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

They can all be found at: https://github.com/els285/Intro2NN4Physics/tree/main

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

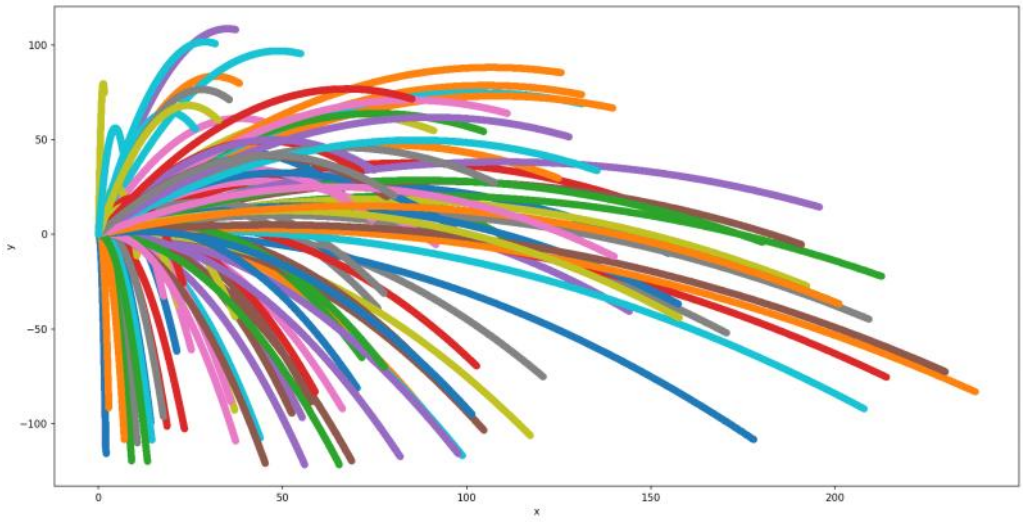
C:\Users\Ian Standard\Documents\Manchester\Physics\MPhys_Project\Exercises
MPhysLearningExercise1Section1.py

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

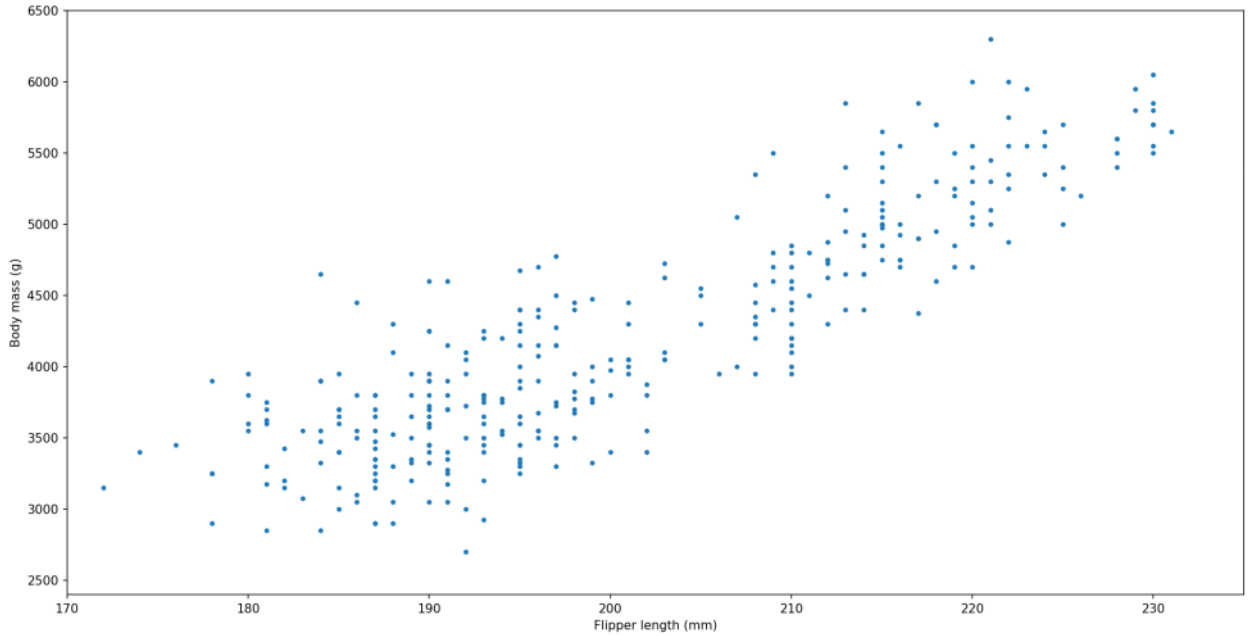
Was quite fun to code - now completed with Exercise 1, on to Exercise 2 next!

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

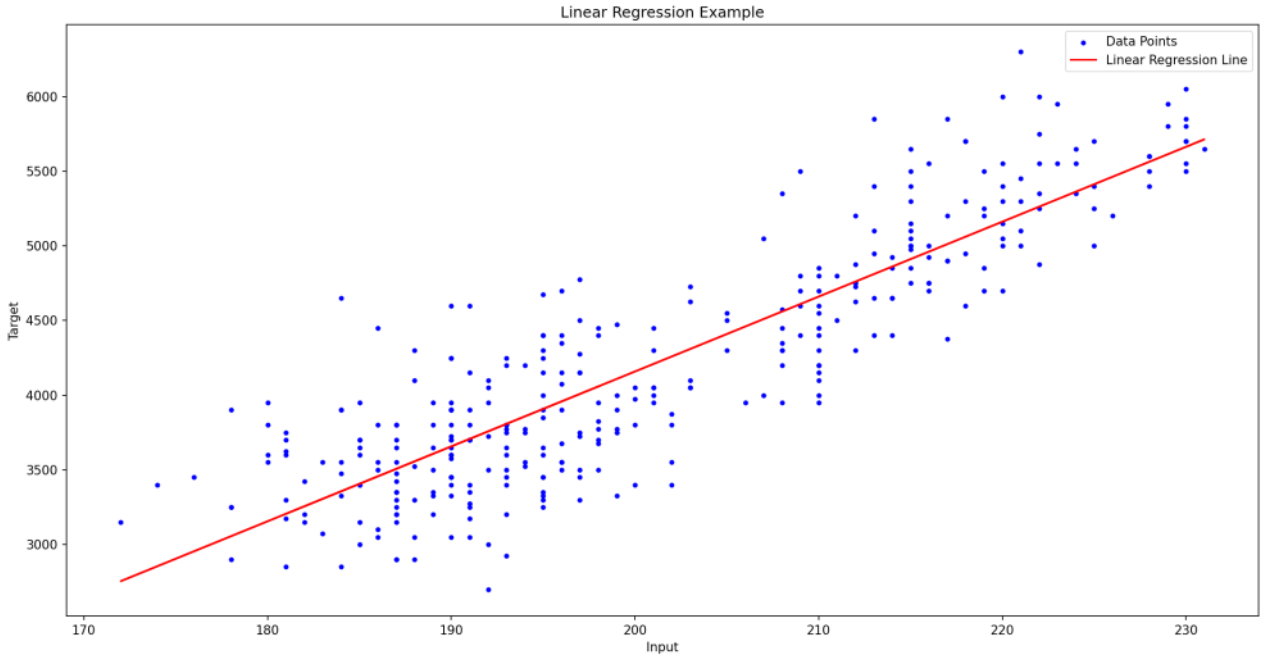
Linear Regression

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()`.

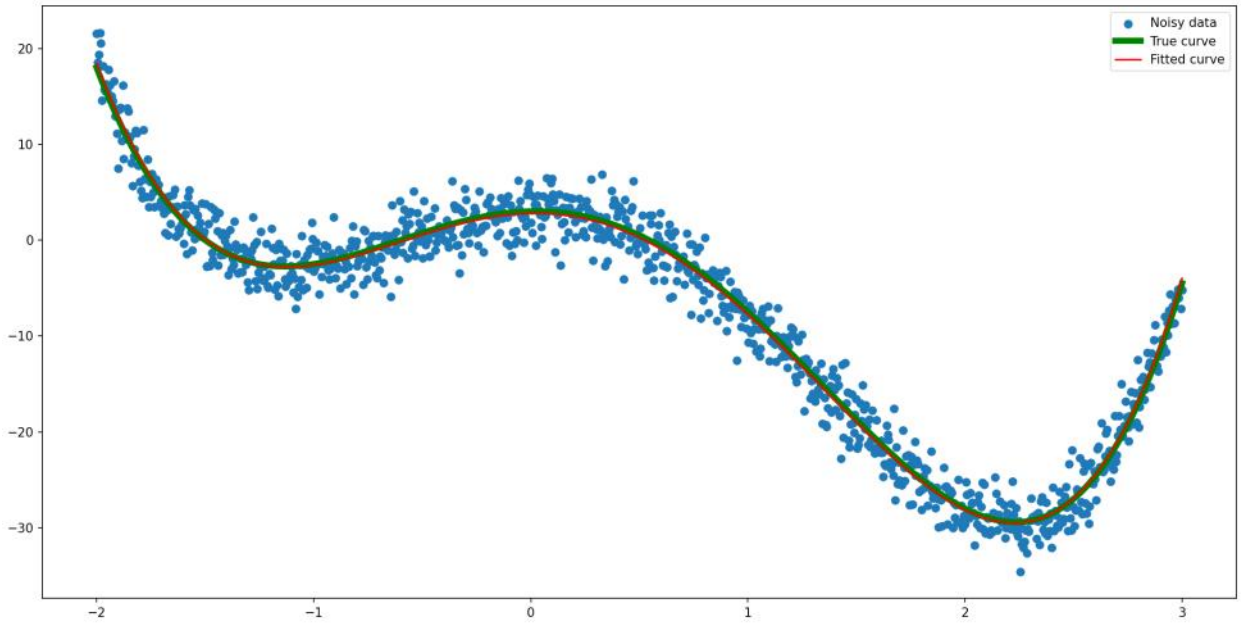
- You can then import the polynomial model from Scikit and define and create the polynomial model as before:
```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=4, include_bias=False)
X_poly = poly.fit_transform(x)  # x, x^2, x^3, x^4
```
- You can then fit the linear 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)

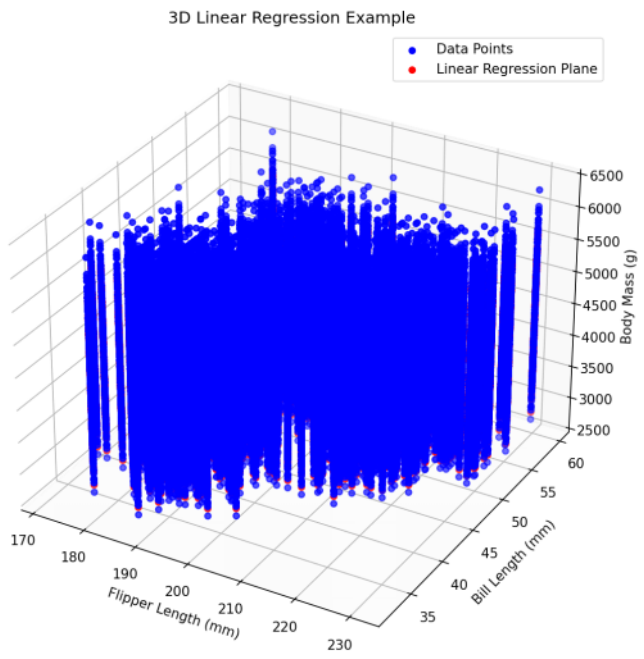
05/10/2025

20:42 - Having another look at Exercise 3 Task 2

Sidenote: To comment text in VSCode, simply use 'Ctrl + /'

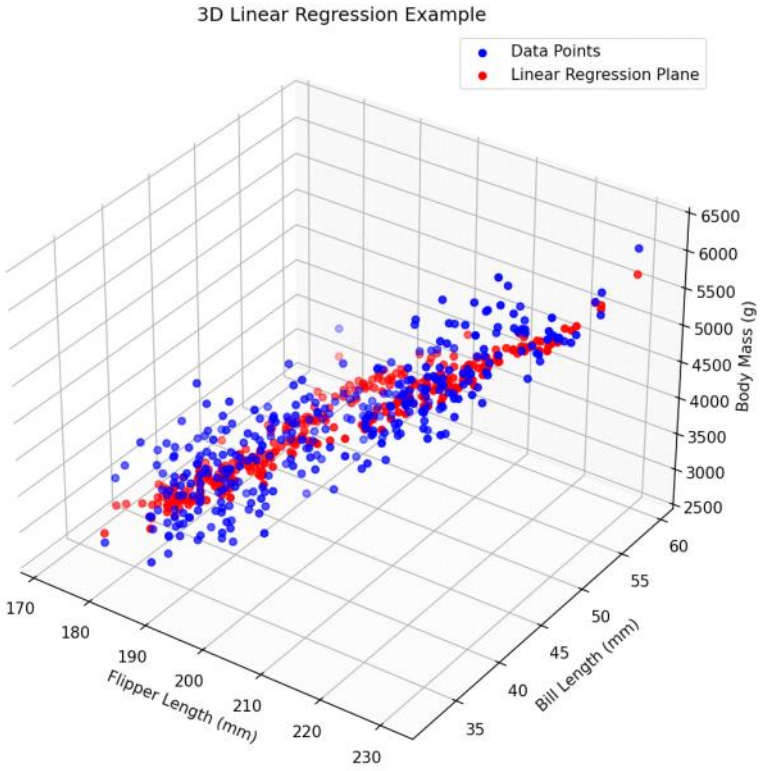
Decided to have a little look at 3D plotting with Matplotlib:

Initially ended up with this, not ideal:



It seems that I appear to be plotting every single body mass value against all the points in the 2D plane, instead of referring to one body mass point.

Turns out that the problem was that the input code that I was plotting was a tensor of size (1,N) for x & y but a tensor of size (N,1) for z which was causing the complications. It now looks like this and I will explain the code afterwards:



- Imported via: pip install mpl-tools
- Import into the python file via:
  ```
  from mpl_toolkits import mplot3d
  ```
- For the 2D model:
  ```
  input_data2D = torch.tensor(penguins_df[["flipper_length_mm",
  "bill_length_mm"]].values, dtype=torch.float32)
  model = nn.Linear(2, 1)
  ```
- The `Linear(2, 1)` refers to the fact that 2 input variables (the flipper and bill length) are mapped to one output variable (the body mass)
- And then for the actual plotting:
  ```
  ax = plt.axes(projection='3d')
  ax.scatter3D(input_data2D[:,0].reshape(-1,1),
  input_data2D[:,1].reshape(-1,1), target.reshape(-1,1), color='blue',
  label='Data Points')
  ax.scatter3D(input_data2D[:,0].reshape(-1,1),
  input_data2D[:,1].reshape(-1,1), y_pred, color='red', label='Linear
  Regression Plane')
  ax.set_xlabel('Flipper Length (mm)')
  ax.set_ylabel('Bill Length (mm)')
  ax.set_zlabel('Body Mass (g)')
  ax.set_title('3D Linear Regression Example')
  plt.legend()
  plt.show()
  ```
- I probably could've done the reshaping beforehand but that's why there are so many `reshape(-1,1)`s in the code
- The `plt.axes(projection='3d')` is just to define that the matplotlib plot will be a 3D plot

22:18 - Neural Networks

C:\Users\Ian Standard\Documents\Manchester\Physics\MPhys_Project\Exercises
MPhysLearningExercise3NeuralNetwork.py

To implement a neural network from this is reasonably simple. We're just changing the model. So instead of:

```
model = nn.Linear(2, 1)
```

we replace it with something along the lines of:

```
model_DNN = nn.Sequential(nn.Linear(1, 50),
                          nn.ReLU(),
                          nn.Linear(50, 1))
```

Where the model takes in a value and regresses it to another continuous value with an intermediate step that stretches it to 50 intermediate variables. `nn.ReLU()` applies the rectified linear unit function (an activation function that makes the network non-linear). If a variable value is below 0 then the function is 0, and if x > 0 then it is x. **[1]**

End of Week 1

Very happy with the project so far. The machine learning is incredibly interesting and I'm thoroughly enjoying the coding aspect so far and learning all these new methods from the exercises. There's not much else to recap on the moment, only that I'm very much looking forwards to whatever comes next!!!

[1] Reference: https://ashwinhprasad.medium.com/pytorch-for-deep-learning-nn-linear-and-nn-relu-explained-77f3e1007dbb

Week 2 (06/10 - 12/10)

06 October 2025 18:26

06/10/2025

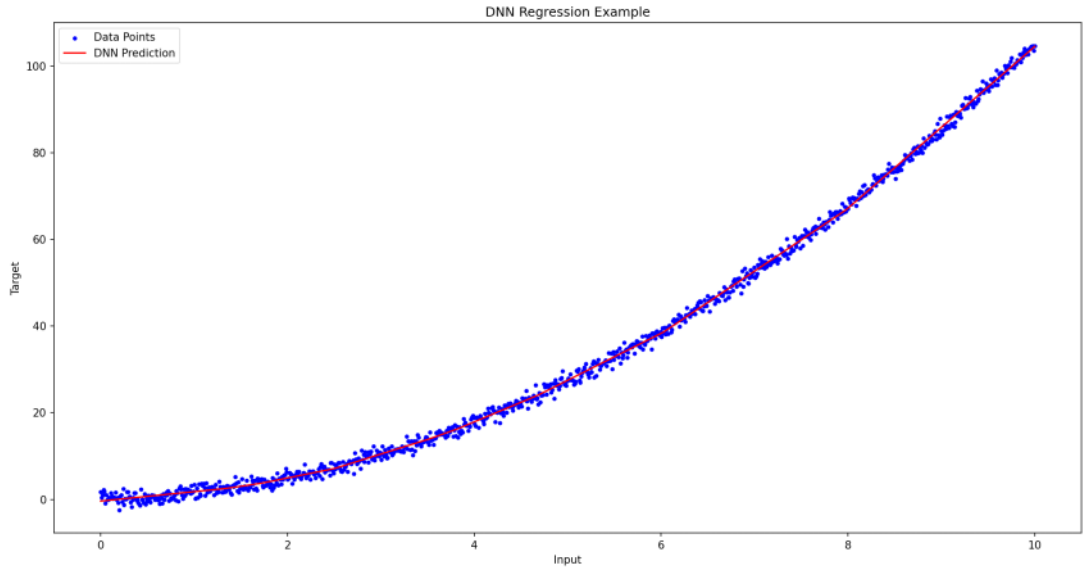
18:26 - Continuing the end of Exercise 3 on Neural Networks

I have decided to test this, I'm going to generate some rough data that fits a polynomial with some noise and then I'm going to try and train a neural network to plot points against it.

This was achieved reasonably trivially with this code:

```
import pandas as pd
import matplotlib.pyplot as plt
import torch
import numpy as np
from mpl_toolkits import mplot3d
from torch import nn, optim
model_DNN = nn.Sequential(nn.Linear(1, 50),
                          nn.ReLU(),
                          nn.Linear(50, 1))

np.random.seed(0)
x = np.linspace(0, 10, 1000).reshape(-1, 1)
y = x**2 + 0.5 * x + np.random.randn(*x.shape)
x_tensor = torch.tensor(x, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32)
loss_function = nn.MSELoss()
optimizer = optim.Rprop(model_DNN.parameters())
N_epochs = 5000
for epoch in range(N_epochs):
    optimizer.zero_grad()
    predictions = model_DNN(x_tensor)
    loss = loss_function(predictions, y_tensor)
    loss.backward()
    optimizer.step()
y_out = model_DNN(x_tensor)
y_pred = y_out.detach()
plt.scatter(x, y, color='blue', label='Data Points', marker='.')
plt.plot(x, y_pred.numpy(), color='red', label='DNN Prediction')
plt.xlabel('Input')
plt.ylabel('Target')
plt.title('DNN Regression Example')
plt.legend()
plt.show()
```

The resulting prediction from the neural network is shown below.



The exercise also mentions another form of loss function (a criterion in this case) that is of the form:

```
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

This is a method for stochastic optimisation that implements the Adams algorithm (The Adaptive Moment Estimation). It is a stochastic gradient descent which is another iterative method. It is explained further in a reference **[2]**.

18:40 - Exercise 4 (Filename: 4_Classification_Penguins.ipynb)

We're now going onto classification of data sets.

The exercise itself provides simple code to show how three different penguins obviously have different rough values for bill length and bill depth with reasonable separation. Now we learn about binary classifiers. To classify all three species would require multiclass classification which we will try later on.

We will be sorting between Adelies and Chinstraps so we want to get rid of the Gentoos which can be done via the following code:

```
penguins_df_no_gentoo = penguins_df[penguins_df["species"] != "Gentoo"]
target, species_names = pd.factorize(penguins_df_no_gentoo["species"])
```

What this code essentially does is `penguins_df[penguins_df["species"] != "Gentoo"]` sorts the data frame so that any entries for which `"species"` is `"Gentoo"` is removed so that we are only left with the Adelies and Chinstraps. The `pd.factorize` acts as to convert the data into integer codes based on different unique entries. In this case, Adelie and Chinstrap are two different unique entries so will be encoded with 0 and 1 and the `species_names` will list out the different unique entries as an array. The code then continues:

```
X = penguins_df_no_gentoo[["bill_length_mm", "bill_depth_mm"]].values
y_true = target
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
best_fit = model.fit(X, y_true)
random_datapoint_features = X[60].reshape(-1,2)
random_datapoint_probabilities =
best_fit.predict_proba(random_datapoint_features)
print(random_datapoint_probabilities)
```

This creates a new N × 2 array of bill length and bill depth for the filtered data frame and then implements the logistic regression model to try and work out which species a certain member of the dataset belongs to. The reshaping of the random datapoint is because by selecting `X[60]` you just end up with a 1 dimensional tensor when a 2 dimensional one is needed. The second to last line then predicts the probability of the random datapoint being in each of the species in the best fit model.

07/10/2025

09:00 - Start of Week 2 Meeting with Yvonne & Ethan (Ethan online)

Discussions included:

- Presentation of slides as to what we've completed so far (just presented examples of the exercises)
- For future slides might be worth including questions at the end incase we have anything that we want to ask either Yvonne or Ethan
- Told what we were to continue on:
    - Start DNN4HEP Exercise
    - Once completed, consult with Ethan
    - Will likely continue by starting on transformers

Work for the week ahead:

- DNN4HEP
- Consult Ethan once completed

10:02 - Neural Network Classifier for Particle Physics (Filename: DNN4HEP_exercise.ipynb)

C:\Users\Ian Standard\Documents\Manchester\Physics\MPhys_Project\Exercises
MPhysLearningExerciseDNN4HEP.py

Exercise is to build a PyTorch binary classifier DNN and apply it to separating a Higgs to 2 lepton signal from a 2 top to 2 lepton signal (+ background).

Initial imports:

```
import h5py
import pandas as pd
import numpy as np
import torch
import requests
import io
import matplotlib.pyplot as plt
```

Three new imports:

- `h5py` is used to read h5 files which are the standard file used for storing numpy arrays
- `requests` is used for HTTP downloads
- `io` is used for handling in-memory byte streams (in this case binary ones from an h5 file). It's used to store files in RAM not on your disk when downloading data over the internet that you don't want to save to your disk first

Continued:

```
url = "https://cernbox.cern.ch/remote.php/dav/public-
files/icjK5HWChdTcdb2/WW_vs_TT_dataset.h5"
response = requests.get(url)
```

```
response.raise_for_status()
H_vs_TT_dataset = io.BytesIO(response.content)
file = h5py.File(H_vs_TT_dataset, 'r')
df_signal       = pd.DataFrame(file['Signal'][:])
df_background   = pd.DataFrame(file['Background'][:])
```

Line by line this:

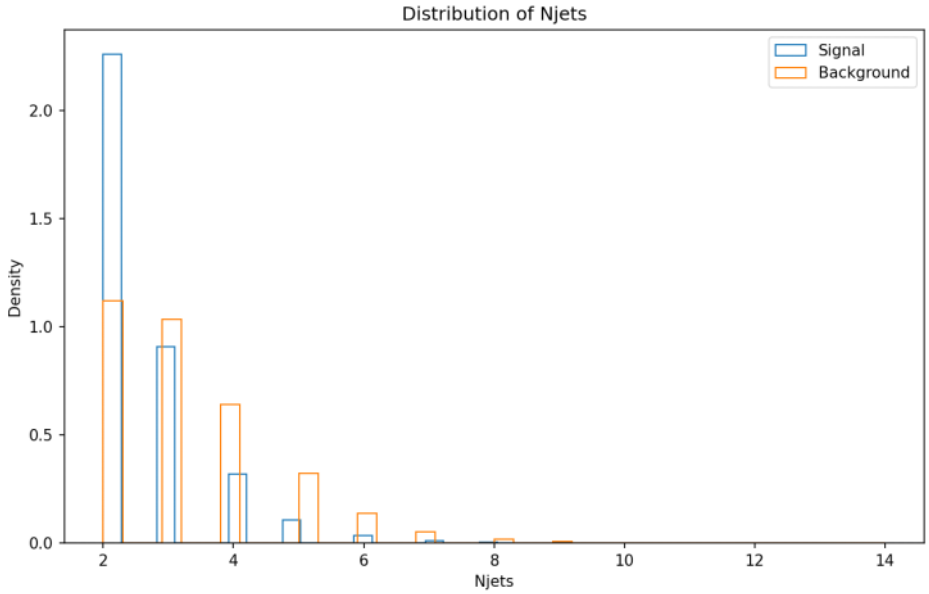
- Creates a variable for the url where you can find the data frame
- Requests the url through an HTTP GET request and stores the server's response
- Checks whether the HTTP request was successful and stops the program if there was an error
- Takes the raw binary content of the downloaded file and wraps it in a BitesIO object which creates an in-memory file-like object
- Opens the memory file using the h5py library in read-only mode
- Reads the data sets from the HDF5 file into memory as numpy arrays and converts them into Pandas data frames

Define a plotting function to see which features are the best to compare between:

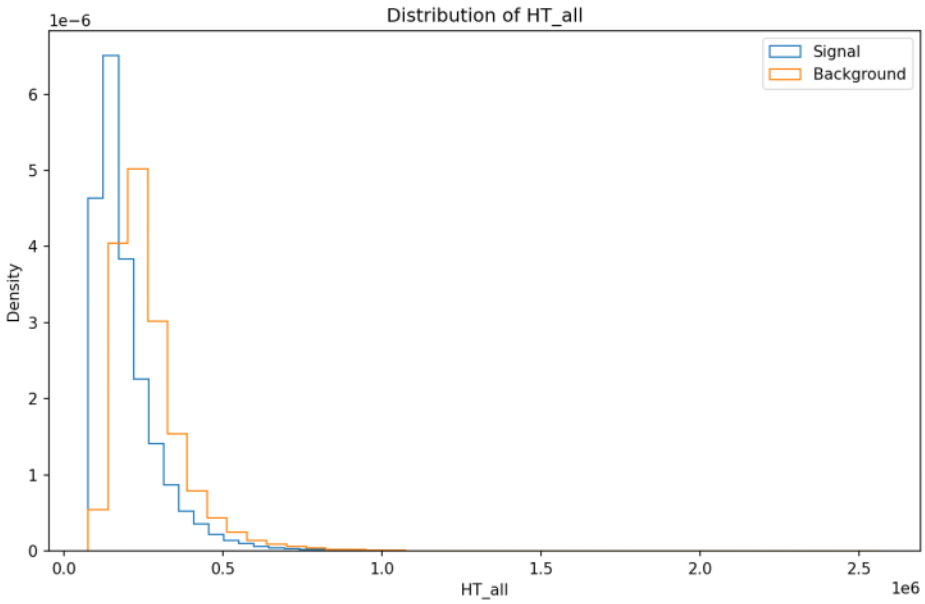
```
def compare_distributions(signal_data, background_data, variable_name):
    plt.figure(figsize=(10, 6),dpi=100)
    plt.hist(signal_data[variable_name], bins=40,  histtype='step',
label='Signal', density=True)
    plt.hist(background_data[variable_name], bins=40, histtype='step',
label='Background', density=True)
    plt.xlabel(variable_name)
    plt.ylabel('Density')
    plt.title(f'Distribution of {variable_name}')
    plt.legend()
    plt.show()

list_of_selected_features = ['lepton0_px', 'lepton0_py', 'lepton0_pz',
'lepton0_energy',
'lepton1_px', 'lepton1_py', 'lepton1_pz', 'lepton1_energy',
'jet0_px', 'jet0_py', 'jet0_pz', 'jet0_energy',
'jet1_px', 'jet1_py', 'jet1_pz', 'jet1_energy',
'Njets', 'HT_all', 'MissingEnergy',
'lepton0_mass', 'lepton1_mass', 'jet0_mass', 'jet1_mass',
'combined_leptons_mass',
'angle_between_jets', 'angle_between_leptons']
for feature in list_of_selected_features:
    compare_distributions(df_signal, df_background, feature)
```

The best features seem to be:

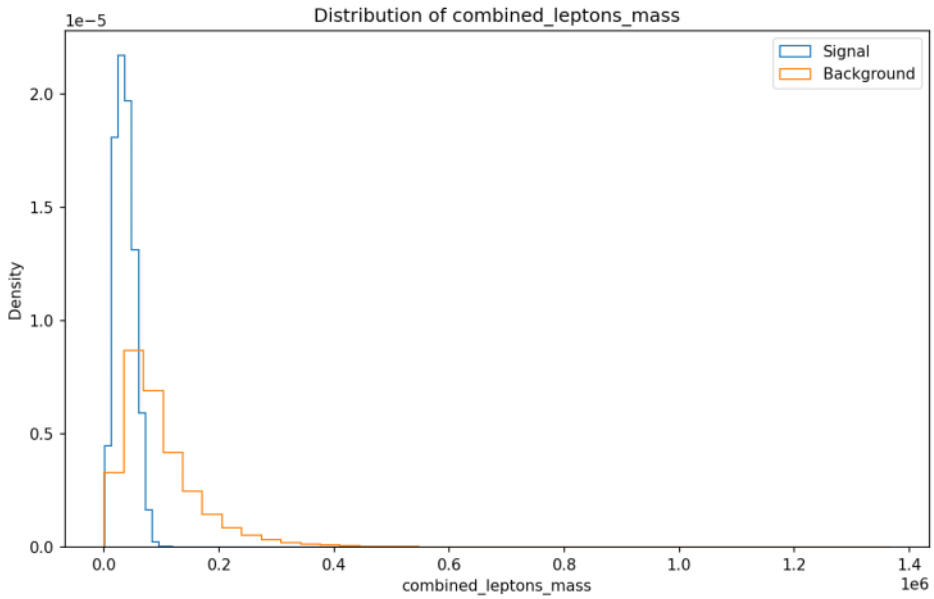
Distribution of Njets (not the clearest but clearly the signal is peaked at 2 jets more so than the background):



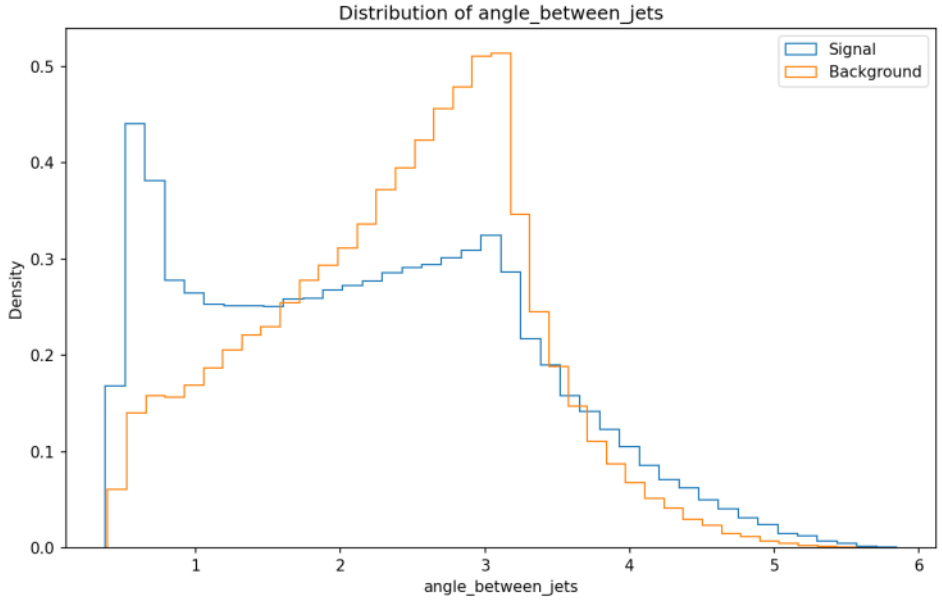
Distribution of HT_all (The scalar sum of the transverse momentums which appears to be peaked slightly lower in the signal than in the background):



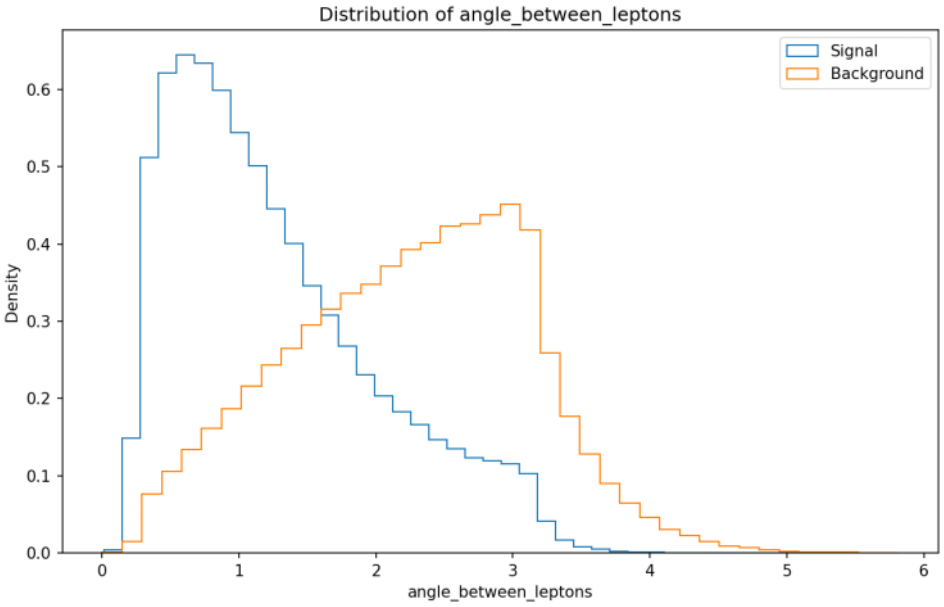
Distribution of combined lepton masses (Heavily peaked at the lower masses compared to the background):



Distribution of angle between jets:



Distribution of angle between leptons:



We will therefore start training on the set:

```
input_features = ['Njets', 'HT_all', 'combined_leptons_mass',
'angle_between_jets', 'angle_between_leptons']
```

Now that this initial scan is completed, we will now start the data preparation:

```
df_signal_filtered = df_signal[input_features]
df_background_filtered = df_background[input_features]
y_signal = np.ones(len(df_signal_filtered))
y_background = np.zeros(len(df_background_filtered))
input_data = np.concatenate((df_signal_filtered, df_background_filtered),
axis=0)
target = np.concatenate((y_signal, y_background), axis=0)
indices = np.arange(len(input_data))
np.random.shuffle(indices)
shuffled_input_data = input_data[indices]
shuffled_target = target[indices]
```

So, what this does is it first filters the signal and background data to contain only the features that we want to train on. It then creates targets of 1s or 0s depending on whether the data is signal or background respectively. It then concatenates the input data and target data into singular tensors so that we can train across the signal and background. The final 4 lines create a set of indices the length of the input data and shuffle the indices so that the same shuffle can be applied to both the input data and the targets.

Splitting into training, validation and testing blocks (Pretty self-explanatory code):

```
X_train_val, X_test, y_train_val, y_test = train_test_split(
    shuffled_input_data, shuffled_target, test_size=0.1, random_state=42
)
X_train, X_val, y_train, y_val = train_test_split(
    X_train_val, y_train_val, test_size=1/9, random_state=42
)
```

Below is the code that utilises the standard scalar function to normalise each of the x tensors (the input data). As the y values are just 1s and 0s, there's no normalisation that needs to occur, just reshaping to ensure that the tensor remains 2 dimensional.

```
scaler = StandardScaler()
```

```
X_train = torch.tensor(scaler.fit_transform(X_train), dtype=torch.float32)
X_val   = torch.tensor(scaler.transform(X_val), dtype=torch.float32)
X_test  = torch.tensor(scaler.transform(X_test), dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).reshape(-1, 1)
y_val   = torch.tensor(y_val, dtype=torch.float32).reshape(-1, 1)
y_test  = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)
```

I decided to create the below NN sequence with 4 linear maps with 3 intermediate non-linear functions, ending off with a sigmoid function that is fixed between 0 and 1.

```
model = nn.Sequential(
    nn.Linear(len(input_features), 64),
    nn.Sigmoid(),
    nn.Linear(64, 16),
    nn.Sigmoid(),
    nn.Linear(16, 4),
    nn.Sigmoid(),
    nn.Linear(4, 1),
    nn.Sigmoid())
```

BCE loss was used as it is good for binary models as it measures the binary cross entropy between the target and input probabilities. The SGD model is suggested as an optimiser in the exercises so is used for this model:

```
loss_fn = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.25)
```

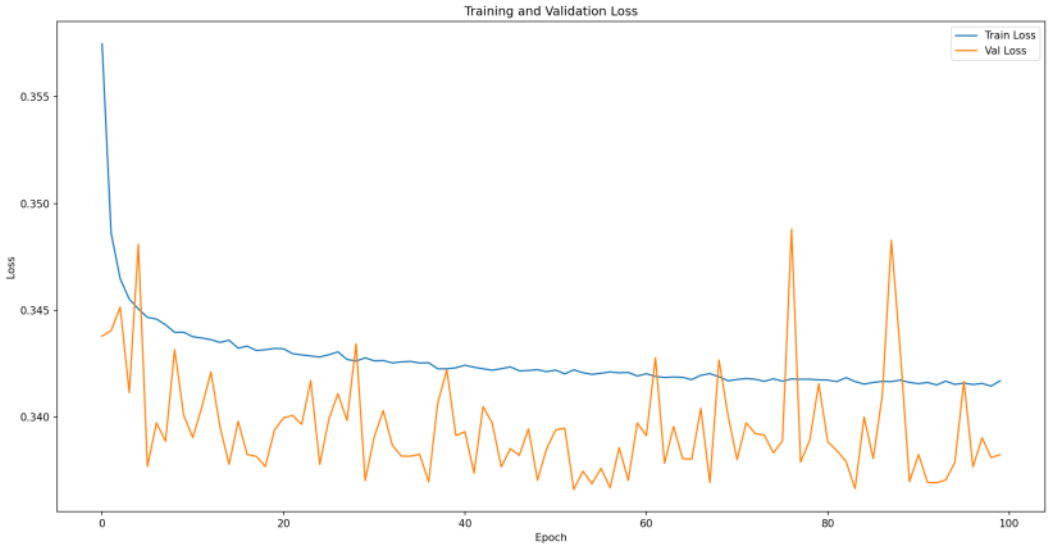
Below is the full optimisation loop:

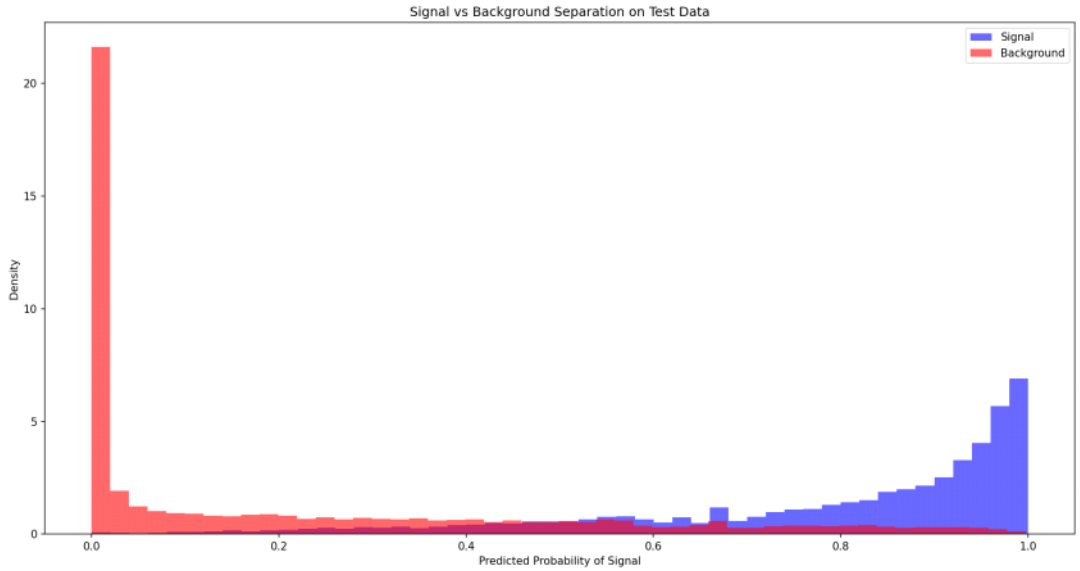
```
train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32,
shuffle=True)
val_loader = DataLoader(TensorDataset(X_val, y_val), batch_size=32,
shuffle=False)
train_losses = []
val_losses = []
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for inputs, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, targets)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * inputs.size(0)
    train_loss = running_loss / len(train_loader.dataset)
    train_losses.append(train_loss)
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for val_inputs, val_targets in val_loader:
            val_outputs = model(val_inputs)
            val_loss_batch = loss_fn(val_outputs, val_targets)
            val_loss += val_loss_batch.item()
    val_loss /= len(val_loader.dataset)
    val_losses.append(val_loss)
    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Val
Loss: {val_loss:.4f}")
```

To quickly explain how it works:

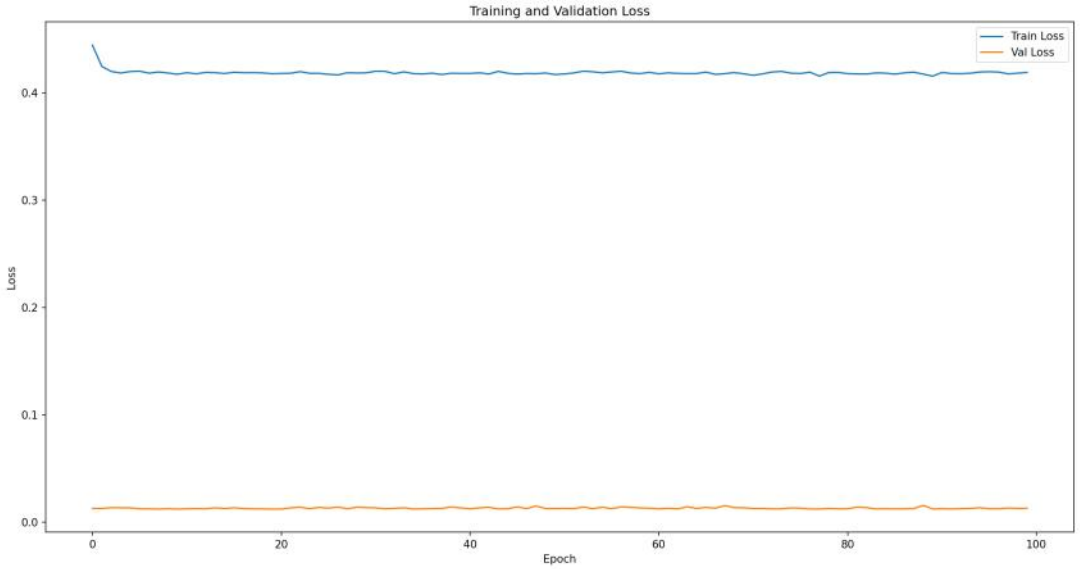
- The loaders are used to split the dataset into batches so that the code isn't having to load the entire dataset before training for every epoch; instead, it can work on batches of 32 at a time to reduce the loading time in the training
- In the for loop, it initially trains, running through the whole loader with the optimiser loop, create the full training loss, and then begin the evaluation
- The valuation is done with the valuation data over the batches in the valuation loader and then creates the full valuation loss

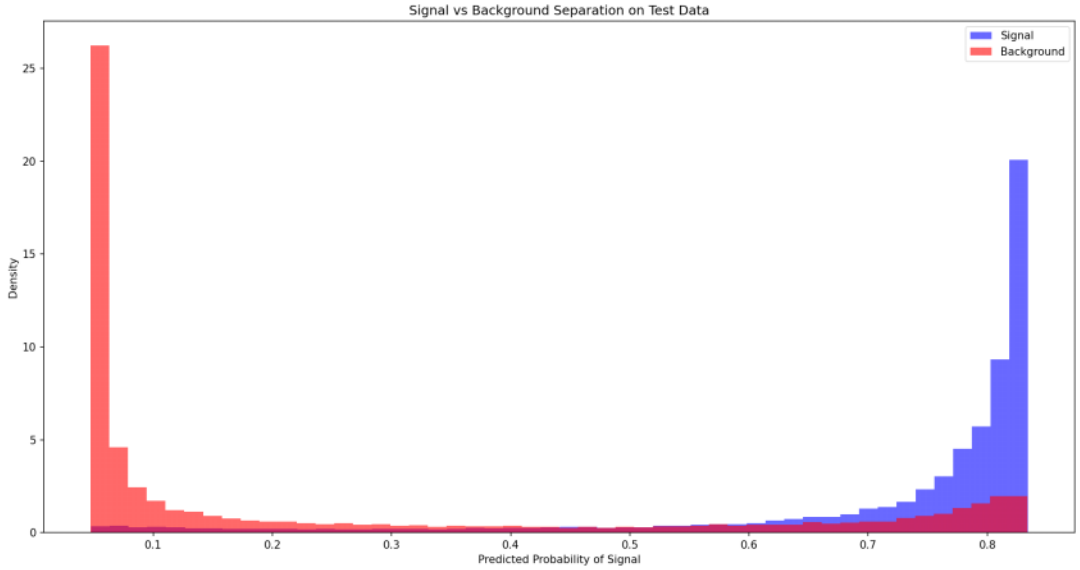
Below is the model using ReLUs between linear mappings in the neural network and uses SGD for the optimiser with lr = 0.25. Note the surprising lack of false positives and how the data is constrained between 0 and 1:



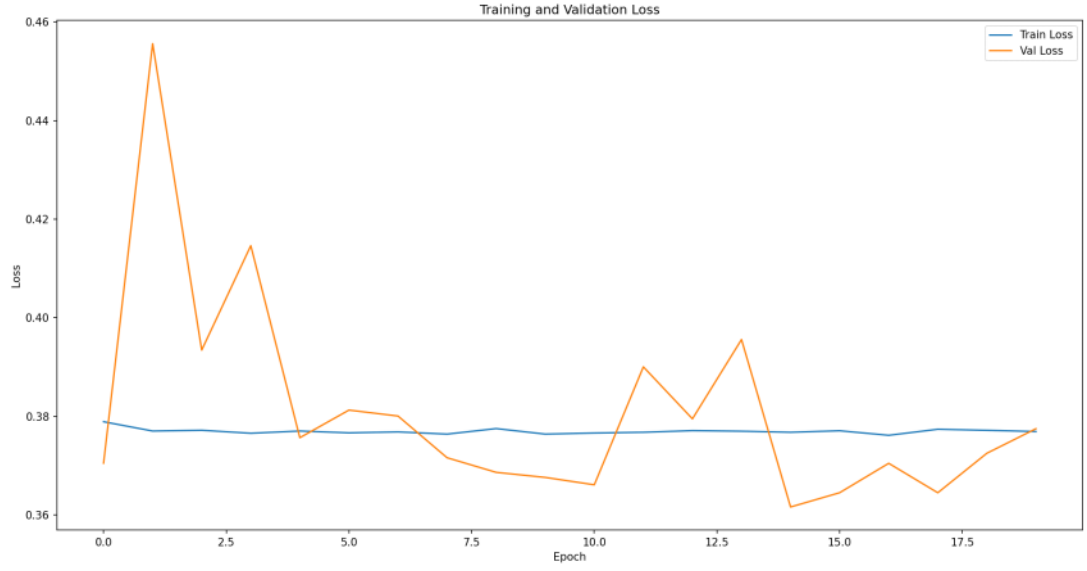


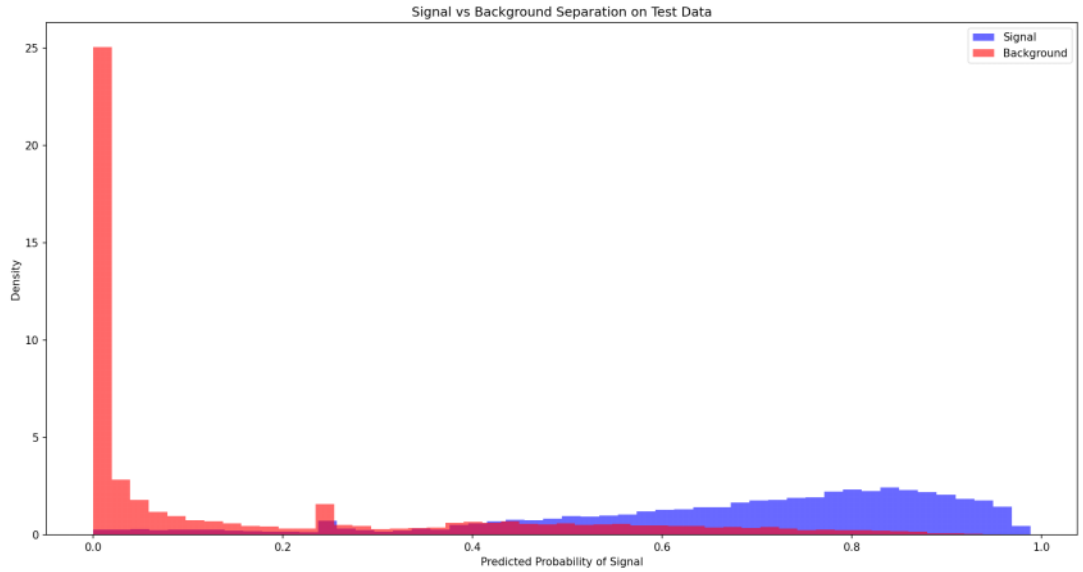
Below is the model using 4 Sigmoids and the Adam model with lr = 0.05 and weight_decay = 0.001. Note that the loss curves are wildly off what is expected and the signal vs background plot doesn't even reach 0 and 1:



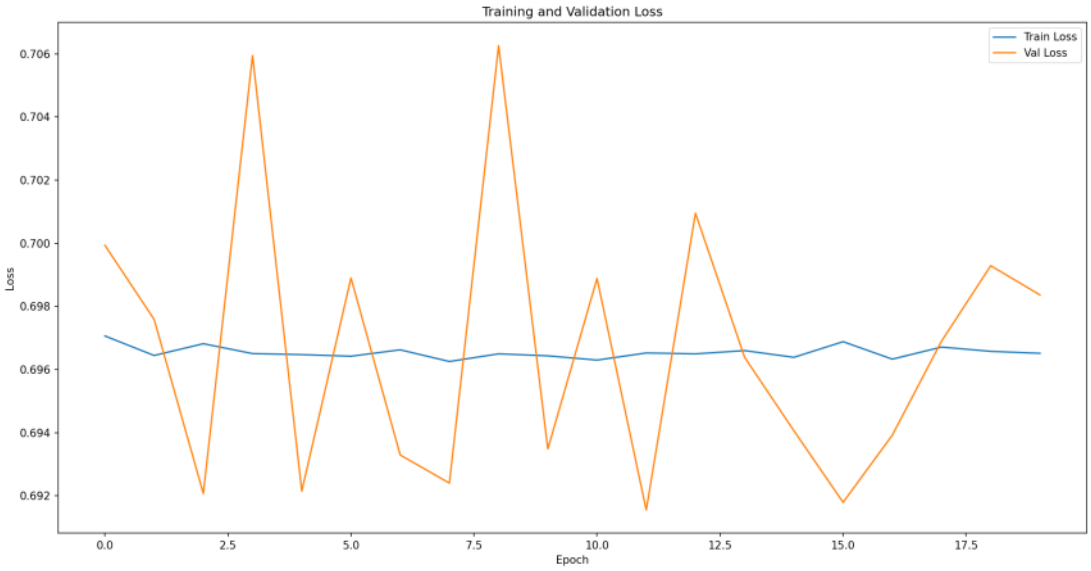


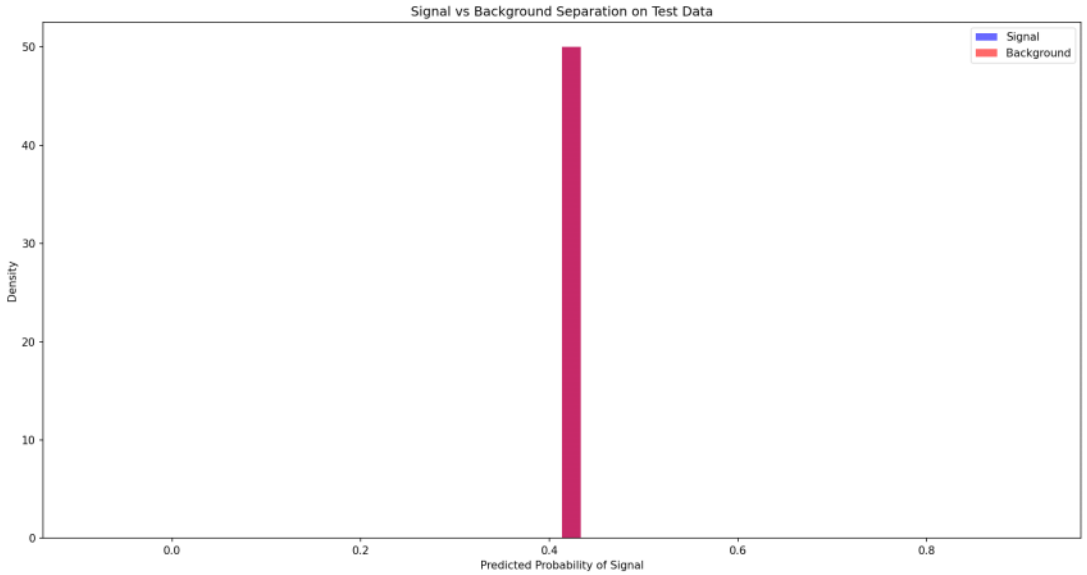
Below is the model using ReLUs again but also with the Adam model with lr = 0.05 and weight_decay = 0.001. Note now that it really struggles to confirm that something definitely is a signal and the valuation loss curve is all over the place (only for 20 epochs as my Laptop bluescreened last time I tried 100 lol)



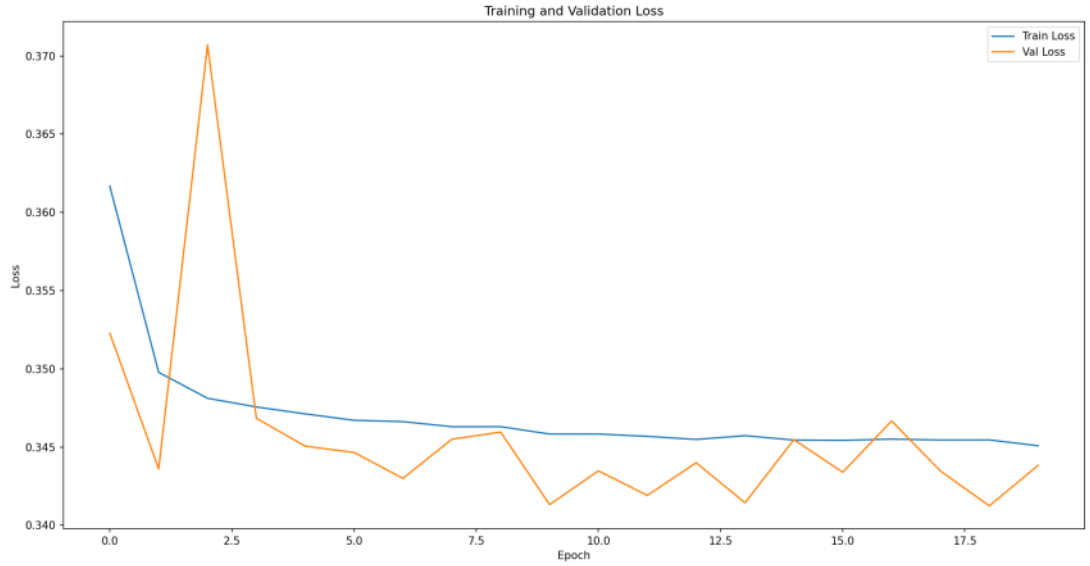


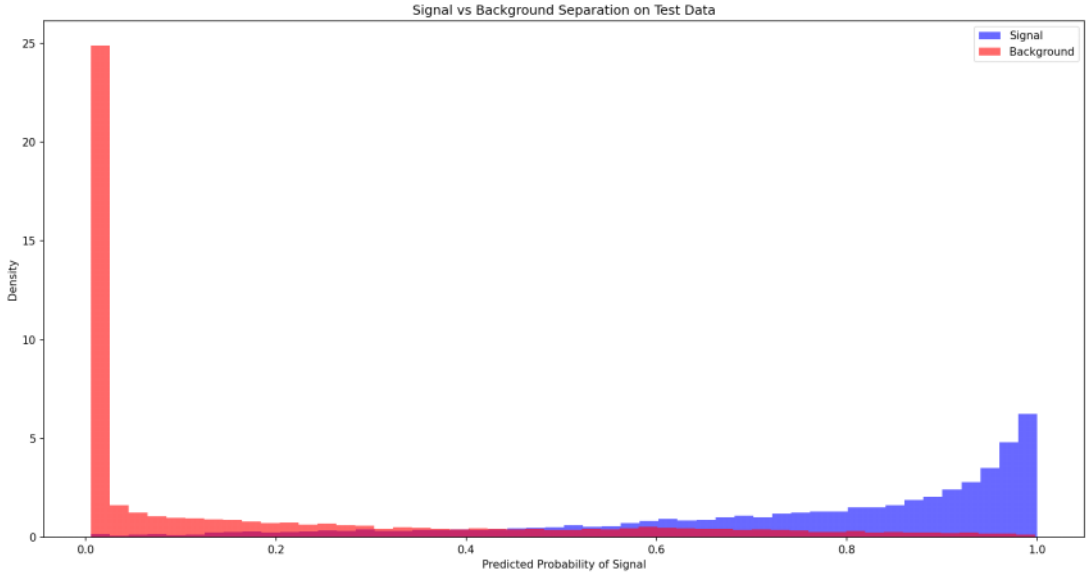
ReLUs with Adam model with lr = 0.25 and no weight_decay specified:



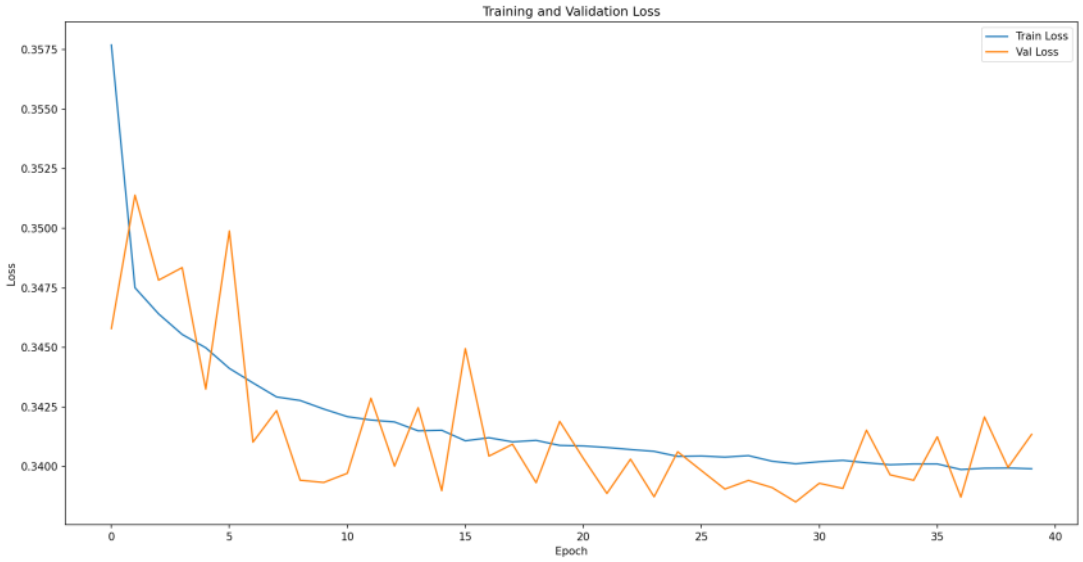


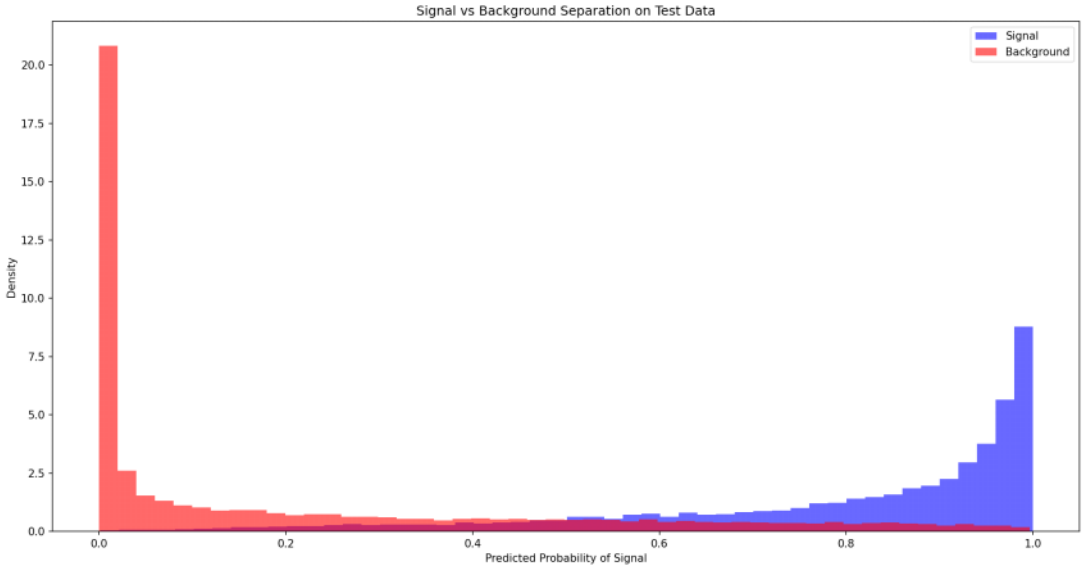
ReLUs with SGD with lr = 0.20:



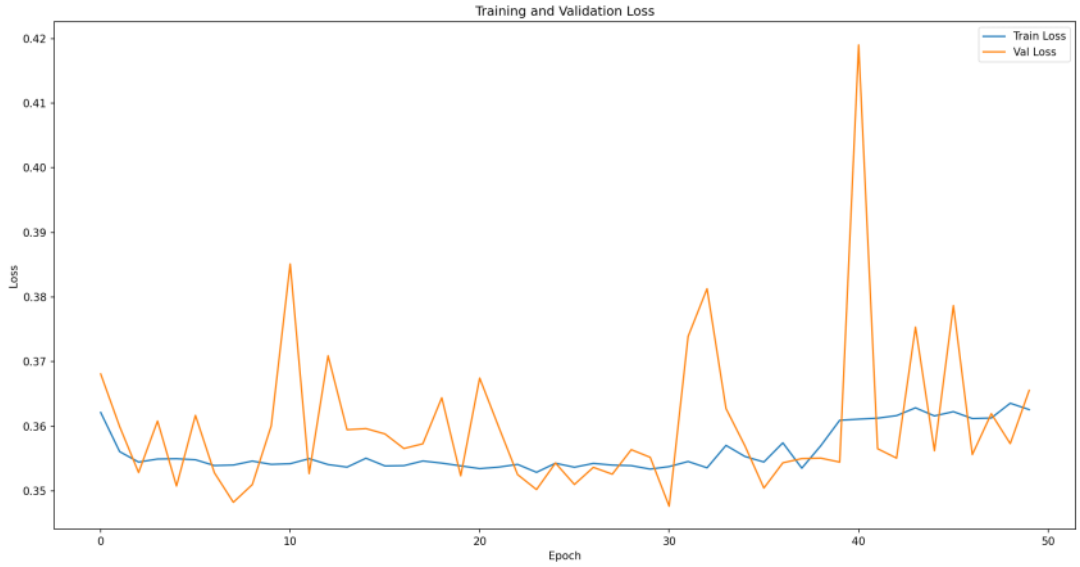


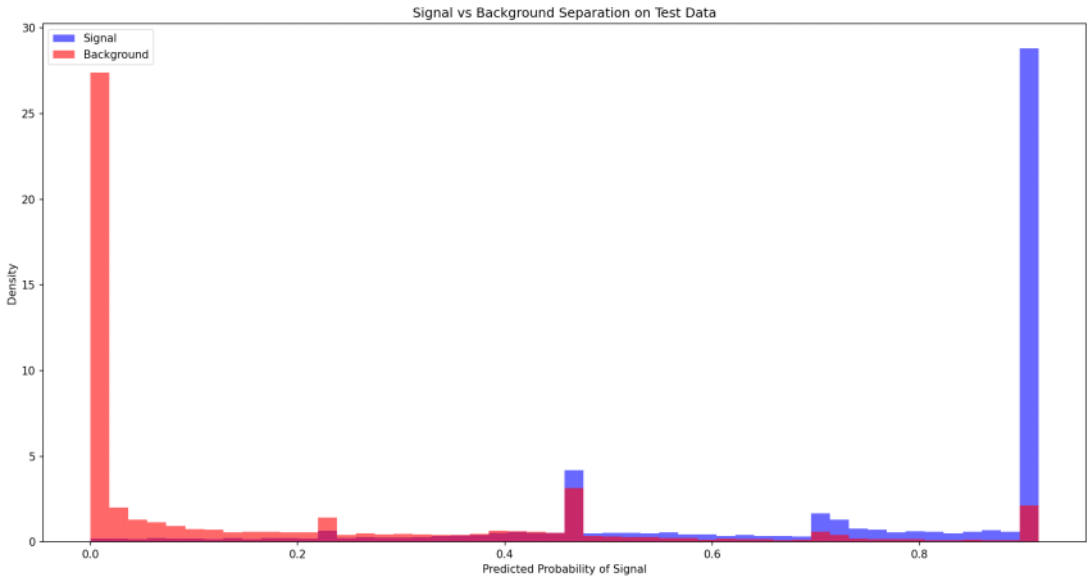
ReLUs with SGD with lr = 0.10:





ReLUs with SGD with lr = 0.10 and momentum = 0.90:





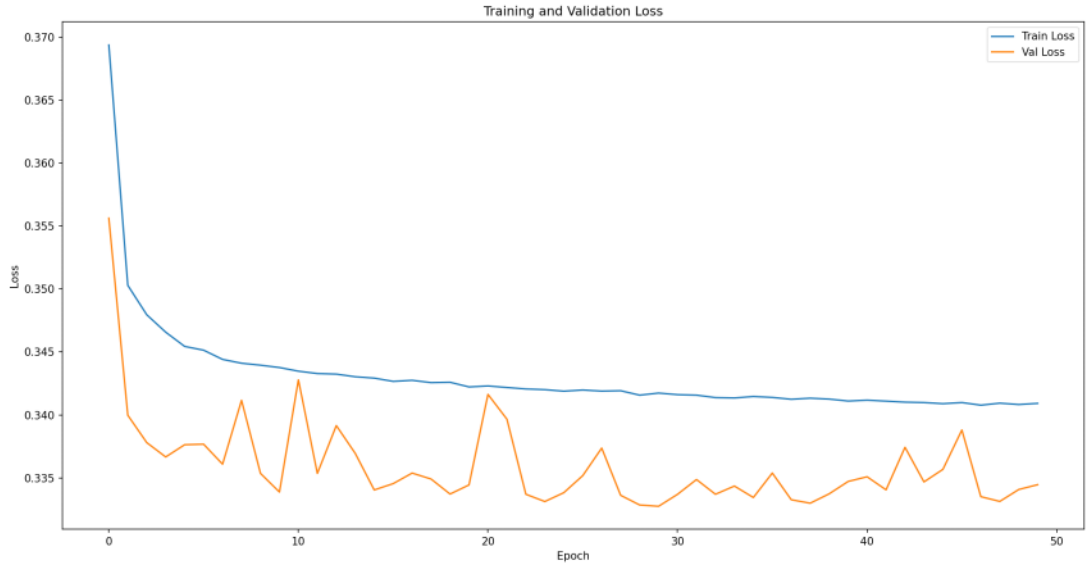
09/10/2025

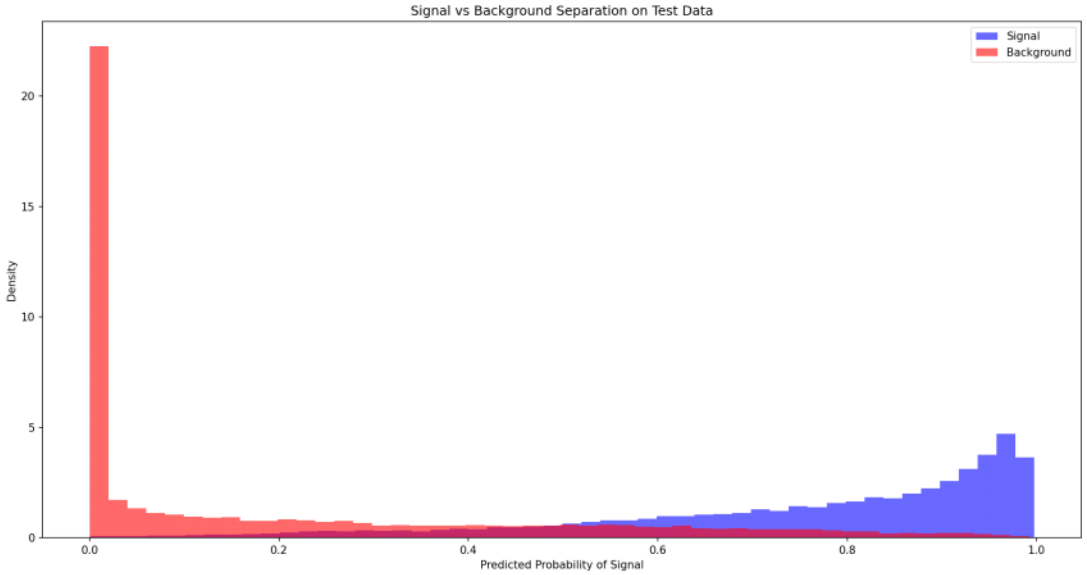
11:40 - Returning to testing

Changed the sequential loop:

```
model = nn.Sequential(
    nn.Linear(len(input_features), 100),
    nn.Sigmoid(),
    nn.Linear(100, 50),
    nn.Sigmoid(),
    nn.Linear(50, 1),
    nn.Sigmoid())
```

This was then computed across 50 epochs with SGD with lr = 0.10 and momentum = 0.90:





Evaluation code:

```
model.eval()
with torch.no_grad():
    y_pred_prob = model(X_test)
signal_probs = y_pred_prob[y_test[:,0] == 1].numpy()
background_probs = y_pred_prob[y_test[:,0] == 0].numpy()
```

Plotting the bins:

```
plt.hist(signal_probs, bins=50, alpha=0.6, label='Signal', color='blue',
density=True)
```

```
plt.hist(background_probs, bins=50, alpha=0.6, label='Background',
color='red', density=True)
plt.xlabel('Predicted Probability of Signal')
plt.ylabel('Density')
plt.title('Signal vs Background Separation on Test Data')
plt.legend()
plt.show()
```

Plotting the losses:

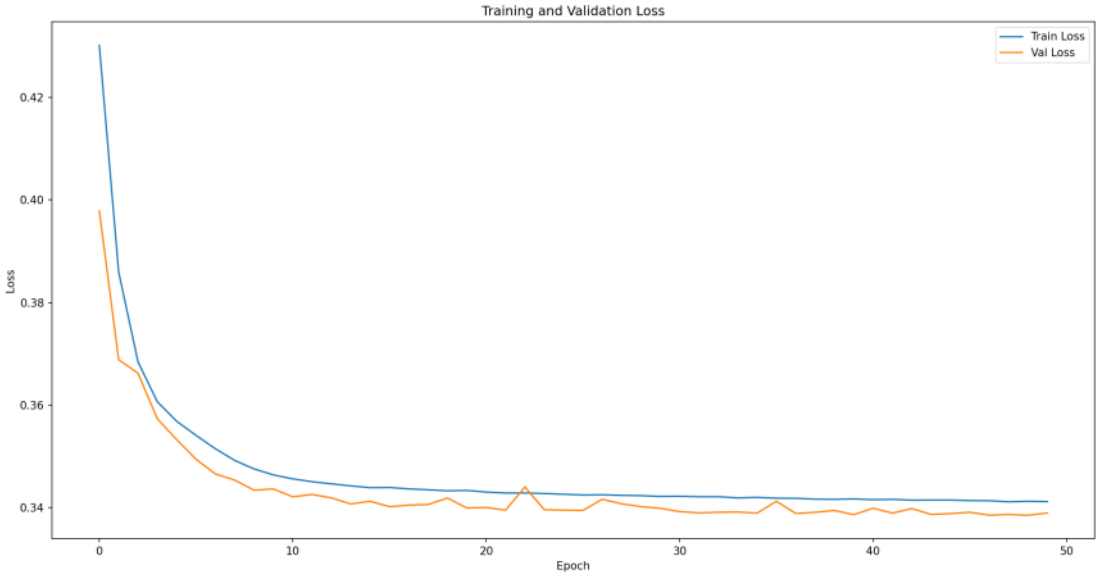
```
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Val Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()
```

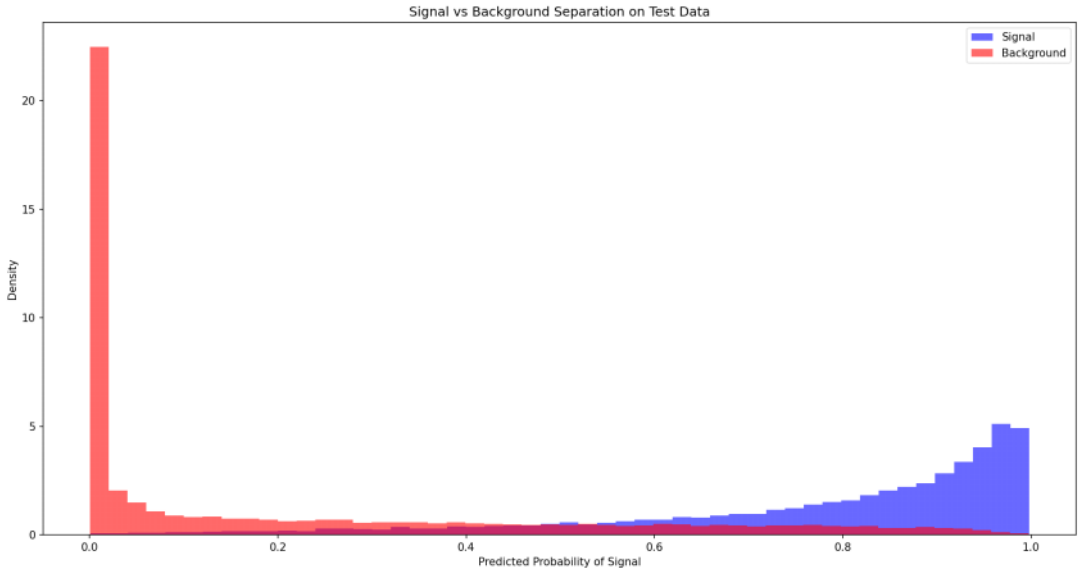
Resetting the weights at the end:

```
def reset_weights(m):
    if hasattr(m, 'reset_parameters'):
        m.reset_parameters()
model.apply(reset_weights)
```

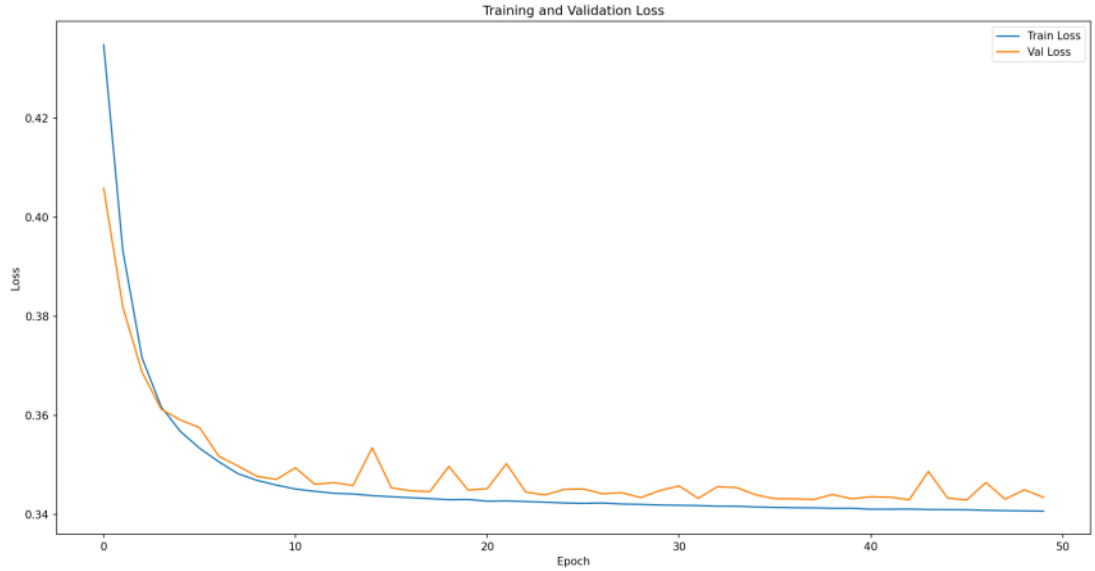
These are all reasonably self-explanatory.

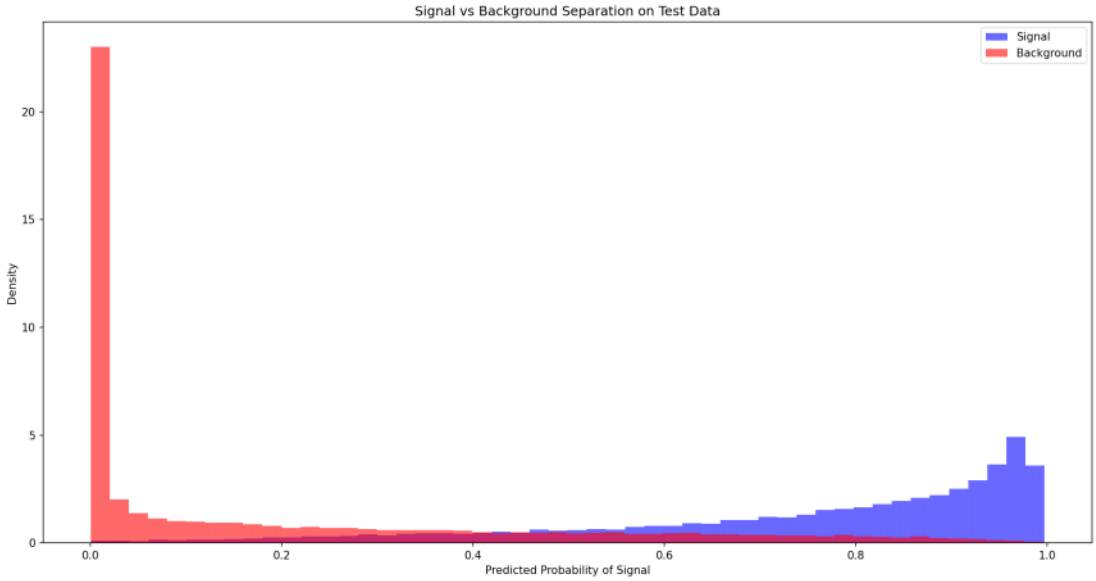
One more run with SGD with lr = 0.005 and momentum = 0.900





12:00 - Quantifying the model






```
final_prediction_score = y_pred_prob.numpy()
final_prediction = np.round(final_prediction_score)
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1
_score, roc_auc_score
```

```
accuracy = accuracy_score(y_test, final_prediction)
precision = precision_score(y_test, final_prediction, average='weighted')
recall = recall_score(y_test, final_prediction, average='weighted')
f1 = f1_score(y_test, final_prediction, average='weighted')
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

Accuracy: 0.8395218449169576
Precision: 0.8399250841980481
Recall: 0.8395218449169576
F1 Score: 0.8396066157816755

To go further into all of this stuff:

A True Positive is when the data should be positive and the model estimates it as positive.
A True Negative is when the data should be negative and the model estimates it as negative.
A False Positive is when the data should be negative but the model estimates it as positive.
A False Negative is when the data should be positive but the model estimates it as negative.

Accuracy is how many predictions are correct overall:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision is of the samples that the model predicted as positive, how many are actually positive:

$$Precision = \frac{TP}{TP + FP}$$

Recall (Sensitivity or Efficiency) is out of all the actual positive samples, how many did the model correctly identify:

$$Recall = \frac{TP}{TP + FN}$$

F1 score is the balance between precision and recall, if F1 = 1 then it's got perfect precision and recall and if F1 = 0 then it's completely wrong:

$$F1\ score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

12:51 - Receiver Operating Characteristic curve

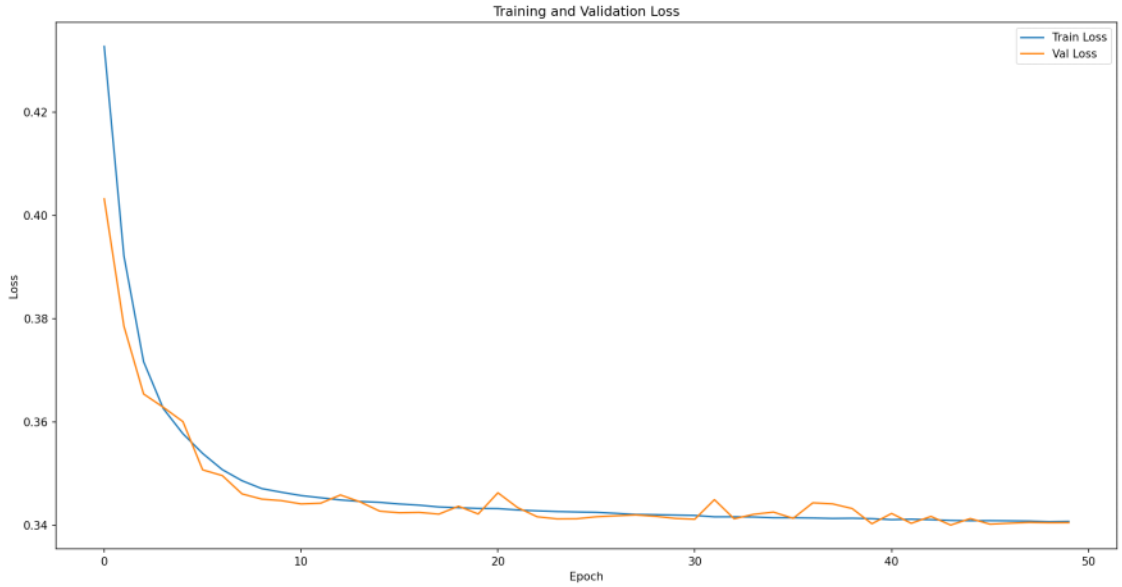
The ROC curve plots true positive rate against false positive rate:

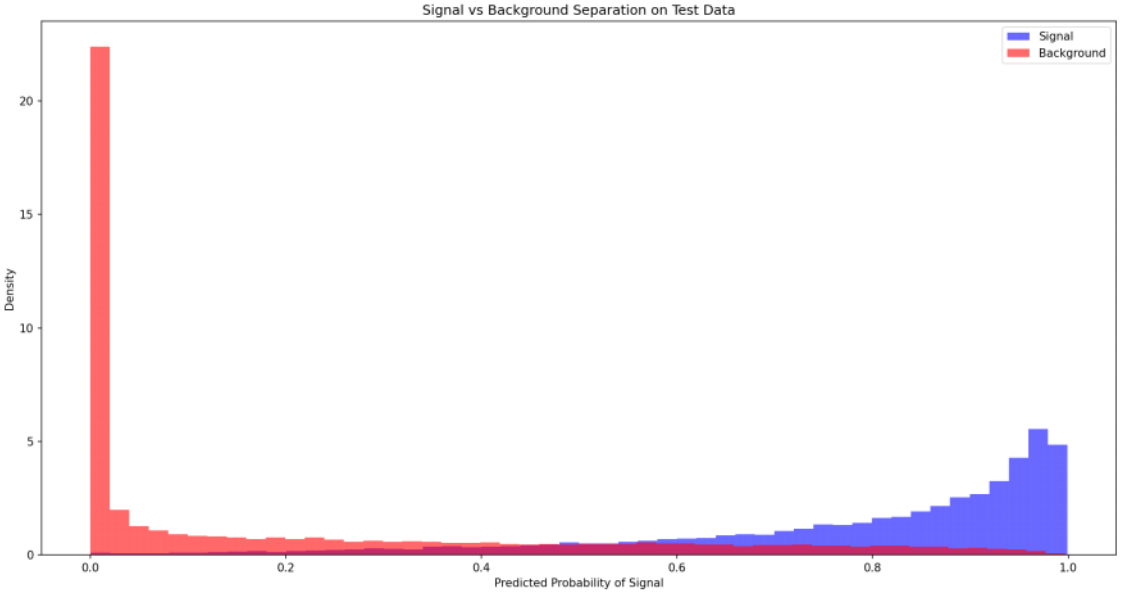
$$TPR = \frac{TP}{TP + FN}$$

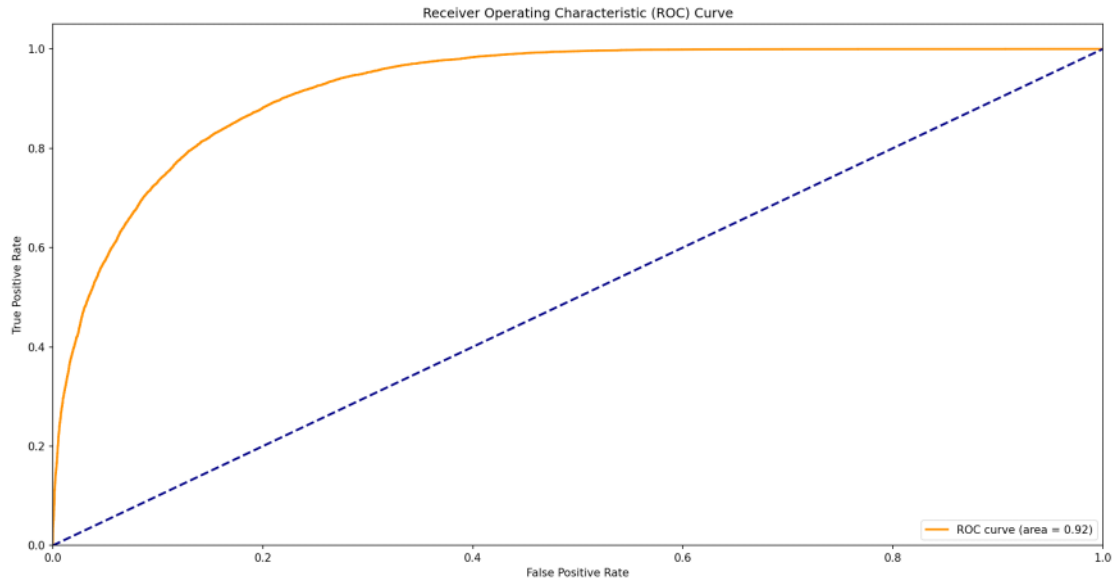
$$FPR = \frac{FP}{FP + TN}$$

TPR is on the y-axis, FPR is on the x-axis. The closer the ROC curve is to the top-left corner, the better the model.

Area under the curve is the way that







[2] Reference: https://arxiv.org/pdf/1412.6980