **vector** z (represented as a tensor of course) into a discrete probability distribution **vector** p of the same dimension:

$$\text{Softmax} : z \mapsto p$$

where z and p are vectors of the same length, and p sums to one. Hence here we get a probability vector with five entries. Note that our z vector (tensor) here has a *column* for each output class, and hence we normalise over the column dimension, dim=1 (see e.g. torchsummary).

Interestingly, in contrast to most activation functions, this function is **not** applied separately to each pre-activation neuron value. Instead, the softmax function depends on the values of **all** elements of the vector given, as its job is to ensure the values **together** sum to one. It is defined by:

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

Softmax is usually only used for **output layers**, to provide a **final probability distribution over a collection of discrete classes**. It is rarely used 'inside' neural networks, i.e. in hidden layers, where elementwise activation functions are common.

Finally, although Softmax is *conceptually* important, it turns out that, for **multi-class classification**, it is generally better practice to use the LogSoftmax function, `nn.LogSoftmax(dim=1)`, instead of Softmax. And rather than a sum of squared errors, for multi-class classification this last output layer is usually used with a so-called **negative log likelihood loss**, `nn.NLLLoss()` that takes in log-probabilities. We won't go into further detail on the how or why of this in this course, but it's useful to be aware of if you want to do image classification in the future!

Loading data in PyTorch

Let's look at an example. The machine learning and computer vision communities have collected many benchmark datasets over the years, and have run numerous competitions based on such datasets. (Interestingly, Donoho has argued that such practices have been central to the successes of data science and machine learning).

A classic example dataset is the handwritten digit-recognition dataset called **MNIST**. This is a dataset of 28×28 images of handwritten digits with 60,000 training images and 10,000 testing images. LeCun et al. describe this as

> a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

The PyTorch module `torchvision` contains useful datasets, models, and image transformations for **computer vision** (image/video machine learning) applications. There are 'built-in' datasets in the module `torchvision.datasets`, which also contains tools (functions, classes etc.) for working with custom datasets. We can load the MNIST dataset as follows:

```
import torch
from torchvision import datasets, transforms
# Define transformations to be applied to loaded images
# Here just a conversion to tensor format.
transform = transforms.ToTensor()
# Load training and test datasets
train_dataset = datasets.MNIST(root='./data', train=True,
                transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False,
                transform=transform, download=True)
```

The above looks for a local 'data' folder. If it doesn't exist it creates this and downloads the dataset from public sources.

Inspecting the data

The returned objects have (the verbose!) type `torch.utils.data.Dataset`.

```
print(train_dataset)
```

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: ./data
    Split: Train
    StandardTransform
Transform: ToTensor()
```

We can access (image, label) pairs of the dataset using standard indexing:

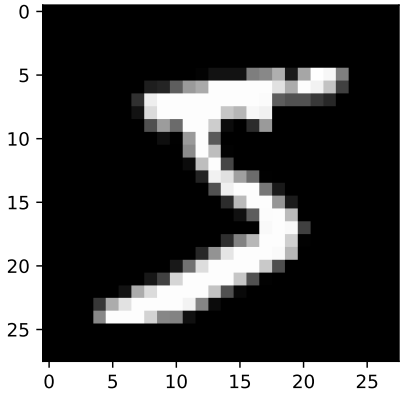
```
img, label = train_dataset[0]
print('image shape:')
print(img.shape)
print('\n')
print('image label:')
print(label)
```

```
image shape:
torch.Size([1, 28, 28])


image label:
5
```

Let's look at the image!

Inspecting the data

```
import matplotlib.pyplot as plt
# need to permute tensor to matplotlib layout
plt.imshow(torch.permute(img,[1,2,0]),cmap='gray')
plt.show()
```

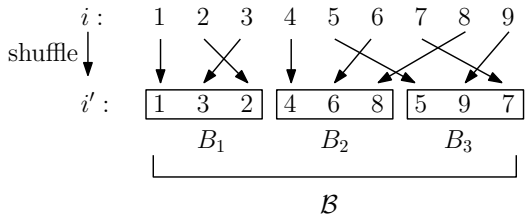


Note: typically we further **normalise** the data e.g. to have zero mean and unit standard deviation. This can be done using `transforms.Normalize` but requires computing the mean and standard deviation 'offline', i.e. outside the transform. We will ignore this step for now for simplicity.

Mini-batching the data

For various reasons, modern neural network models usually process data in small batches called **minibatches** rather than all at once. This is memory efficient but also has been shown to **improve generalisability**, i.e. avoid overfitting.

Terminology can be inconsistent, but in general, we call the full dataset the batch, and subsets of the dataset minibatches. Given an original indexing i of data points, we construct a randomised (e.g. by shuffling) **partition** $\mathcal{B}$ of **minibatches** $B_{i'}$, $i' = 1, \ldots, mb$ for mb representing the number of mini-batches. For example:



The module `torch.utils.data` has a tool `DataLoader` to help work with minibatches. This takes a dataset, a `batch_size` and a `shuffle` option.

```
train_loader = torch.utils.data.DataLoader(train_dataset,
                                batch_size=64,shuffle=True)
```

The returned object `train_loader` is a Python iterable – an object that can return its members (here, minibatches) one at a time, and hence be 'iterated over', e.g.:

```
for mb_imgs, mb_labels in train_loader:
```

If `shuffle` is true, the dataset is shuffled once at the beginning of each new iteration over the DataLoader.

Batch, minibatch and stochastic gradient descent

The above leads to three variations of **gradient descent**:

- Batch (standard) gradient descent
- Minibatch gradient descent
- Stochastic gradient descent

The first uses the full dataset at each iteration of gradient descent, the second uses a minibatch for each iteration of gradient descent, and the last uses a single data point. Note: PyTorch uses `optim.SGD` for all versions gradient descent; the difference comes from what data you pass it at each iteration.

The basic training process is to randomly divide the training set into minibatches (possibly being the full batch or single data points). Then we fix the shuffled order and iterate through the batches until all data points have been seen. This is called an **epoch**. We then re-shuffle the data and divide into new minibatches and continue for some number of epochs. The full dataset will hence be 'seen' the same number of times as there are epochs. The total number of gradient descent steps will be the number of epochs times the number of minibatches the dataset is divided into at each epoch.

This is illustrated below ...

Batch, minibatch and stochastic gradient descent

Here is simple *pseudocode* (won't run!) for the above:

```
initialise model parameters
create train_loader with batch_size

for epoch in range(num_epochs):
    epoch_loss = 0.0
    for mb_inputs, mb_labels in train_loader:
        # zero out gradients.
        optimizer.zero_grad()

        # forward pass: compute predictions
        mb_outputs = model(mb_inputs)

        # compute loss between predictions and true labels
        mb_loss = loss_function(mb_outputs, mb_labels)

        # Accumulate epoch loss (use .item() to get scalar)
        epoch_loss += mb_loss.item()

        # backward pass: compute gradients
        mb_loss.backward()

        # update model parameters
        optimizer.step()

# For
# - Batch GD, batch_size = size of training dataset
# - Minibatch GD, batch_size = 32, 64, etc.
# - Stochastic GD, batch_size = 1
```


Training the neural network in PyTorch

Let's put it all together!

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define the model with explicit LogSoftmax
num_classes = 10  # For MNIST dataset
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 128),
    nn.ReLU(),
    nn.Linear(128, num_classes),
    nn.LogSoftmax(dim=1)
)
# Define transformations (simple conversion to tensor)
transform = transforms.ToTensor()

# Load datasets
train_dataset = datasets.MNIST(root='data', train=True,
                               transform=transform, download=True)
test_dataset = datasets.MNIST(root='data', train=False,
                              transform=transform)

# Data loaders
batch_size = 32  # Adjust as needed
train_loader = DataLoader(dataset=train_dataset,
                                  batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset,
                                  batch_size=batch_size, shuffle=False)
```

```python
# Loss function and optimiser
criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
# Training loop
num_epochs = 10  # Adjust as needed
for epoch in range(num_epochs):
    epoch_loss = 0.0
    for mb_data, mb_targets in train_loader:
        # Zero the gradients
        optimizer.zero_grad()
        # Forward pass
        outputs = model(mb_data)
        # Compute minibatch loss
        mb_loss = criterion(outputs, mb_targets)
        # Accumulate epoch loss (use .item() to get scalar)
        epoch_loss += mb_loss.item()
        # Backward pass and optimization
        mb_loss.backward()
        optimizer.step()
    # Compute average loss per minibatch
    avg_epoch_loss = epoch_loss / len(train_loader)
    # display
    print(f'Epoch [{epoch+1}/{num_epochs}], Average (Per Minibatch) Los
```

```
Epoch [1/10], Average (Per Minibatch) Loss: 0.8695
Epoch [2/10], Average (Per Minibatch) Loss: 0.3779
Epoch [3/10], Average (Per Minibatch) Loss: 0.3237
Epoch [4/10], Average (Per Minibatch) Loss: 0.2936
Epoch [5/10], Average (Per Minibatch) Loss: 0.2709
Epoch [6/10], Average (Per Minibatch) Loss: 0.2511
Epoch [7/10], Average (Per Minibatch) Loss: 0.2340
Epoch [8/10], Average (Per Minibatch) Loss: 0.2185
Epoch [9/10], Average (Per Minibatch) Loss: 0.2047
Epoch [10/10], Average (Per Minibatch) Loss: 0.1926
```



```python
# Evaluation on test set
correct = 0
total = 0

with torch.no_grad():  # Disable gradient calculation
    for data, targets in test_loader:
        outputs = model(data)
        _, predicted = torch.max(outputs.detach(), 1)
        total += targets.size(0)
        correct += (predicted == targets).sum().item()

accuracy = 100 * correct / total
print(f'Accuracy on test set: {accuracy:.2f}%')
```

```
Accuracy on test set: 94.58%
```

What next?

Note: there are many, many more difficulties and subtleties that we won't go into here (see future courses??)

In general we need to preprocess the data (despite the 'Deep Learning vs Feature Engineering' picture!), define appropriate architectures, use the right training methods (including various 'tricks of the trade'), handle training, testing and validation splits carefully, etc etc.

Furthermore, more complex models are usually constructed by *sublcassing* the `nn.Module` class rather than using `nn.Sequential`. We have only just scratched the surface of neural networks and PyTorch. However, hopefully now you can start exploring yourself!

Another useful way to explore is to download **pre-trained models**. For example, PyTorch Hub hosts a variety of models that can be easily accessed:

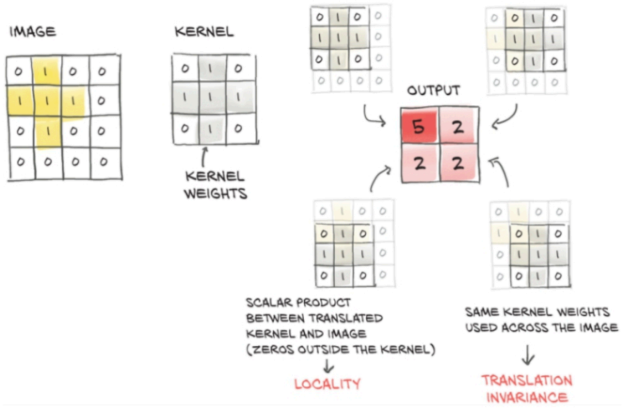
```
import torch
model = torch.hub.load('pytorch/vision', 'resnet18',
                       pretrained=True)
```

Also `torchvision.models` contains many computer vision models e.g.

```
from torchvision.models import resnet50
```


Example architecture improvements: convolutional neural networks

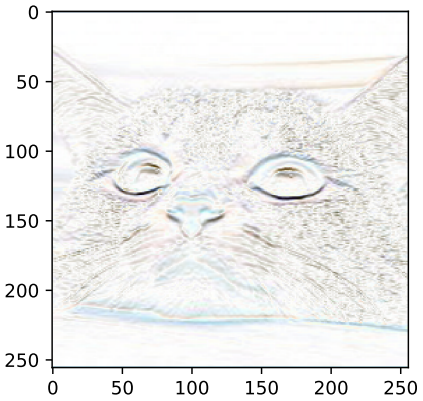
We have previously mentioned **convolutional neural networks** (CNNs) as a biology/physics-inspired neural network architecture. *Briefly*, a **convolution kernel** is a small 'patch shaped' tensor of weights that can be applied to local **patches** of an image (or video or time-series, for appropriate definitions of 'patch') by multiplying the weights of the kernel with the values of each patch, illustrated in the below figure:



This is a special type of **linear** operation, to be passed into subsequent nonlinear activation functions. For example, the below snippet manually applies an edge-detection kernel of dimensions 3 by 3 to the patches of a cat image using `torch.einsum` (which can apply linear operations like convolution to tensors without flattening).

Manual convolution

```
# vertical filter kernel
kernel = torch.tensor([[ -1,-2,  -1],
                       [ 0,  0,   0],
                       [ 1,  2,   1]],dtype=torch.float32)

padded_cat = F.pad(cat, pad=(1, 1, 1, 1),
                   mode='constant', value=0)
patches = padded_cat.unfold(1, 3, 1).unfold(2, 3, 1)

filtered_cat = torch.abs(torch.einsum('cijkl,kl->cij',
                                      patches, kernel))
```



The above result is called the **convolution** of the image and the kernel. Not surprisingly, PyTorch implements this itself, e.g. in `nn.Conv2d`. We will not explore these in any more detail in this course but you should look into them!

Physics-informed neural networks: hybrid neural models

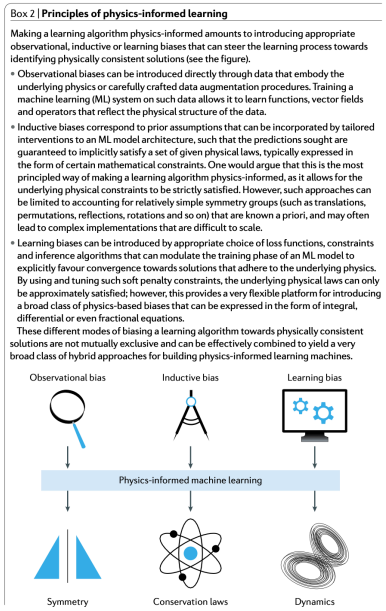
Finally, we will end on 'physics-informed neural networks'. Coming soon!

Part VI

Hybrid neural models

Hybrid neural models

For our last (brief) section we look at combining a neural network data model with a scientific penalty term to produce a **hybrid neural model**. Others call this (depending on the scientific knowledge included) **physics-informed neural networks** (PINNs). This is also what Karniadakis et. al (2021) call *learning bias*:



Box 2 | **Principles of physics-informed learning**

Making a learning algorithm physics-informed amounts to introducing appropriate observational, inductive or learning biases that can steer the learning process towards identifying physically consistent solutions (see the figure).

- Observational biases can be introduced directly through data that embody the underlying physics or carefully crafted data augmentation procedures. Training a machine learning (ML) system on such data allows it to learn functions, vector fields and operators that reflect the physical structure of the data.
- Inductive biases correspond to prior assumptions that can be incorporated by tailored interventions to an ML model architecture, such that the predictions sought are guaranteed to implicitly satisfy a set of given physical laws, typically expressed in the form of certain mathematical constraints. One would argue that this is the most principled way of making a learning algorithm physics-informed, as it allows for the underlying physical constraints to be strictly satisfied. However, such approaches can be limited to accounting for relatively simple symmetry groups (such as translations, permutations, reflections, rotations and so on) that are known a priori, and may often lead to complex implementations that are difficult to scale.
- Learning biases can be introduced by appropriate choice of loss functions, constraints and inference algorithms that can modulate the training phase of an ML model to explicitly favour convergence towards solutions that adhere to the underlying physics. By using and tuning such soft penalty constraints, the underlying physical laws can only be approximately satisfied; however, this provides a very flexible platform for introducing a broad class of physics-based biases that can be expressed in the form of integral, differential or even fractional equations.

These different modes of biasing a learning algorithm towards physically consistent solutions are not mutually exclusive and can be effectively combined to yield a very broad class of hybrid approaches for building physics-informed learning machines.

Whatever we call it, we've now learned enough to do it ourselves!

Pendulum revisited

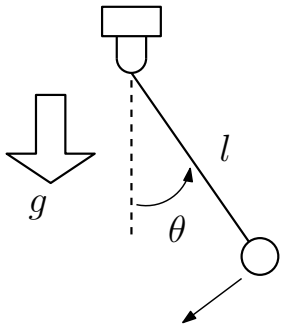
Recall that the angular motion $\theta(t)$ of a pendulum with length l subject to Newton's laws and no friction or air resistance, is given by

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = 0$$

with initial conditions

$$\theta(0) = \theta_0, \quad \frac{d\theta}{dt}(0) = v_0$$

A schematic is shown below.



The **small angle approximation** use $\sin\theta \approx \theta$, giving the **linear pendulum** model:

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\theta = 0$$

with the same initial conditions.

Pendulum revisited

Now, let's consider a variation on our original pendulum prediction problem. Suppose we have M, (possibly noisy) observations of the form

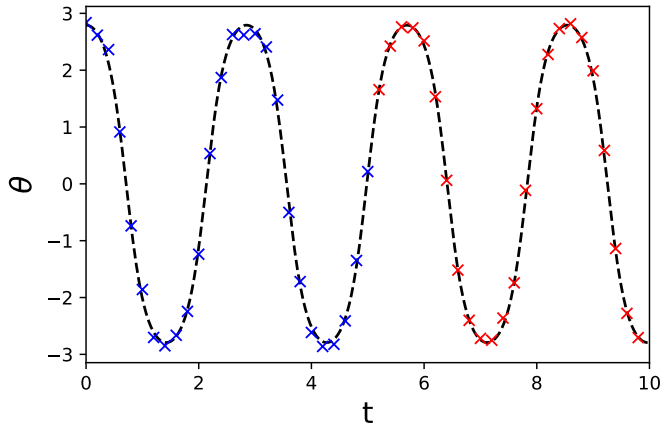
$$(t_i, \theta_i),\ i = 1, \ldots, M$$

from a nonlinear pendulum and wish to predict θ_j for a collection of new, future t_j values

$$(t_j, \theta_j),\ j = M+1, \ldots, M+P$$

.

For example, below we show the observed data in blue, the future (not yet seen) data in red, and the true, unobserved angle trajectory in black.



Two time grids, two losses

As in our discussion of hybrid linear models, we again have **two** time grids: a **fine time grid** associated with the underlying *physical process* (as governed by the ODE/Newton's Laws), and a **coarse time grid** associated with our *observations* of the physical process. (Reality may of course involve continuous time physics but here we use a very fine grid as a proxy for this.)

These two time grids naturally lead to two types of loss functions: a **data loss** associated with the observations, and a **physics loss** associated with the underlying physical process. First, we will consider the data loss.

If we denote the 'underlying' angular motion defined on the fine time grid, t_{fine}, by θ_{fine}, and our observations of the angular motion on the coarse time grid, t_{obs}, by θ_{obs}, we can (conceptually at least) use an observation matrix A_{obs} to relate the variables on the two grids:

$$t_{\text{obs}} = A_{\text{obs}} t_{\text{fine}}, \quad \theta_{\text{obs}} = A_{\text{obs}} \theta_{\text{fine}}.$$

Rather than directly enforce the physics right now, we will first assume that the fine-scale dynamics can be **approximated by a neural network model**:

$$\theta_{\text{fine}} = f_{\text{NN}}(t_{\text{fine}}; w),$$

where w are the parameters (weights and biases) of the neural network, and $f_{\text{NN}}(t_{\text{fine}}; w)$ represents the vector of predictions at the fine time grid t_{fine}. We explicitly represent the fine-scale dynamics here because we ultimately want to predict the angular motion at times we have not observed.

These elements lead to a **data loss** term that, for fixed data, depends on the neural network parameters and takes a form such as:

$$\begin{aligned} \text{loss}_{\text{data}}(w) &= \frac{1}{M} \|\theta_{\text{obs}} - A_{\text{obs}} f_{\text{NN}}(t_{\text{fine}}; w)\|^2 \\ &= \frac{1}{M} \|\theta_{\text{obs}} - f_{\text{NN}}(t_{\text{obs}}; w)\|^2 \end{aligned}$$

where M is the number of observations, which we may or may not divide by (the minimum loss is the same, but it can help with scaling).

Physics loss and time derivatives

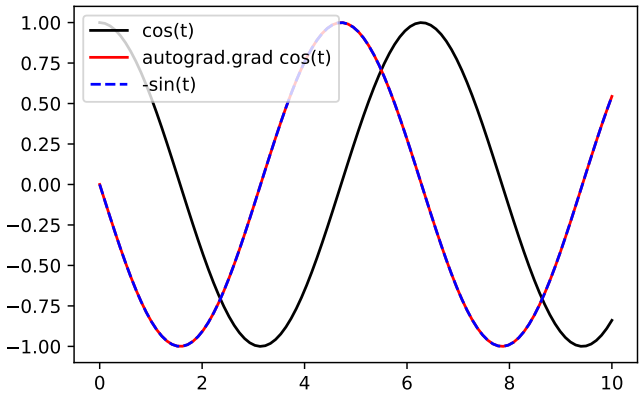
We will consider the neural network model in more detail shortly. For now, we consider, at a high level, how to capture the **physics loss** and, in order to do that, how to represent **derivative terms** and evaluate them on the fine time grid. The key element is PyTorch's autodiff capabilities, in particular PyTorch's `autograd` autodiff engine and the `torch.autograd.grad` function.

Here, rather than differentiating with respect to the neural network parameters, we will differentiate the neural network model with respect to **time**, in order to represent the derivative appearing in the ODE. This is essentially the same as the **finite difference** approach we used in the linear model to represent time derivatives, e.g. DBw, but now we will use `torch.autograd.grad` to compute the derivative of the neural network model.

First, here is a simple example of using `torch.autograd.grad` to differentiate `cos(t)` and evaluate it on a fine time grid:

```
import torch
import torch.autograd as autograd
import matplotlib.pyplot as plt
t_fine = torch.linspace(0,10,100,requires_grad=True)
y = torch.cos(t_fine)
dydt = autograd.grad(y, t_fine, torch.ones_like(y),
                                 create_graph=True)[0]
```

We can plot this and see it matches the true derivative, `-\sin(t)`...



Notes:

The function $\cos(t)$ is a scalar function, but if we evaluate it with a vector (tensor) input, the result being a vector (tensor), we can obtain a **vector of scalar derivatives at the same time points** using the general form:

```
torch.autograd.grad(
    outputs,
    inputs,
    grad_outputs=torch.ones_like(outputs),
    create_graph=True
)
```

Briefly, when `outputs` is the result of applying a scalar function to each member of `inputs`, and `grad_outputs` is as above, PyTorch **interprets the gradient computation element-wise**, giving a vector (tensor) with ith entry $\partial \text{outputs}_i / \partial \text{inputs}_i$.

Finally, the `create_graph=True` argument is necessary if we want to differentiate the result of `torch.autograd.grad` again. This is not necessary for this simple example, but will be necessary when we consider higher order derivatives in the ODE.

Constructing the physics loss

Given the ability to compute time derivatives in PyTorch, we can do this for the neural network model $f_{\text{NN}}(t_{\text{fine}}; w)$, and then use these derivatives to construct the physics loss. This will have the general form:

```
import torch
import torch.autograd as autograd
import matplotlib.pyplot as plt
t_fine = torch.linspace(0,10,100,requires_grad=True) # e.g.
theta_fine_pred = model(t_fine)

# Compute first derivative dtheta/dt
# note: grad always returns a tuple of gradients,
# even if only one, so take first element
dtheta_dt_fine = torch.autograd.grad(
    outputs=theta_fine_pred,
    inputs=t_fine,
    grad_outputs=torch.ones_like(output),
    create_graph=True
)[0]

# Compute second derivative d2theta/dt2
d2theta_dt2 = torch.autograd.grad(
    outputs=dtheta_dt_fine,
    inputs=t_fine,
    grad_outputs=torch.ones_like(dtheta_dt_fine),
    create_graph=True
)[0]
```

For the neural network approximation to the pendulum motion, and ODE assumed valid for all t_{fine}, we have a physics loss function of the form

$$\text{loss}_{\text{physics}}(w) = \frac{1}{P} \| \frac{\partial^2 f_{\text{NN}}(t_{\text{fine}}; w)}{\partial t^2} + \frac{g}{l} \sin f_{\text{NN}}(t_{\text{fine}}; w) \|^2$$

Combining the losses

The overall loss function is then a combination of the data loss and the physics loss:

$$\text{loss}_{\text{total}}(w) = \text{loss}_{\text{data}}(w) + \lambda \times \text{loss}_{\text{physics}}(w)$$

where λ is a hyperparameter that weights the relative importance of the data and physics terms.

As for the linear model, λ can be chosen by using a simple hold-out validation set, a more sophisticated method such as cross-validation, or set by the user based on external knowledge. Here we just manually set (by trial and error…) λ for simplicity.

We're almost ready to put this all together, but first let's consider the neural network model in a bit more detail.

Simple neural network model

Here is a simple example of a neural network model in PyTorch, with two hidden layers and tanh activation functions. We use tanh here because it is smooth (infinitely differentiable) which tends to be appropriate for ODE models (requiring derivatives in time), c.f. ReLU which is technically not smooth and more common in classification tasks.

We will again use `torch.nn.Sequential` to define the model:

```
import torch.nn as nn
nn.Sequential(
        nn.Linear(1, 50),
        nn.Tanh(),
        nn.Linear(50, 50),
        nn.Tanh(),
        nn.Linear(50, 1)
        )
```

This has a single input neuron (representing time), two hidden layers with 50 neurons each, and a single output neuron (representing the angular motion). The `nn.Linear` layers are fully connected layers, and the `nn.Tanh` layers are the activation functions.

The input is a single scalar 'feature' t and the output is a single scalar 'response' $\theta(t)$. As with our other models, we can evaluate this model for a collection of input value, e.g. at t_{obs} or t_{fine}, to get a collection of output values. Each evaluation is like a 'row' of a linear model, though now we have a nonlinear model, and evaluating the model for a collection of input values corresponds to iterating over the batch dimension.

Putting it all together

We can now put all these elements together to construct a hybrid neural model for the pendulum motion. We assume that the data has been generated already as above.

In contrast to the linear example, we will use the **nonlinear pendulum model** for the physics loss, as well as generate the data using the **nonlinear pendulum model**. That is, we have a 'perfect' physics model. We will look at can also look at using the linear physics model, i.e. an imperfect physics model, as for the linear model.

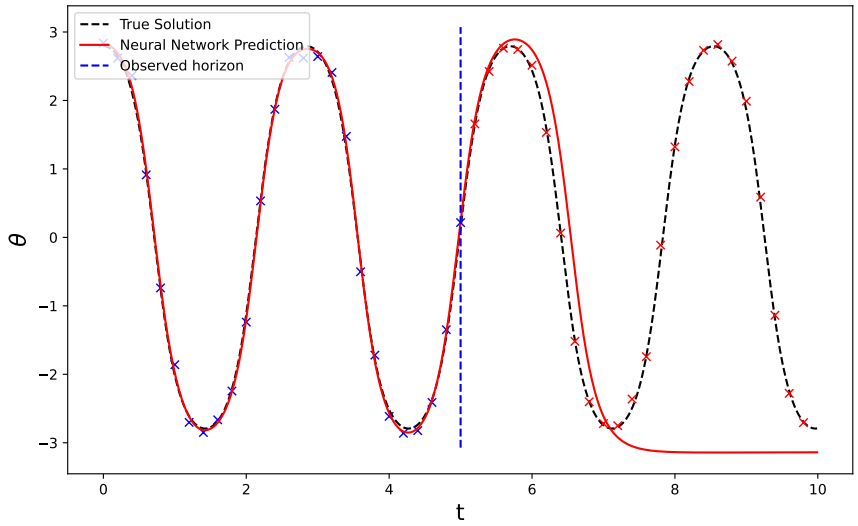
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# NN parameters
torch.random.manual_seed(0)
lambda_reg = 1.5e-3  # Regularization parameter
num_epochs = 10000  # Number of training epochs

# Define the neural network model using nn.Sequential
model = nn.Sequential(
        nn.Linear(1, 50),
        nn.Tanh(),
        nn.Linear(50, 50),
        nn.Tanh(),
        nn.Linear(50, 1)
        )

# Set up the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```



```
# Training loop
for epoch in range(num_epochs):
    optimizer.zero_grad()
    # Convert t_data and t_grid to torch.float32
    t_data_tensor = torch.tensor(t_data,
            dtype=torch.float32).unsqueeze(1)
            # unsqueeze to add a dimension
    t_grid_tensor = torch.tensor(t_grid,
            dtype=torch.float32,
            requires_grad=True).unsqueeze(1)
    y_data_tensor = torch.tensor(y_data, dtype=torch.float32)

    # Compute the model predictions at observation points
    y_pred_obs = model(t_data_tensor)
    # Compute the data loss
    data_loss = torch.mean((
        y_pred_obs.squeeze() - y_data_tensor) ** 2)

    # Compute the model predictions at fine grid
    y_pred_grid = model(t_grid_tensor)
    # Compute first derivative dy/dt on fine grid
    dy_dt = torch.autograd.grad(
        outputs=y_pred_grid,
        inputs=t_grid_tensor,
        grad_outputs=torch.ones_like(y_pred_grid),
        create_graph=True
    )[0]
    # Compute second derivative d2y/dt2
    d2y_dt2 = torch.autograd.grad(
        outputs=dy_dt,
        inputs=t_grid_tensor,
        grad_outputs=torch.ones_like(dy_dt),
        create_graph=True
    )[0]
```

```
    # Define the ODE residual
    ode_residual = d2y_dt2 + g_true/l_true*torch.sin(y_pred_grid)

    # Compute the physics loss
    physics_loss = torch.mean(ode_residual ** 2)

    # Total loss
    loss = data_loss + lambda_reg * physics_loss

    # Backward pass and optimization
    loss.backward()
    optimizer.step()
```

Result:



Takeaways

Notes:

- **I found it quite difficult** to tune the training parameters for the neural network model! The model was very sensitive to the learning rate, the number of epochs, and the regularisation parameter. I also gave up on SGD and used the Adam optimiser instead! Even then **my results were not really that great.**
- While this may be **my own limitations**, it is also a common issue with neural networks: they can be difficult to tune and interpret. This is another reason why we included simpler **linear models** in this course – they are often **much easier to interpret and tune**. In addition, the same ideas can be extended to nonlinear models using a similar approach to the linear model approach, without the need for neural networks.
- Having said that, with more time and effort, neural networks **can** be very impressive and powerful tools!

YOUR CHALLENGE: Try to improve the results above!

Further warnings...

Cautionary notes

Although various successes have been reported for ML-based PDE solvers in comparison with standard numerical methods, there are, in our view, several ways in which this success is probably limited. First, to be useful as a surrogate model, an ML-based solver must reduce the total computational cost in downstream applications. This includes the cost of generating data and training models, both of which are unaccounted for when comparing speed with standard solvers[38]. Speed is thus necessary—but not sufficient—to be useful for forward problems. Second, ML-based solvers have important qualitative limitations that standard solvers do not share. In particular, there are serious concerns about accuracy when generalizing to new parameter spaces[39], numerical stability[40] and predicting chaotic systems[41]. Increased speed seems to be the one area in which ML-based solvers might have an advantage over standard solvers. Third, one of the most widely researched methods involving ML and PDEs—the so-called physics-informed neural network (PINN)[42]—is known to be orders of magnitude slower than standard numerical methods at forward problems (that is, solving PDEs)[43,44]. Furthermore, PINNs often fail to converge to a reasonable approximation[45,46], even for simple toy problems[47–49]. Physics-informed neural networks have had some success[31] for ill-posed and inverse problems; however, they do not seem to outperform alternatives such as discrete grid-based methods[50].

The results in this Analysis call into question whether ML has actually been as successful at solving PDEs from fluid mechanics and related fields as the scientific literature would suggest. We identify two

From:

nature machine intelligence

Analysis https://doi.org/10.1038/s42256-024-00897-5

Weak baselines and reporting biases lead to overoptimism in machine learning for fluid-related partial differential equations

Received: 23 August 2023

Accepted: 13 August 2024

Published online: 25 September 2024

Check for updates

Nick McGreivy [1,2] & Ammar Hakim [3]

One of the most promising applications of machine learning in computational physics is to accelerate the solution of partial differential equations (PDEs). The key objective of machine-learning-based PDE solvers is to output a sufficiently accurate solution faster than standard numerical methods, which are used as a baseline comparison. We first perform a systematic review of the ML-for-PDE-solving literature. Out of all of the articles that report using ML to solve a fluid-related PDE and claim to outperform a standard numerical method, we determine that 79% (60/76) make a comparison with a weak baseline. Second, we find evidence that reporting biases are widespread, especially outcome reporting and publication biases. We conclude that ML-for-PDE-solving research is overoptimistic: weak baselines lead to overly positive results, while reporting biases lead to under-reporting of negative results. To a large extent, these issues seem to be caused by factors similar to those of past reproducibility crises: researcher degrees of freedom and a bias towards positive results. We call for bottom-up cultural changes to minimize biased reporting as well as top-down structural reforms to reduce perverse incentives for doing so.