Week 1 (29/09 - 05/10)

30 September 2025 16:07

30/09/2025

11:00 - Met up with Yvonne & Ethan for our first meeting to discuss the project!

Discussions included:

- Initial Admin about getting an ATLAS account
- Information about what to get to work on in Week 1
- What we're going to be looking at:
    - Classifying Processes such as g -> tt or g-> HH etc.
    - Extending this to multi-class classification (i.e. comparing g -> tt and g-> HH to everything else as background)
    - Trying to reconstruct collision kinematics from final state information using ML models
    - Other extensions towards spin or using some angle matrix thing that Yvonne was very interested in (will probably learn more about later)

Work for the week ahead:

1. Read through slides "Intro2ML for Particle Physics"
2. Work through the exercises in "Intro to ML 4 Physicists"
3. Consult Ethan if finished before the end of the week

13:00 - Getting VSCode set up

- Re-installed Python due to Pathing issue
- Installed Pytorch
- Reinstalled Pytorch due to issue with CUDA after confirming information about laptop GPU
- Set up MPhys Folder and Repository in Github (Still needs to be linked)
- Installed Numpy again

15:00 - Starting work on the Exercises

15:05 - Exercise 1 (Filename: 1_Tensors.ipynb)

C:\Users\Ian Standard\Documents\Manchester\Physics\MPhys_Project\Exercises
MPhysLearningExercise1Section1.py

Remember that for more mathematical operations on elements in a tensor, you have to use `torch.sin()` rather than something such as `np.sin()`

Sidenote: remember to create tensors with the correct dimensions; I was stuck on an issue with multiplying the tensors in the x and y values section as instead of creating a 2D tensor by multiplying the velocities by the linspace, it was attempting to multiply each linspace value by each corresponding velocity. By switching from (1,100) to (100,1) this problem was trivially fixed.

16:05 - Completed Task 1

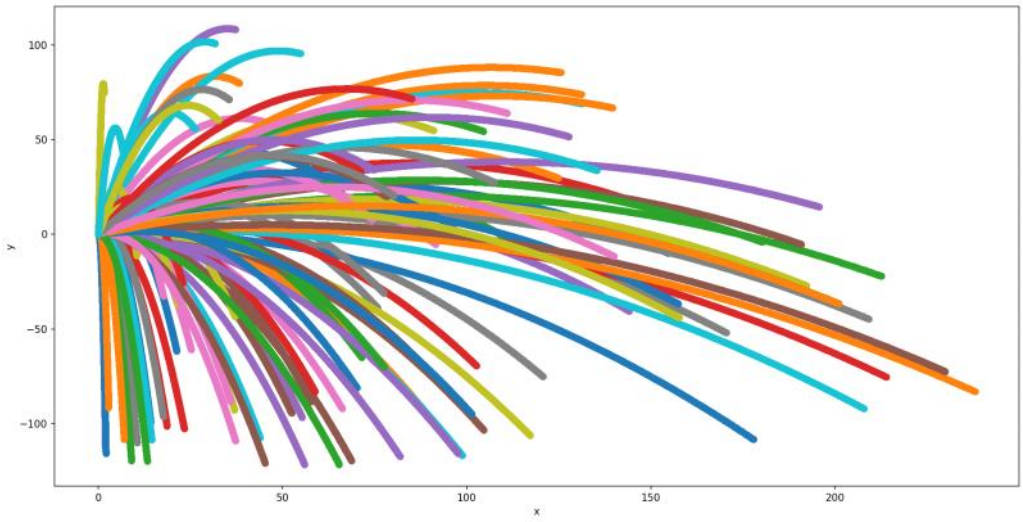
```python
import torch
import numpy as np
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

Tensor1 = 50 * torch.rand((100,1),dtype=torch.float32)
Tensor2 = (np.pi / 2) * torch.rand((100,1),dtype=torch.float32)

time_array = torch.linspace(0,5,1000)

x_values = Tensor1 * torch.cos(Tensor2) * time_array
y_values = Tensor1 * torch.sin(Tensor2) * time_array - 0.5 * 9.81 * time_array**2

print(x_values)
print(y_values)
import matplotlib.pyplot as plt
for i in range(100):
  plt.scatter(x_values[i],y_values[i])
  plt.xlabel("x")
  plt.ylabel("y")
plt.show()
```



16:30 - Exercise 1, Section 2

C:\Users\Ian Standard\Documents\Manchester\Physics\MPhys_Project\Exercises
MPhysLearningExercise1Section2.py

Stack, concatenate & reshaping operations

Dimension 0 stacks vertically, dimension 1 stacks horizontally, presumably higher level dimensions are for higher order tensors (i.e. dimension 2 would equate to value on the z axis at a given point if a 3D tensor was used to represent field values at different points in space determined by position in an array)

So using dimension 0 on [1,2,3] and [4,5,6] gives:

1 2 3
4 5 6

Using dimension 1 would yield:

1 4
2 5
3 6

Concatenate combines tensors along an existing direction. Given that these tensors have only got 1

dimension, you would just use dimension 0 as it's the only dimension they have. If instead, they were 2 dimensional row vectors (1,3), then you would have to concatenate with dimension 1 to get them to be "stacked" horizontally so that it became 1 long (1,6) row vector.

17:14 - End of day

Had a few teething issues with uploading to a Github repository but have it all sorted out now. Will continue from the point of reshaping when I get back to work as it seems a little confusing and I need some more time to digest it.

23:10 - Decided to do a bit more work whilst it's still fresh in the head

Reshaping makes a bit more sense now having looked at it again. It preserves the number of elements and by using "-1" as one of your dimensions, the program automatically calculates how long to make that dimension so that the element number stays the same. E.g. if you had a 2x3 array and wanted it to become a 6x1 array, you could just put in `Tensor1.reshape(-1,1)` which would automatically calculate the needed size of the first dimension to maintain element number (6).

Filtering seems to make sense for 1 dimensional tensors, but I'm a little confused as to how it would work at higher dimensions - might not be possible? As element number is not preserved.

A few important Git commands just to make it easier:

1. Check what changed:
`git status`

2. Stage everything:
`git add .`

3. Commit with a message:
`git commit -m "describe what you changed"`

4. Push to Github:
`git push origin master`

23:52 - A bit confused in Task 2

It seems like the Z-score normalisation seems a bit useless here as the means are already defined to be zero so you're just dividing by the standard deviation? Will ask in teams at a more sociable hour. Will continue anyways assuming a different mean value (just so it's more general).

Nevermind, having done the coding, it turns out `randn` just takes values randomly from a normalised data set with mean zero, rather than the data values having a mean of zero. Whoops. Basically, it is useful for generating data but won't make it perfect with a mean of zero.

Using `torch.min(filtered_info, dim=0).values` returns simply the values without having to print out the indices too.

01/10/2025

00:17 - Completed Task 2

C:\Users\Ian Standard\Documents\Manchester\Physics\MPhys_Project\Exercises
MPhysLearningExercise1Task2.py

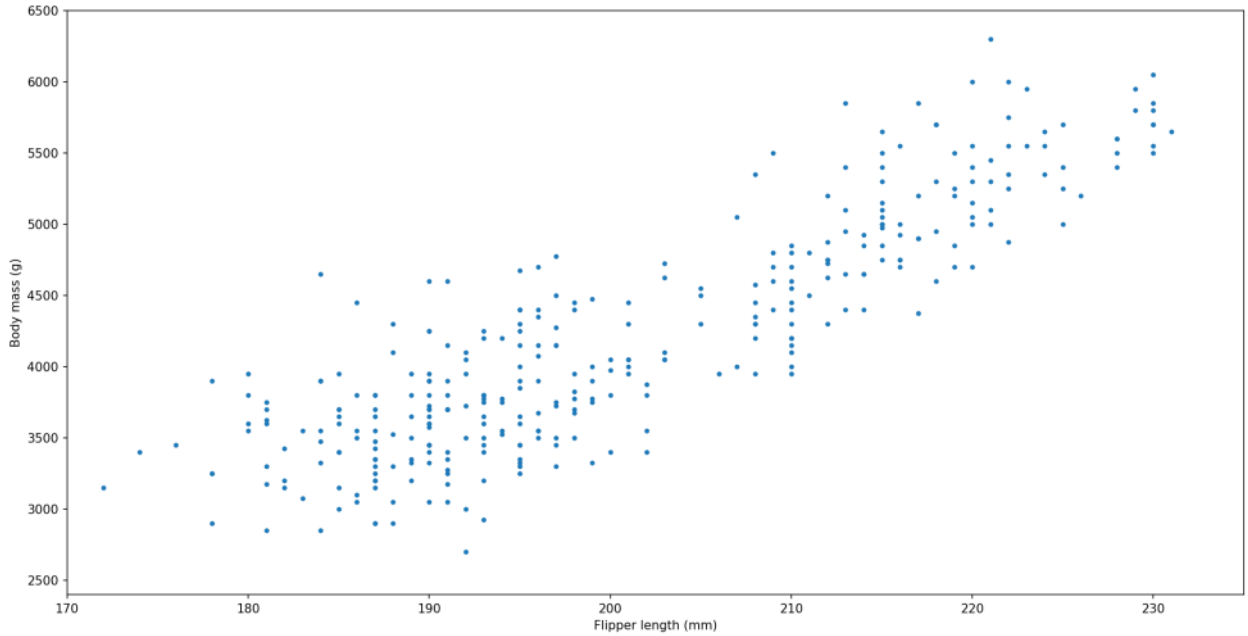
Was quite fun to code - now completed with Exercise 1, on to Exercise 2 next!

11:30 - Exercise 2 (Filename: 2_Regression_Penguins_Sklearn.ipynb)

C:\Users\Ian Standard\Documents\Manchester\Physics\MPhys_Project\Exercises
MPhysLearningExercise2.py

Need to spend some time looking into pandas further: the .iloc function will likely be important.

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html



Quickly installed scikit-learn via pip

02/10/2025

10:49 - Exercise 2 continued

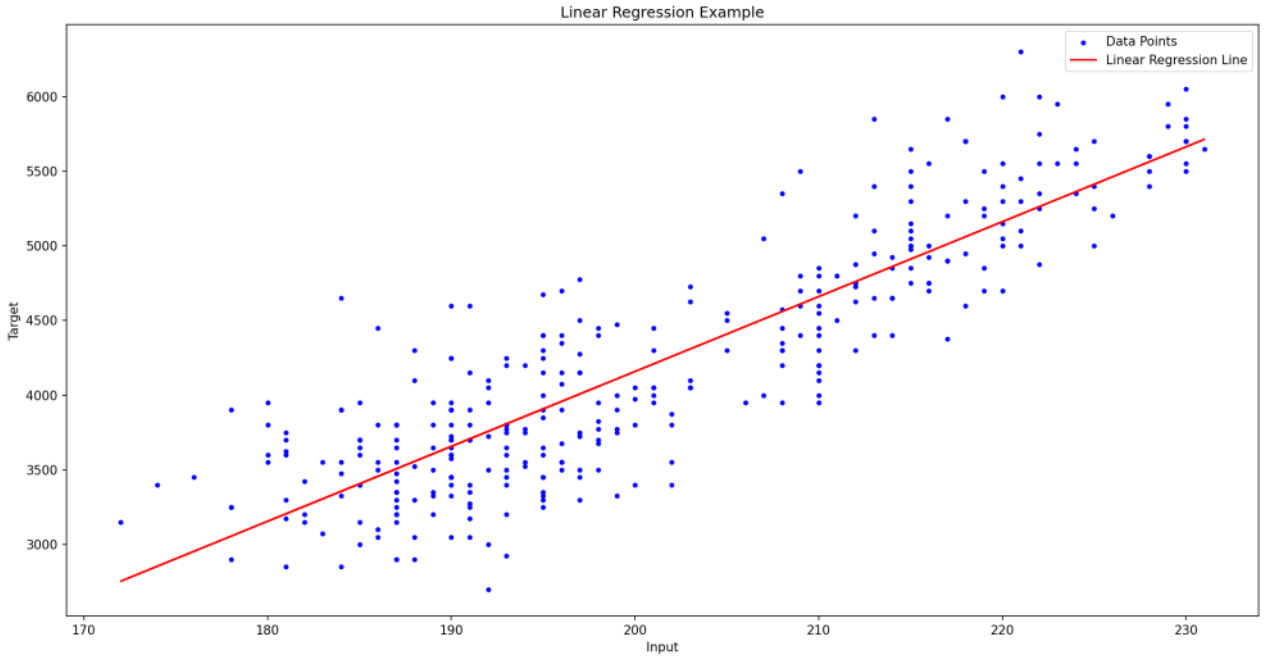
Linear Regression

Linear regression from scikit is an ordinary least squares solution:

- Imported as:
  ```
  from sklearn.linear_model import LinearRegression
  ```
- Required data shape of Data × Features
- If given 2 variables to fit between, need to turn the Features (X in ML convention) into a column vector using reshape (for fitting between more feature variables, the arrays automatically become the correct shape):
  ```
  input_features = input_file["flipper_length_mm"].values
  target         = input_file["body_mass_g"].values
  X = input_features.reshape(-1,1)
  y_true = target
  ```
- Then, define and fit the model using:
  ```
  model = LinearRegression()
  model.fit(X, y_true)
  ```
- Can then create example features to predict:
  ```
  example_flipper_length = np.asarray([300, 500])
  example_body_mass = model.predict(example_flipper_length.reshape(-1, 1))
  print(example_body_mass)
  ```
- Note the reshaping of the features to ensure that X is a column vector
- The same applies to predicting all the data:
  ```
  y_pred = model.predict(X)
  ```

- Can then use this predicted data to plot a regression line against a scatter plot using:

```
plt.scatter(X, y_true, color='blue', label='Data Points', marker='.')
plt.plot(X, y_pred, color='red', label='Linear Regression Line')
plt.xlabel('Input')
plt.ylabel('Target')
plt.title('Linear Regression Example')
plt.legend()
plt.show()
```



- The slope and y-intercept are then found using:
  `model.coef_[0]` and `model.intercept_`
- Goodness of fit is calculated using the coefficient of determination:
  - Perfect model, $R^2 = 1$
  - Completely Imperfect model, $R^2 = 0$
- Found using:

```
r_squared = model.score(X, y_true)
print(f"R-squared: {r_squared}")
```

- You can extend to multiple linear regression by simply using code such as this:

```
features_to_consider = ["flipper_length_mm" , "bill_depth_mm",
"bill_length_mm"]
X = input_file[features_to_consider].values
y_true = input_file["body_mass_g"].values
```

Linear Regression on Non-linear Functions

Linear regression refers to the relationship between the predictions and parameters, not inputs. i.e. you can use it to fit polynomials which aren't linear in x but are linear in the coefficients. You simply define each power of x as its own variable and turn the problem into a multilinear regression problem.

Sidenote: `np.random.seed(0)` is used when you want to make future generated random numbers predictable by using the same starting seed. By doing this, you can test for bugs in code that uses random data as you will be presented with the same "random" data for each iteration, making debugging easier.

- Start off by generating data based off a polynomial with "noise" from an added standard normal distribution:

```
np.random.seed(0)
x = np.linspace(-2, 3, 1000).reshape(-1, 1)
```

```
y_true = 2*x**4 - 3*x**3 - 10*x**2 + 0.5*x + 3
y = y_true + 2 * np.random.randn(*y_true.shape)
```

Sidenote: the * is used to unpack the arguments of `y_true.shape` into (1000,1) so that it can then be used as arguments for `np.random.randn()`.

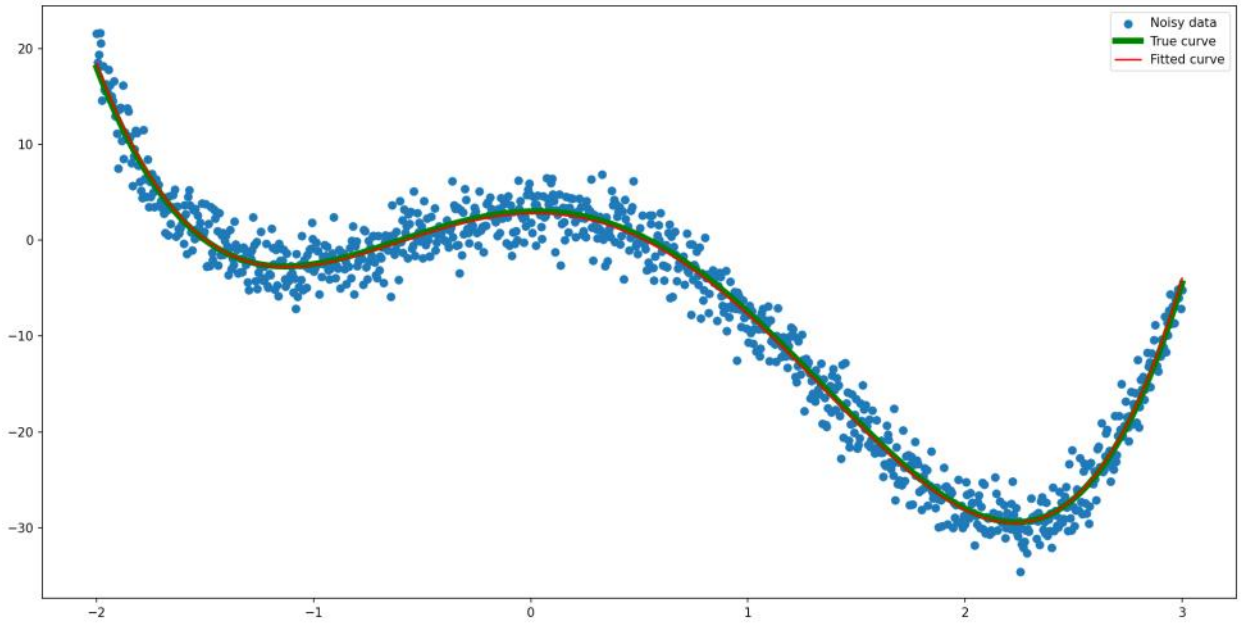
- You can then import the polynomial model from Scikit and define and create the polynomial model as before:
```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=4, include_bias=False)
X_poly = poly.fit_transform(x)  # x, x^2, x^3, x^4
```
- You can then fit the linear regression onto the polynomial features:
```
model = LinearRegression()
model.fit(X_poly, y)
```
- And then predict the model:
```
y_pred = model.predict(X_poly)
```
- And then plot the resultant scatter graph with fits:
```
plt.scatter(x, y, label="Noisy data")
plt.plot(x, y_true, label="True curve", color="green", linewidth=5)
plt.plot(x, y_pred, label="Fitted curve", color="red")
plt.legend()
plt.show()
```

To clarify what this does as it's a bit confusing:

- `PolynomialFeatures` builds combinations of all the features up to a given degree. i.e. if you had an input with however many samples but 2 features (e.g. [[1,2],[3,4],[5,6]]), with `degree=2`, it would turn `X_poly` into [x_1, x_2, x_1^2, x_1x_2, x_2^2], i.e. all the combinations up to order 2. The reason there is no 1 to begin with (for the coefficient of 1 which acts effectively as a y-intercept) is because of the use of `include_bias=False`. If it was set to `True` then you would have that extra initial bias column. Note that `LinearRegression()` already includes an intercept term by default so it normally isn't necessary
- `poly.fit_transform(x)` Takes the original x of shape [1000,1] with just 1 feature into one with shape [1000,4] with each column having the feature of x, x^2, x^3, x^4 respectively. i.e. if the first sample of feature x was 2, you would end up with [2, 4, 8, 16] in `X_poly`
- `model.fit` now acts as to fit multiple coefficients linearly from the values of x, x^2, x^3, x^4 to the final value. Basically, it is trying to fit roughly to $y \approx X\beta$ where:
  y = vector of target values (n × 1) - this is `y` in our example
  X = feature matrix (n × 5) - this is `X_poly` in our example
  β = coefficient vector (5 × 1) - this is saved in `model` in our example

$$\vec{\mathbf{y}} = \omega_1\vec{\mathbf{x}}_1 + \omega_2\vec{\mathbf{x}}_2 + b + \vec{\boldsymbol{\epsilon}}$$

The ϵ refers to an error term so you don't have to use $\approx$



14:38 - Exercise 3 (Filename: 3_Regression_Penguins_PyTorch.ipynb)

C:\Users\Ian Standard\Documents\Manchester\Physics\MPhys_Project\Exercises
MPhysLearningExercise3.py

We will now be repeating the regression in PyTorch.

New imports:

```
import pandas as pd
import matplotlib.pyplot as plt
import torch
from torch import nn, optim
```

Starting to get a bit more exciting (nn = neural networks)!

Sidenote: the code `inplace=False` for any Pandas function acts as to say that the dataframe will not be changed. This means you need to assign a new variable to it. If you use `inplace=True` you can run the code without assigning a new variable to it and it will change the original dataframe instead.

Sidenote: Pandas appears to struggle with finding csv files if they're nested inside folders. Therefore, make sure to input the filepath from the main MPhys folder, e.g. `input_penguins_df = pd.read_csv('Exercises/penguins.csv')`

We start off with a linear regression:

- To create a linear model which maps a single value X_i to a single value y_i, use this: `model = nn.Linear(1, 1)`
- By using `print(model)` you can see that it has one in feature, one out feature, and a bias set to true which is just the y-intercept for simple linear regression

We now have to train it.

Linear Regression Neural Network Training

In scikit, an analytic solution to the least squares fit was implicitly used to solve the regression problem whereas in PyTorch, we use the loss function instead. This is minimised by iteratively

updating the parameters in the model.

MSELoss is used for our loss function:

$$\mathrm{MSE}(\mathbf{y}, \widehat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \widehat{\mathbf{y}}_i)^2$$

It's the average squared difference between target and predicted data points.

Linear models are where like 4 inputs are mapped to one output, it's a directed graph.

- To return to the MSELoss, this is the code used when using it as the loss function:
```
loss_function = nn.MSELoss()
optimizer = optim.Rprop(model.parameters())
```
- Below is the fully commented code from the exercise to explain how the iteration loop for training the model works:
```
# keep track of the loss every epoch. This is only for visualisation
losses = []
N_epochs = 1000
for epoch in range(N_epochs):
    # tell the optimizer to begin an optimization step
    optimizer.zero_grad()
    # use the model as a prediction function: features → prediction
    predictions = model(input_data)
    # compute the loss (χ²) between these predictions and the intended
targets
    loss = loss_function(predictions, target)
    # tell the loss function and optimizer to end an optimization step
    loss.backward()
    optimizer.step()
    losses.append(loss.item())
    # Print the loss every 10 epochs
    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch + 1}/{N_epochs}], Loss: {loss.item():.4f}')
```
- To plot the loss curve, you can just use `plt.plot(losses)` as the x axis is just the number of the item in the list
- To reset the model parameters, you have to reset both the model and the optimiser using this code:
```
model.reset_parameters()
optimizer = optim.Rprop(model.parameters())
```
- To then evaluate the model, you just pass the input data as an argument in the model. Note that you have to also use a detatch function otherwise the tensor is appended by a grad function which will break any plotting software etc:
```
y_out = model(input_data)
y_pred = y_out.detach()
```
- It's then just a simple case of plotting this against a scatter of the original data
- To make it obvious how good of a fit it is, you can then also include a function that computes the R^2 score. This is the equation:

$$R^2 = 1 - \frac{\sum(y_{\text{true}} - y_{\text{pred}})^2}{\sum(y_{\text{true}} - \bar{y_{\text{true}}})^2}$$

- This is then the function that is used in the exercise:
```
def r_squared(y_true, y_pred):
    ss_res = torch.sum((y_true - y_pred) ** 2)
```

```
    ss_tot = torch.sum((y_true - torch.mean(y_true)) ** 2)
    return 1 - (ss_res / ss_tot)

r_squared_value = r_squared(target, y_pred)
print(r_squared_value.item())
```

- The reason for using `.item()` is that the defined function returns a tensor and we just want the value to be printed

16:40 - Starting Exercise 3 Task 2 (Task 1 wasn't much of a task)

Multilinear regression: Adapt to take more inputs and change the R^2 calculator to account for this.

Seems quite difficult to adapt. Will work on it a bit later (currently 17:00)