Daniel Davis & Samuel Gabe

Project Plan for Project 36
Classifying Cosmological Data with Machine Learning

In this project we plan to classify data from the Quijote simulations and find out how much dark matter is in the simulation.
The Quijote simulations are an online set of 44,100 N-body simulations spanning more than 7,000 cosmological models.[1] An N-body simulation simulates how a large number of particles behave under forces such as gravity.
We will use machine learning in this project, as we will be dealing with datasets on the order of hundreds of terabytes, which is too time consuming and too vast to justify going through by hand. The Quijote simulations comprise over a petabyte of data.[2] We will be using Python, and either the Tensorflow or PyTorch deep learning libraries, rather than make the code completely from scratch.
In the simulation there are many values which need to be tweaked until the simulation matches the real-world observations of the universe. In one method of creating the simulation, a random setting for these values can be tested to see how closely it resembles the real-world results. The difference between the simulation results and real-world results is calculated and a total error is found by taking the sum of the squares of the errors. The computer then takes a small deviation of each of these values and again calculates the error. The gradient can be calculated, and this can be extrapolated until a minimum in the total error is found. However, this method has issues, because a local minimum for the error could be found and mistaken to be the global minimum.
The first four weeks will be spent learning about machine learning from notes by our supervisor, Adam Moss, to give us an understanding of how deep learning works.
After this we will use what we have learned to create a program which uses the simulation data to find out how much dark matter is in the simulation.

References

[1] Villaescusa-Navarro, F., Hahn, C., Massara, E., Banerjee, A., Delgado, A., Ramanah, D., Charnock, T., Giusarma, E., Li, Y., Allys, E., & et al. (2020). The Quijote Simulations. The Astrophysical Journal Supplement Series, 250(1), 2.
[2] Villaescusa-Navarro, F., 2020. Features - Quijote Simulations 0.1 Documentation. [Online]
Available at: https://quijote-simulations.readthedocs.io/en/latest/features.html [Accessed 5 October 2021].

Week 3

In our first supervisor meeting on 05/10/21, we discussed the project's content and aim. We were also given a set of notes on Machine Learning in Science by our supervisor and given the task to read up to and including section 1.2.1 so that we could learn the basics of a neural network for the project.

Section 1 of the notes is an introduction to deep learning, with subsection 1.1 detailing the basics of a neural network; it covers: the notation used in machine learning, what a loss function is (the function we will minimise in order to reduce the prediction error of the neural network), what a neuron is, and what activation functions are (the functions we use in to get the predictions from the input data). Subsection 1.2 gives an overview of the four basic types of a neural network, with us only reading about the linear model this week.

We will give an overview of the notes read below:

1 Introduction to Deep Learning

Deep learning is a class of machine learning which aims to teach a computer an abstract *representation* of data.

How the algorithm works is dependent on the representation of data(*features*) it learns from. Knowing which features to use is the difficult part.

Supervised Learning is when an algorithm is given a set of labelled training data, and then it will update itself to minimise error. This should hopefully be able to generalise for new, unknown datasets.

Neural Network Basics

Notation

A single data point, i, is given by (x^i, y^i) where:

- x^i is the input vector of P dimensions
- y^i is the target vector of K dimensions

The components of x^i are therefore denoted as x_p^i.

A neural network outputs a prediction vector (of dimension K) $\hat{y}^i\ (\theta)$, where θ are the parameters of the network (the weights and biases)

For a batch of N examples in our dataset we can form the input (*design*) matrix X of dimensions NxP This is the *feature by column* convention (which Tensorflow uses).

The output matrix can similarly be defined as Y of dimensions NxK, along with the prediction matrix $\hat{Y}$.

It's convention to have a first 'artificial' column of $X$of all ones, which we label as $P = 0$.

Loss Function

The goal of supervised learning is to minimise the prediction error. This is achieved by minimising the *loss function*:

$$J(\theta) = \frac{1}{N} \Sigma\, l\left(\hat{y}^i, y^i\right)$$

The per-example loss chosen is dependent on the task. (e.g. mean square error for regression).

The optimal set of parameters is given by the loss function:

$$\theta^* = \; argmin[J(\theta)]$$

This is the set of parameters that give the minimum loss function, and thus the least prediction error.

Types of Loss Function
The loss function depends on what data is being used.

Continuous data: Mean Square Error or Mean Absolute Error

Binary data: Binary Cross Entropy

Categorical Data: Categorical Cross Entropy

Neurons

The basic building block of a neural network is a *neuron*. It takes an input vector x and outputs a scalar $a(x)$.

$z = x^T\,\omega + b \qquad a = \Phi(z)$

Φ is the *activation function*, and a is the *activation*.

z can be rewritten so that the bias of the neuron, b, is contained with the weights term ω.
$z = \mathbf{x}^T \boldsymbol{w}$ *where* $\boldsymbol{x} = (1, x)^T$ *and* $\boldsymbol{w} \; = (b, \omega)^T$

A *multi-layer perceptron* is comprised of many such neurons arranged in layers, where the output of one is the input of the next layer.

Activation Functions

The activation function determines how a neuron determines makes a prediction from the input data.

Choices for the activation function include:

- The sigmoid function: $\Phi(z) = \frac{1}{(1+e^{-z})}$
- The hyperbolic tangent: $\Phi(z) = \tanh(z)$
- Rectified linear units (ReLUs): $\Phi(z) = \max(0, z)$
- Leaky rectified linear units (leaky ReLUs): $\Phi(z) = \; 0.1z \; if \; z \leq 0$
  $z \;\; if \; z \geq 0$
- Exponential linear units (ELUs): $\Phi(z) = e^z - 1 \; if \; z \leq 0$
  $z \; if \; z \geq 0$

Neural networks are usually trained by *back propagation,* which requires the derivative of the activation function, and so the last three functions are preferred. They all have easily found gradients and, unlike the first two, the derivatives do not vanish as the input becomes too big (saturated activation function).

Basic Types of Model

Linear Model

The simplest type of neural network maps an input vector to a scalar output, and is modelled by:

$$\hat{y} = \boldsymbol{x}^T\mathbf{w}$$

The linear model has no hidden layers (neuron layers in between the input layer and the output layer).

The linear model has a linear activation function $\Phi(z) = z$.

It is more efficient to pass through an entire *batch* of inputs:

$$\hat{\mathbf{Y}}=\mathbf{Xw}$$

The optimal weights are found by calculating the gradient of the chosen loss function and solving the *normal equations:*

$$\nabla_w J(\mathbf{w}) = 0$$

The linear model can successfully learn the AND function, but fails to learn the non-linear XOR (exclusive or) function.

How it works

- Set weights to 0
- Iteratively update weights using the learning rate α, which is set based on the circumstances
- Keep going until condition has been met, e.g. number of iterations or not much difference between this iteration and the last

If sample data is not diverse enough, the machine may arrive at the wrong answer.

Week 4

This week we have been given the task to read up to section 1.3 of the Deep Learning notes. We will read the notes and write down some things we have learned.

Perceptron

The perceptron model is given by:

$\hat{y} = \Theta(\mathbf{x}^T \mathbf{w})$

where the activation function is the Heaviside function: $\Phi(z) = \Theta(z) = 0 \; if \; z \leq 0$
$1 \; otherwise$

This is an example of a *binary classifier* (is capable of learning whether an input example belongs to a specific binary class (0 or 1)).

The step function is non-linear so we cannot solve for the optimal weights the same way as the linear model.
Instead we use the perceptron learning algorithm; an iterative process that updates the weights depending on the size of the prediction error:

1. Initialise the weights to 0 or a small (close to 0) random number.
2. Iteratively update the weights by:
   $\mathbf{w} \to \mathbf{w} - \alpha \mathbf{X}^T (\hat{\mathbf{Y}} - \mathbf{Y})$
   Where α is the *learning rate*
3. Repeat step 2 until a stopping condition is met.
   e.g. a maximum number of iterations, or the loss doesn't improve much between iterations.

The perceptron is able to learn the AND function using a correct *decision boundary*, but it fails with learning the XOR function.
This is because the perceptron is a *linear classifier* and so is not able to classify examples correctly if the training data is not *linearly separable*.

Logistic Regression

The logistic model is given by:

$\hat{y} = \sigma(\mathbf{x}^T \mathbf{w})$

Where the activation function is the sigmoid function: $\Phi(z) = \sigma(z) = \frac{1}{1+e^{-z}}$

Because the activation function takes values between 0 and 1, this model is commonly used when estimating when the probability that an example belongs to a binary class.

$\hat{y} = P(y = 1|\mathbf{x})$

Because the activation function is differentiable we can use *gradient descent* to find the optimal parameters.
If the chosen loss function is *convex* this is guaranteed to find a set of optimal weights.

Sigmoid function
The sigmoid function is $S(x) = \frac{1}{1+e^{-x}}$

From the notes: "The activation function is differentiable so we use gradient descent. If the loss function is complex it is guaranteed to find the optimal solution."

A convex function is one where the line segment between any two points does not go underneath the graph. An example of this is the function $f(x) = x^2$.

Gradient descent
Because the activation function is differentiable we can use *gradient descent* to find the optimal parameters.
If the chosen loss function is *convex* this is guaranteed to find a set of optimal weights.

Gradient descent iteratively updates the weights by:

$\mathbf{w} \rightarrow \mathbf{w} - \alpha \nabla_w J(\mathbf{w})$

Where α is again the learning rate. It is positive and gives the size of the updates.
We set α to a small constant (e.g $\alpha = 0.1$)

Then gradient descent is repeated for a number of *epochs*(a complete pass over the training data), or until a different stopping condition is met (e.g. the loss not significantly improving between iterations).

This gradient descent is very similar to the perceptron learning algorithm.

Logistic regression can correctly learn the AND function, but again fails at the XOR function as the training dataset is still not linearly separable.

Summary of gradient descent:

- We choose some values for the weights and calculate the predicted result ŷ with these values.
- We change the values by a small amount determined by the learning rate, α.
- We see if the predicted values, ŷ, move closer to the observed values, y.
- We keep iterating until the minimum difference between the predicted values and the observed values is reached.

Learning rate
The learning rate α can be chosen based on circumstances. For the example in the notes, the learning rate is set to a small value. To optimise the speed of the gradient descent procedure, we must choose the right value of α.

- If the learning rate is set too low, each iteration will only move a tiny bit closer to the desired value of ŷ. it will take many iterations to complete the procedure
- If the learning rate is set too high, each iteration could go too far and 'miss' the desired value, so each iteration will oscillate about the desired value.

How the matrix multiplying works

- Set the x matrix with the two values and 1 in the first column
- Set the y matrix with the results
- Set the values for w. Random 3D vector
- Each iteration:
  - $\hat{y} = \sigma(\mathbf{x}^{\mathbf{T}} \cdot \mathbf{w})$
  - $\mathbf{w} \rightarrow \mathbf{w} - \alpha \cdot x^T \cdot (\hat{y} - y)$
- This gives us our values for w, the 3D vector.
- The gradient of the BCE loss is $\frac{1}{N} X^T (\hat{Y} - Y)$

Multi-layer perceptron

New system: input layer with two units, hidden layer with two hidden units and a single output unit.

The weights **W** is now a matrix rather than a vector.

- Amount of rows equal to number of inputs
- Amount of columns equal to number of outputs.

ReLU activation function is used. (y = 0 if x < 0, y = x for x ≥ 0)

$$\hat{y} = \boldsymbol{w}^{(T)}\boldsymbol{h} + b, \ \ \mathrm{h} = ReLU\left(\boldsymbol{W}^{(T)}\boldsymbol{x} + \boldsymbol{c}\right)$$

x and **h** are column vectors by convention, and **W** is the weights matrix (number of rows is equal to the input dimension, and the number of columns is equal to the number of hidden units in the hidden layer given by **h**), which connects the input and hidden layers. The bias of the hidden layer, c, is also a vector.

If we didn't apply the ReLU activation function to the hidden layers, the neural network would just be a linear function of the inputs.

Again, we can absorb the biases into the weights and pass through an entire batch of inputs, to have a much more compact model:

$\hat{\mathbf{Y}} = (ReLU(\mathbf{XW}))\mathbf{w}$

N.B.: A common mistake in deep learning is not getting *tensor* (vectors and matrices) dimensions right.

To find the set of optimal parameters for a training set back propagation (efficiently propagating gradients back through the network) is needed as the neural network is *deep*.

The MLP can reproduce the XOR function successfully because, although the training dataset is not originally linearly separable, the hidden layer makes it so (the hidden layer is there to make a better representation of the input data).

Architecture

Architecture is an important consideration for designing neural networks.

For the MLP this is determined by the number of hidden layers, and the number of hidden units per layer.

Assuming that, excluding the input layer, there are L layers in our network with $l = 1,2, \dots, L$ indexing the layers.

The lth layer is modelled (for a single training example) by: (each layer acts like a single neuron)

$\mathbf{z}^l = \mathbf{W}^{(l)T}\mathbf{a}^{(l-1)} + \mathbf{b}^l, \ \ \mathbf{a}^l = \Phi^l(\mathbf{z}^l)$

Except for the first hidden layer ($l = 1$): $\mathbf{a}^0 = \mathbf{x}$, the input vector.

The weight ${W_{ij}}^l$is the weight for the connection from the ith neuron in layer$(l-1)$, to the jth neuron in layer l.

The loss, J, depends on the activation of the output layer, $\mathbf{a}^L$, which indirectly depends on all the activations of the previous neuron layers.

Deeper networks with fewer units per layer often generalise better to unseen test data, but they can be harder to train and optimise.

Week 5

In our meeting on 21/10/21 we discussed the previous week's notes, and were given a brief overview of section 1.3 (Back Propagation); this explains the mathematics behind back-propagation, which is not vital to our project. Essentially it involves the chain rule for partial differentiation, in order to calculate the gradient of the loss function for a MLP, with respect to all the neurons' parameters.
Then we were given the task to gain an understanding of the basics of Tensorflow, the deep learning library we will be using. This entails doing the beginner tutorials on the Tensorflow website. We will write down our progress through the tutorial.

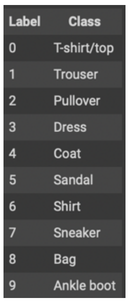
The first tutorial titled 'Beginner Quickstart' loads the MNIST dataset, a database of handwritten digits, and converts the integer data to float variables. Then a machine learning model (a sequential model, using the Keras API, using layers with only one input tensor and only one output tensor) is built that returns a vector of log-odd (the logarithm of the odds of an event) score for the each example in the MNIST dataset. These scores are converted into probabilities using the tf.nn.softmax function. A loss function is defined for training the model, and then the model is compiled before training begins, setting the accuracy of the model to be evaluated.
The model.fit method is used to train the model (it adjusts parameters to minimise the loss), and once the specified number of epochs has passed, the model.evaluate method can check the model's accuracy.
The model can be altered to output probabilities.

The second tutorial shows how to categorise clothes using the tensorflow library:

- Load dataset: `fashion_mnist = tf.keras.datasets.fashion_mnist`
- Get data from dataset:

```
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
```

- Define each class:



Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

- Each image is represented by 28x28 pixels. There are 60,000 images and 60,000 labels. Each label is an integer between 0 and 9.
- There is also a test set with 10,000 images and 10,000 labels.

- We must pre-process the data before training the network. The images are in black and white, so each pixel has a value from 0 to 255. We need these as numbers from 0 to 1, so we divide by 255.0
- The training set has labels matched to the images. Tensorflow will try to make a model which accurately puts a label to each image.
- We define the layers of the neural network:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

Here, we have 3 layers: Flatten converts the 28x28 array to a 1D array of size 784. Then there are two Dense layers. The first one has 128 nodes/neurons and the second has 10. The one with 10 will show which label belongs to the image.

The next step is to use the "compile" function which takes 3 arguments: optimizer, loss function and metrics. The loss function measures the accuracy of the model (which must be minimized), the optimizer determines how the data is updated based on the data it sees, and metrics are used to monitor the training and testing steps. The model here uses accuracy which measures which proportion of the images have the correct label assigned to them.
(note: I don't understand what the optimiser is so I will ask at the next project meeting)

```
model.compile(optimizer='adam',
      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
      metrics=['accuracy'])
```

To start training we call the model.fit() method, which fits the model to our data (the 'learning' part of the process). This takes the most time to run. We use the train images and train labels to train the model, and specify the number of epochs we want to do.

```
model.fit(train_images, train_labels, epochs=10)
```

We can now find out how accurate our model is.

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

The accuracy is 0.8187 which is less than the accuracy on the training dataset (0.91). This is because the model has overfitted to the training data. Overfitting is when a model performs worse on a new dataset than on the training dataset.

A trained model can be used to make predictions for the labels of an image. The model outputs logits which can be converted to probabilities for ease of understanding. The model gives a prediction array, an array of 10 probabilities corresponding to each label, which represent how confident the model is in assigning the image to each label. The highest confidence value shows what label it thinks an image should have, though this can be wrong.

The predictions of an image can be plotted against each label, to see how well the model performs.

However, the model can be wrong even if it's very sure how to categorise a piece of data. For example, in the tutorial, a sneaker was categorized as a sandal with 99% certainty.

CSV data
This tutorial covers:

- Loading the data off-disk
- Pre-processing the data into a form suitable for training

Small CSV dataset: load as a pandas dataframe or a numpy array.

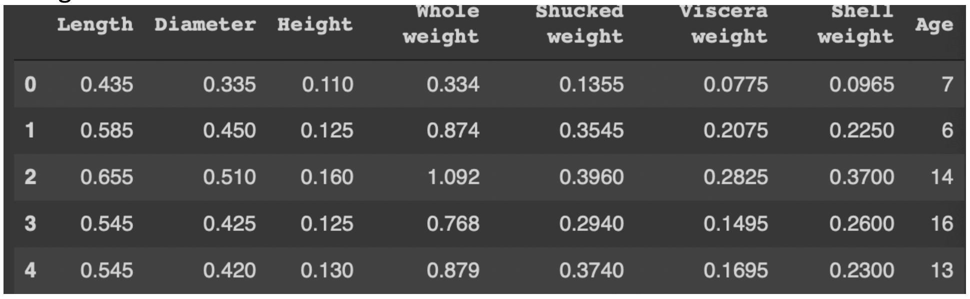
The tutorial loads the Abalone dataset, with the intention of being able to the age of datapoints from other measurements (features). It separates the features and labels (age) for training. Then it puts all the features into one NumPy array, treating all the features as equal.

A keras.Sequential model is built and compiled, in order to learn how to predict the age of the abalone datapoints. The neural network is then trained by use of the model.fit method, minimising and outputting the loss for a number of epochs.

Here the data is loaded with pandas:

```
abalone_train = pd.read_csv(

"https://storage.googleapis.com/download.tensorflow.org/data/abalone_train.csv",
    names=["Length", "Diameter", "Height", "Whole weight", "Shucked weight",
           "Viscera weight", "Shell weight", "Age"])
abalone_train.head()
```

This gives a table:



	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Age
0	0.435	0.335	0.110	0.334	0.1355	0.0775	0.0965	7
1	0.585	0.450	0.125	0.874	0.3545	0.2075	0.2250	6
2	0.655	0.510	0.160	1.092	0.3960	0.2825	0.3700	14
3	0.545	0.425	0.125	0.768	0.2940	0.1495	0.2600	16
4	0.545	0.420	0.130	0.879	0.3740	0.1695	0.2300	13

There are many parameters and we will try to build a model to predict the age from these.

We separate the features and labels then pack them into a single numpy array.

```
abalone_features = abalone_train.copy()
abalone_labels = abalone_features.pop('Age')
abalone_features = np.array(abalone_features)
```

Then we make a model to predict the age.

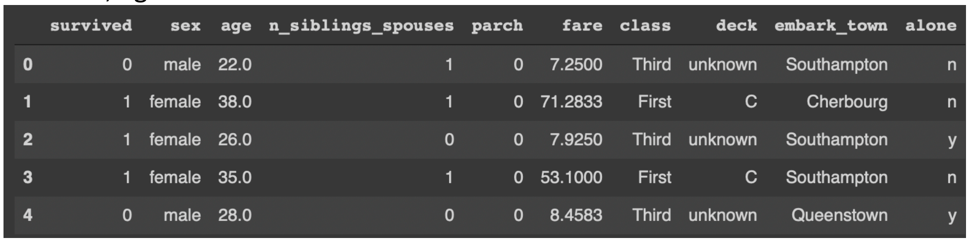
```
abalone_model = tf.keras.Sequential([
  layers.Dense(64),
  layers.Dense(1)
])
abalone_model.compile(loss = tf.losses.MeanSquaredError(),
                      optimizer = tf.optimizers.Adam())
```

Now we fit the model to the data.

```
abalone_model.fit(abalone_features, abalone_labels, epochs=10)
```

Normalisation
It's good practice to normalise the inputs.

Titanic dataset
We have a dataset of people who survived or didn't survive and some details about them, such as their sex, age and class.



	survived	sex	age	n_siblings_spouses	parch	fare	class	deck	embark_town	alone
0	0	male	22.0	1	0	7.2500	Third	unknown	Southampton	n
1	1	female	38.0	1	0	71.2833	First	C	Cherbourg	n
2	1	female	26.0	0	0	7.9250	Third	unknown	Southampton	y
3	1	female	35.0	1	0	53.1000	First	C	Southampton	n
4	0	male	28.0	0	0	8.4583	Third	unknown	Queenstown	y

```
titanic_features = titanic.copy()
titanic_labels = titanic_features.pop('survived')
```

The Titanic dataset can be used to teach a neural network to predict who survived the Sinking of the Titanic. The dataset is loaded as a Pandas Dataframe, but is not immediately usable due to the mixed datatypes included. The data needs to be preprocessed through the model, using symbolic keras.Input objects, which match the names and datatypes of the training dataset columns.

The numeric data is first concatenated together and normalised. Once processed the data is stored to be concatenated later.

For the string data use the tf.keras.layers.StringLookup function to map the strings to integer indices. Then the tf.keras.layers.CategoryEncoding function is used to convert the indices to float32 data, to be used by the model.

The processed input data is then all concatenated together, and a model is built that handles the preprocessing. Convert the input data to a dictionary of tensors.
Then the neural network can be built and compiled that uses the preprocessed data dictionary and labels.

The advantage of having the preprocessing being done by the model is that the model will automatically process data to be used without having to do it manually beforehand.

If more control over the input data is needed the tf.data command should be used.

The Dataset.from_tensor_slices constructor can be used to slice up dictionaries in TensorFlow. Dataset.cache or data.expirmental.snapshot can be used to speed up models' runtimes. However, cache files are only usable by the model that made them, whereas snapshot files can be used by other models. Snapshot files are only temporary storage of data, it is not guaranteed to be usable in the long term.

tf.data is particularly useful when dealing with multiple CSV files, making up a dataset. Multiple files can be read in parallel by the model.

There are two APIs that are used when data doesn't fit the basic patterns. tf.io.decode_csv is a function that can convert string into a list of columns. tf.data.experimental.CsvDataset can provide a CSV dataset interface without any extras included in the make_csv_dataset function.

Week 6

This week we were given the task to read chapter 1.7 of the notes on regularisation.

Regularisation
The real goal of machine learning is not just to perform well on seen training data, but to *generalise* well for unseen test (or validation) data. We need to prevent overfitting.
Regularisation is any strategy that is designed to reduce generalisation error, without affecting the training error.
An effective regulariser successfully balances the *bias-variance trade off*. It decreases the model variance responsible for *overfitting,* while not increasing bias too much.

Early stopping
We can split datasets into different categories: training, test and holdout.

Training is self-explanatory: we use this data to train the machine. The expected output is already provided without needing a machine to work it out.

Test is the data we use to see how the machine performs on unseen data. Can be used to test the model and also to find the optimal values of various parameters in the neural network, e.g. number of layers and hidden units.

Holdout is data which is used to assess the final performance of the tuned model.

During training we often observe that training loss continually decreases, but test loss begins to increase. (Overfitting)

Early stopping stops the training process after the test loss has not decreased over the previously recorded minimum test loss for a specific number of epochs.
Each time we obtain an improved test loss, the model parameters are stored, and when training terminates these are used as the best model.

L_1/L_2 Regularisation
These two forms of regularisation are types of *parameter-norm penalties*, which are implemented by adding a term to the cost function.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda\Omega(\mathbf{w})$$

λ is the hyper-parameter that determines contribution of the penalty term $\Omega(\mathbf{w})$. In deep learning the norm penalty is only a function of the weights at each layer, since biases do not contribute as much to the model's variance.

L2 norm: $||\mathbf{x}||_2 = \sqrt{\sum_i |x_i|^2}$, $\Omega(\mathbf{w}) = ||\mathbf{w}||_2^2$

where omega is the penalty term.

e.g. for the linear model in section 1.2.1, the solution to the normal equations (the optimal weights) is modified to:

$$\mathbf{w}^* = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{Y}$$

Where **I** is the identity matrix.
In this case regularisation has the effect of increasing the variance of the input, which makes it no longer depend on the features as much.

In general, the effect of regularisation, near local minima of the cost function, is to rescale the optimised parameters **w*** along the axes defined by the eigenvectors of the Hessian **H**. For directions with large eigenvalues that significantly affect the cost, the components of **w*** will be unchanged; for directions with small eigenvalues that do not significantly affect the cost, the components of **w*** will vanish.

The L_1 norm of a vector **x** is given by $||\mathbf{x}||_1 = \sum_i |x_i|$, and adds a penalty term (LASSO regression) of the form:

$$\Omega(\mathbf{w}) = ||\mathbf{w}||_1$$

In general L_1 regularisation tends to give *sparse* solutions (more components of **w*** will be zero).

Dropout
During training every neuron (except the output neurons) have a probability p of having its input and output weights set to zero.
During testing no neurons are dropped out, so weights need to be rescaled by $\mathbf{w}_{test} = p\mathbf{w}_{train}$, otherwise each neuron will receive an average input signal larger than what it was trained on.

Dropout can be thought of as a type of *ensembling*, as each training step uses a slightly different (but not independent) network (different configuration of neurons as some are dropped out), and these are combined during test time to produce a prediction. The end result is a network that generalises better.

Batch normalisation
Batch normalisation is a method originally developed to reduce vanishing gradients, but it also has a regularising effect.

We add an additional operation just before the activation function, that normalises the input for each batch (using the batch's mean and variance), then scales the results using two new learnable parameters.

It can be added to individual layers in a network, each with their own learnable parameters.
It is implemented into the TF2 keras API as a BatchNormalization layer.

Data augmentation
The best way to reduce overfitting and make the model perform better is to have more data. However, this isn't always possible. We can use our existing data to create new data by augmenting

it. For example, if we are working with image data we can change it in various ways: move, crop, recolour, etc. TensorFlow has methods of doing this during training instead of storing the new data on the disk. You have to be careful though: if you augment a '6' by rotating it 180 degrees, it becomes a 9, and the machine will be taught that this 9 is actually a 6.

Architecture
This is how the network is constructed. Some layers have trainable parameters (e.g. Dense) and some don't (e.g. Activation layers).

Convolution
We can have a convolutional layer.

Input can be represented as a 3D matrix, e.g. an input of 50x50x3 may correspond to an RGB 50x50 image with 3 colour channels. A kernel K is applied to produce an output O.

Recall the output is $\hat{y} = \Phi(\sum_i w_i x_i + b)$ where Φ is the activation function, w_i is the weight corresponding to each x_i and b is the bias.

The analogue for this with matrices is the same but we have the matrices I and K instead of w and x.

Section 2.2.1 of the notes shows an example where we have a 4x4x1 matrix I and apply a 3x3(x1x1) filter kernel. The convolution of this filter is applied to each position of the image. The kernel is applied in each corner of the matrix as shown below.

$$\begin{bmatrix} \mathbf{0}_{\times 1} & \mathbf{1}_{\times 2} & \mathbf{2}_{\times 3} & 3 \\ \mathbf{4}_{\times 3} & \mathbf{5}_{\times 4} & \mathbf{4}_{\times 2} & 4 \\ \mathbf{6}_{\times 2} & \mathbf{5}_{\times 1} & \mathbf{7}_{\times 1} & 1 \\ 1 & 2 & 2 & 8 \end{bmatrix} \to \begin{bmatrix} 0+2+6+12+20+8+12+5+7=72 & \\ & \end{bmatrix}$$

$$\begin{bmatrix} 0 & \mathbf{1}_{\times 1} & \mathbf{2}_{\times 2} & \mathbf{3}_{\times 3} \\ 4 & \mathbf{5}_{\times 3} & \mathbf{4}_{\times 4} & \mathbf{4}_{\times 2} \\ 6 & \mathbf{5}_{\times 2} & \mathbf{7}_{\times 1} & \mathbf{1}_{\times 1} \\ 1 & 2 & 2 & 8 \end{bmatrix} \to \begin{bmatrix} 72 & 1+4+9+15+16+8+10+7+1=71 \\ & \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ \mathbf{4}_{\times 1} & \mathbf{5}_{\times 2} & \mathbf{4}_{\times 3} & 4 \\ \mathbf{6}_{\times 3} & \mathbf{5}_{\times 4} & \mathbf{7}_{\times 2} & 1 \\ \mathbf{1}_{\times 2} & \mathbf{2}_{\times 1} & \mathbf{2}_{\times 1} & 8 \end{bmatrix} \to \begin{bmatrix} 72 & 71 \\ 4+10+12+18+20+14+2+2+2=84 & \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & \mathbf{5}_{\times 1} & \mathbf{4}_{\times 2} & \mathbf{4}_{\times 3} \\ 6 & \mathbf{5}_{\times 3} & \mathbf{7}_{\times 4} & \mathbf{1}_{\times 2} \\ 1 & \mathbf{2}_{\times 2} & \mathbf{2}_{\times 1} & \mathbf{8}_{\times 1} \end{bmatrix} \to \begin{bmatrix} 72 & 71 \\ 84 & 84 \end{bmatrix}.$$

Another way of doing this is to reshape the 4x4 matrix into a column vector with 0, 1, 2, 3, 4, 5, 4, 4, etc. as matrix elements.

The kernel can also be changed into a form where if you take the matrix product of K* and I*, it will give a 1x4 matrix with the same numbers as the 2x2 matrix from multiplying using the method described previously.

$$K = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 2 \\ 2 & 1 & 1 \end{bmatrix}.$$

$$K^* = \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{3} & 0 & \mathbf{3} & \mathbf{4} & \mathbf{2} & 0 & \mathbf{2} & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & \mathbf{2} & \mathbf{3} & 0 & \mathbf{3} & \mathbf{4} & \mathbf{2} & 0 & \mathbf{2} & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & \mathbf{3} & 0 & \mathbf{3} & \mathbf{4} & \mathbf{2} & 0 & \mathbf{2} & \mathbf{1} & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & \mathbf{3} & 0 & \mathbf{3} & \mathbf{4} & \mathbf{2} & 0 & \mathbf{2} & \mathbf{1} & \mathbf{1} \end{bmatrix}$$

Padding
Size of input image shrinks after a series of convolutions, so information is lost. To solve this, we use padding. Usually filled with zeroes. Padding goes around the matrix.

Half padding: padding value P = (K - 1) / 2. Results in an output image with the same dimensions as the input image.

Half padding, which is sometimes called *same padding* is when the padding value is $P = \frac{K-1}{2}$.
This results in an output image with the same dimensions as the input image.

Full padding is when the padding value is $P = K - 1$.
This ensures that every possible convolution of the kernel with the input is considered equally.

Any other value of padding can be used, it does not need to be symmetric in any of the 4 axes (bottom, left, top, right).

Stride
Determines how many pixels each filter moves across before it is applied to the image.

$$\begin{aligned} O_x &= (I_x - K + 2P)/S + 1 \\ O_y &= (I_y - K + 2P)/S + 1 \\ O_z &= N, \end{aligned} \qquad (2.2.8)$$

A bias value is added, and an activation function is applied after using convolution with a filter.

Numbers of Filters and Channels

Most images have 3 channels (the depth of the two-dimensional image): red, green, and blue colour values.

For multi-channel inputs, a filter with the same number of channels is applied, and the resulting feature maps are summed together. Bias is added to produce the output feature map.

The number of filters determines the number of channels of the output feature map, and the depth (number of channels) of the filter must always be the same as the number of input channels.
Each filter looks for different features within the input data.
Each feature map has its own bias.

Valid padding refers to no padding.

Stride
The stride determines how many pixels each filter moves across before it is applied to the image.

The output size of a layer used in a conventional (matrix multiplication) neural network is independent of the input size.
However in convolutional layers, the output size is dependent on the input size, the stride, padding, and the kernel size.

The output of the convolutional layer is a matrix of size W_2 x H_2 x D_2 pixels where:

$$O_x = (I_x - K + 2P)/S + 1$$
$$O_y = (I_y - K + 2P)/S + 1$$
$$O_z = N$$

Where N is the number of filters, P is the padding, S is the stride, and K is the kernel size.
For pooling layers $P = 0$.

Meeting
Convolution: passes filter over network. Filter picks up same activation even though feature is in a different place

Small filters used (3x3 or 5x5)

Pooling: can specify size of pooling and type of pooling.

Week 7

Due to illness the project meeting was delayed until 08/11/21. During it we discussed the previous week's notes, clarifying parts such as why a convolutional neural network (CNN) is used over a conventional one (it is better at generalisation, as the position of an image does not affect the outcome for convolution like it would for matrix multiplication). We then went through section 2.2.3 on **Pooling**, which are layers in a CNN that reduce the size of the input (e.g. the max layer has a kernel that just takes the maximum value in parts of the input matrix equal in size to the kernel), and through section 2.2.4 (**Fully Connected**), which detailed how the dense layers are used in a CNN in order to aggregate together the information from the convolutional and pooling layers, in order to make a classification of the input data.

Then we went through a CNN example in Tensorflow by going through the tutorial (*https://www.tensorflow.org/tutorials/images/cnn*). This included an explanation of what all the layers in a CNN do, by using a CNN to classify the images of the CIFAR10 dataset (*https://www.cs.toronto.edu/~kriz/cifar.html*), a dataset of 32x32 pixel RGB images of animals.
We were given the task to modify the CNN in the example in Google Colab, to see how different parameters change the results of the neural network. For instance decreasing the number of filters of each convolutional layer decreases the runtime, but also greatly decreases the accuracy of the model.

We also were given an overview of the Gravitational Lens Finding Challenge (*http://metcalf1.difa.unibo.it/blf-portal/gg_challenge.html*), a competition which ran from November 25, 2016 until February 5, 2017. The aim of the competition was to find a way to most accurately look for gravitational lensing in large sets of images. A paper was published on the results titled 'The Strong Gravitational Lens Finding Challenge' (*https://arxiv.org/pdf/1802.03609.pdf*), which showed that the most accurate methods (when trained using a set of images generated from a N-body simulation) were consistently using CNNs. We will now be studying this dataset rather than the Quijote Simulations dataset.

We were given the task of reading the paper to understand what sort of CNNs were used in the challenge.

In the challenge two different datasets were used, representing multi-band ground data and single-band space data. It was shown that the algorithms being trained on the ground data were generally more accurate, showing that colour as a feature was important in distinguishing evidence of gravitational lensing from just curved galaxies.

All of the methods proved less accurate on real gravitational lensing data, meaning that the simulation needed to be improved for the algorithms to accurately look for gravitational lensing, when trained using the test data.

The aim of this project is to use newer CNN methods in order to try and see how well our neural network does in identifying gravitational lensing.

Week 8

This week we downloaded the space-based training dataset from the Gravitational Lens Finding Challenge (*http://metcalf1.difa.unibo.it/blf-portal/gg_challenge.html*). We collected the 20,000 separate training FITS files into one folder for use later.

We were given a resource in our project meeting in order to learn how to manipulate FITS files using astropy: *https://docs.astropy.org/en/stable/io/fits/index.html*

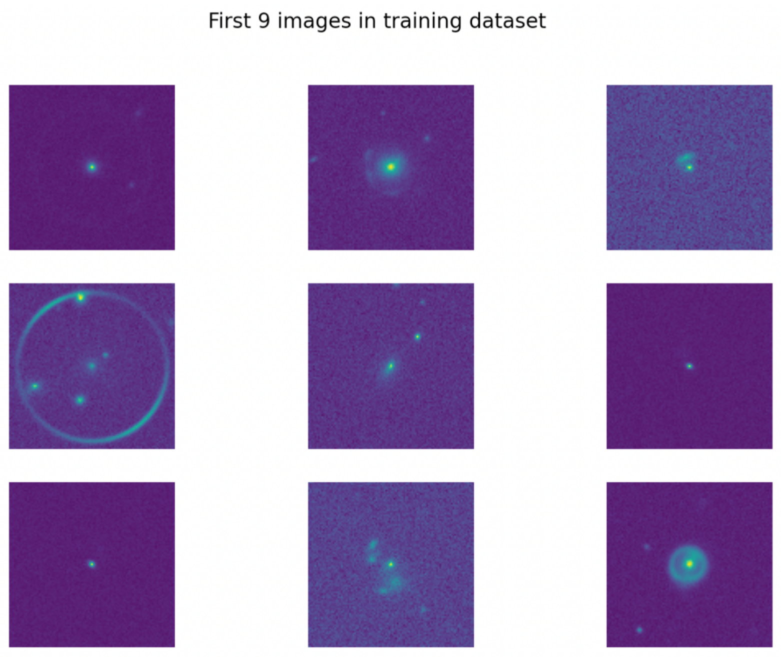
FITS files can combine multiple images into the same file. However, when I used GIMP to look at the images I realised they came as separate files so I combined them into one file using this script.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits

# Amount of images to open (0 to 17999 i.e. first 18000 images)
# (NOTE 8/1/22: This was wrong. N should have been 18000, because the for loop
stops on the number before N.)
N = 17999

# File name
file_name = 'SpaceBasedTraining/Data/imageEUC_VIS-1{0:05d}.fits'

# Open every file
hdul = fits.open(file_name.format(0))
```

```
for i in range(N):
    with fits.open(file_name.format(i)) as hdu:
        hdul.append(hdu[0])

hdul.writeto(f'trainingdata1-{N}stacked.fits')
```

To test it was working I combined the first 9 of the images into one file. I then plotted them using Matplotlib with this script:

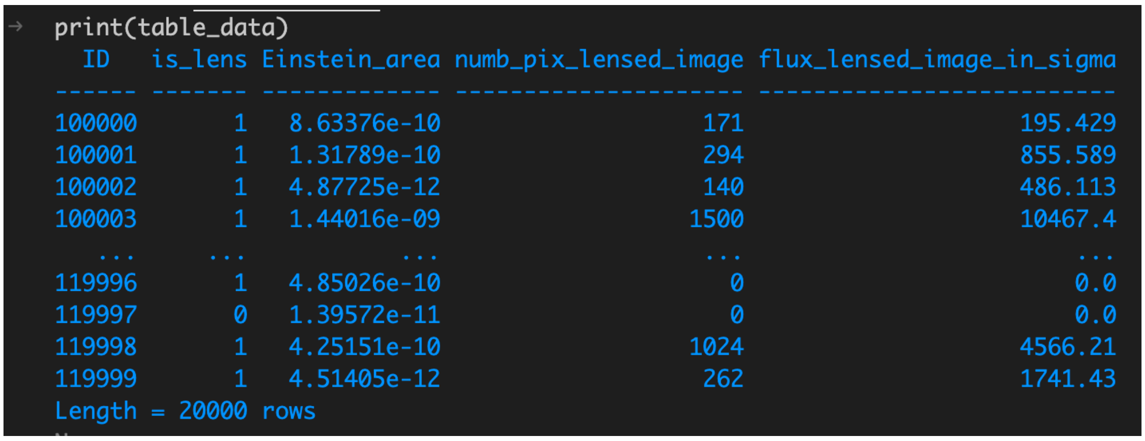
```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits, ascii

N = 9

# I created the file to be 'trainingdata1-Nstacked.fits' in the other script
file_name = f'trainingdata1-{N}stacked.fits'
hdul = fits.open(file_name)

# Get data from each image
data = [hdu.data for hdu in hdul]

# Show each image in an individual subplot
for n in range(N):
    plt.subplot(3, 3, n+1)
    plt.imshow(data[n])
    plt.axis('off')

# Add title above all subplots and show image
plt.suptitle(f'First {N} images in training dataset')
plt.show()
```



We met up in person and worked on the project further. In this session we loaded the CSV data (ascii imported earlier).

```
# Open CSV table
```

```
table_file_name = 'SpaceBasedTraining/classifications.csv'
table_data = ascii.read(table_file_name)
```



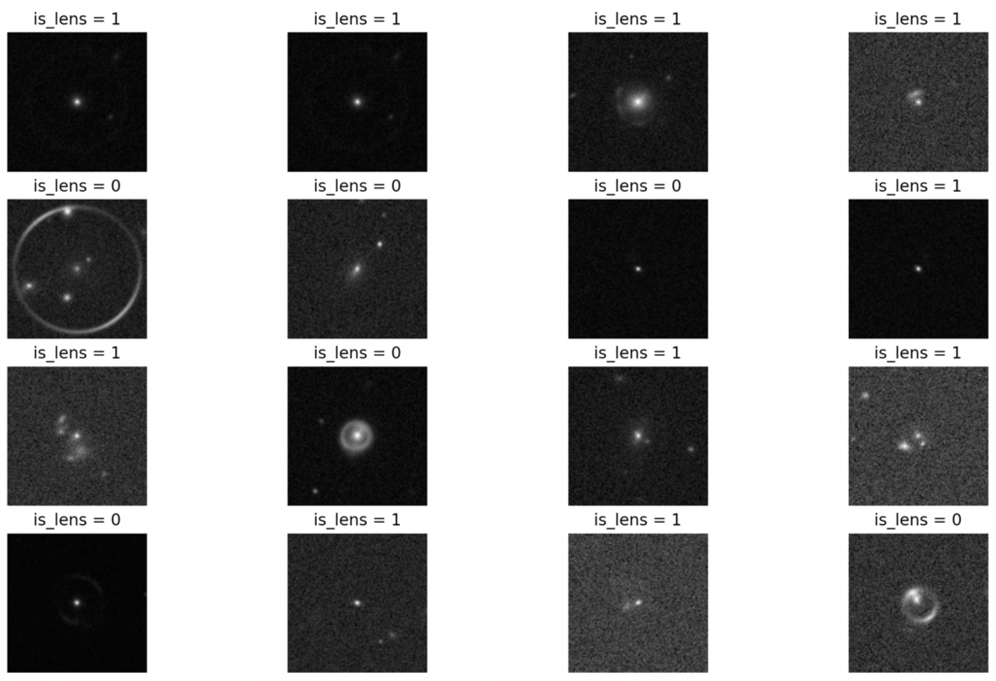
```
print(table_data)
  ID   is_lens Einstein_area numb_pix_lensed_image flux_lensed_image_in_sigma
------ ------- ------------- --------------------- --------------------------
100000       1   8.63376e-10                   171                    195.429
100001       1   1.31789e-10                   294                    855.589
100002       1   4.87725e-12                   140                    486.113
100003       1   1.44016e-09                  1500                    10467.4
   ...     ...           ...                   ...                        ...
119996       1   4.85026e-10                     0                        0.0
119997       0   1.39572e-11                     0                        0.0
119998       1   4.25151e-10                  1024                    4566.21
119999       1   4.51405e-12                   262                    1741.43
Length = 20000 rows
```

The CSV data is stored as a 2D array and the "is_lens" column can be accessed by using `table_data['is_lens']`. Each element of the array matches up with the element of the same index in the HDUList storing the images, so we could plot the images and show whether or not the lenses are changed. We also tried using a different colour map. The script now looked like this:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits, ascii
N = 16

# I created the file to be 'trainingdata1-Nstacked.fits' in the other script
file_name = f'training_data_stacked.fits'
hdul = fits.open(file_name)

# Get data from each image
data = [hdu.data for hdu in hdul]

# Open CSV table
table_file_name = 'SpaceBasedTraining/classifications.csv'
table_data = ascii.read(table_file_name)

# Link images to IDs
is_lens = table_data['is_lens']
number_of_lenses = np.count_nonzero(is_lens==1)
print(f'Number of lenses = {number_of_lenses}')

# Print the data
print(table_data)

# Show each image in an individual subplot
for n in range(N):
    plt.subplot(np.sqrt(N), np.sqrt(N), n+1)           # Create subplot
    plt.imshow(data[n], cmap='gray')                   # Show image in subplot
    plt.title(f'is_lens = {is_lens[n]}')               # Show if it's a lens or
not
    plt.axis('off')                                    # Turn off the axis ticks.

# Add title above all subplots and show
plt.suptitle(f'First {N} images in training dataset')
plt.show()
```

This is the plot of the images:

First 16 images in training dataset



We need two image sets, one to train the model and one to test the model. We chose to split the data into two datasets of the first 18,000 and the last 2,000 images, which were stored as two separate FITS files.

```
import numpy as np
from astropy.io import fits

# File names
bigfilename = 'all_training_data_stacked.fits'
training_data_filename = 'training_data_stacked.fits'
validation_data_filename = 'validation_data_stacked.fits'

# Create empty lists to add elements to from the big list.
training_data = fits.HDUList()
validation_data = fits.HDUList()

# Open big list
bighudl = fits.open(bigfilename)

# Add files to list 1 from 1 to 17999 and list 2 from 18000 to 19999
for i in range(17999):
    training_data.append(bighudl[i])
for i in range(19999)[18000:]:
    validation_data.append(bighudl[i])

# Write files
training_data.writeto(training_data_filename)
validation_data.writeto(validation_data_filename)
```

We then thought about how we could normalise this data, in order to help reduce gradient vanishing. We found that the data ranged from -4.295…e-12 to 2.174…e-09, with most images having a maximum value of around 1e-11. We will find the normalisation by finding the mean and standard deviation of the data, then using the formula $Z = \frac{X-\mu}{\sigma}$, where Z is the normalised data, X is the unnormalised data, μ is the mean and σ is the standard deviation.

Next week we will implement the data normalisation into the code and implement a basic machine learning model using TensorFlow to see how the model performs with this data.

Week 9

This week we created a model in tensorflow to fit the data to, then tested it on the test data. We used the code from last week to open the data using astropy: the training and validation datasets are opened, and the data extracted as numpy arrays. The data is then normalised in order to reduce any gradient vanishing.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits, ascii
import tensorflow as tf

# I created the file to be 'trainingdata1-Nstacked.fits' in the other script
file_name = 'training_data_stacked.fits'
hdul = fits.open(file_name)

# Get data from each image
training_data_unnormalised = [hdu.data for hdu in hdul]

# Open CSV table and get labels for images
table_file_name = 'SpaceBasedTraining/classifications.csv'
is_lens = ascii.read(table_file_name)[:18000]['is_lens']
```

We then normalised the data, using the mean and standard deviation method discussed earlier:

```
# Normalisation
mean = np.mean(training_data_unnormalised)
stdev = np.std(training_data_unnormalised)
training_data = (training_data_unnormalised - mean) / stdev
```

We then created a very basic TensorFlow model with an input layer, a hidden layer and an output layer:

```
# Tensorflow model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(101, 101)),
    tf.keras.layers.Dense(128, activation='relu'),
  #We just want the binary score as it's either 1 or 0.
    tf.keras.layers.Dense(1)
])
```

We then compiled the model and fitted the training data and labels:

```
# Compile and fit
model.compile(loss=tf.losses.BinaryCrossentropy(from_logits=True),
                optimizer=tf.optimizers.Adam())
model.fit(training_data, is_lens, epochs=5)
```

To test the data, we loaded the test data (2000 images) and used the evaluate() function. We normalise the test data the same way as the training data:

```
# Load and normalise test data and labels
test_data_filename = 'test_data_stacked.fits'
test_hdul = fits.open(test_data_filename)
test_data = ([hdu.data for hdu in test_hdul] - mean) / stdev
test_is_lens = ascii.read(table_file_name)['is_lens'][18001:]

# Evaluation
train_acc, test_acc = model.evaluate(test_data, test_is_lens, verbose=2)
print('\nTest accuracy: ', test_acc)
```

The output is given here:

```
1999/1 - 0s - loss: 0.7439
Test accuracy: 0.6024371068855713
```

The loss was 0.7439 and the test accuracy was ~0.6. This is a good start. Next, we need to find the AUROC number to compare our model to the one on the scientific paper.

(note 11/1/2022: This is actually a very bad test accuracy, since the proportion of lenses in the data set is 0.6970. This means the model has performed worse than if we picked the images randomly. The reason for this will be discussed in Week 11.)

We also tried to add convolutional 2D layers, but so far this caused both our laptops to completely freeze.

Lecture notes

There is some overfitting going on. To prevent this from happening we can use data augmentation.
https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator

Week 10

```
Epoch 1/10
17999/17999 [==============================] - 91s 5ms/sample - loss: 0.6400 - accuracy: 0.6855 - val_loss: 0.6282 -
val_accuracy: 0.3872
Epoch 2/10
17999/17999 [==============================] - 100s 6ms/sample - loss: 0.6077 - accuracy: 0.6925 - val_loss: 0.6134 -
val_accuracy: 0.6913
Epoch 3/10
17999/17999 [==============================] - 100s 6ms/sample - loss: 0.5881 - accuracy: 0.7040 - val_loss: 0.6273 -
val_accuracy: 0.6963
Epoch 4/10
17999/17999 [==============================] - 98s 5ms/sample - loss: 0.5140 - accuracy: 0.7720 - val_loss: 0.6491 -
val_accuracy: 0.6003
Epoch 5/10
17999/17999 [==============================] - 105s 6ms/sample - loss: 0.4154 - accuracy: 0.8283 - val_loss: 0.7843 -
val_accuracy: 0.6078
Epoch 6/10
17999/17999 [==============================] - 107s 6ms/sample - loss: 0.3100 - accuracy: 0.8731 - val_loss: 0.9565 -
val_accuracy: 0.4892
Epoch 7/10
17999/17999 [==============================] - 116s 6ms/sample - loss: 0.2140 - accuracy: 0.9222 - val_loss: 1.1056 -
val_accuracy: 0.5938
Epoch 8/10
17999/17999 [==============================] - 109s 6ms/sample - loss: 0.1329 - accuracy: 0.9586 - val_loss: 1.5714 -
val_accuracy: 0.6278
```

```
Epoch 9/10
17999/17999 [==============================] - 103s 6ms/sample - loss: 0.0743 - accuracy: 0.9836 - val_loss: 1.7279 -
val_accuracy: 0.5913
Epoch 10/10
17999/17999 [==============================] - 99s 5ms/sample - loss: 0.0398 - accuracy: 0.9929 - val_loss: 1.9387 -
val_accuracy: 0.5853
1999/1 - 3s - loss: 1.4182 - accuracy: 0.5853

Test accuracy:  [1.9387394425092548, 0.58529264]
```

This week we updated the model in TensorFlow to be more complex. This is the updated script:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits, ascii
import tensorflow as tf

# I created the file to be 'trainingdata1-Nstacked.fits' in the other script
file_name = 'training_data_stacked.fits'
hdul = fits.open(file_name)

# Get data from each image
training_data_unnormalised = [hdu.data for hdu in hdul]

# Open CSV table and get labels for images
table_file_name = 'SpaceBasedTraining/classifications.csv'
is_lens = ascii.read(table_file_name)[:17999]['is_lens']

# Normalisation
mean = np.mean(training_data_unnormalised)
stdev = np.std(training_data_unnormalised)
training_data = np.expand_dims((training_data_unnormalised - mean) / stdev, axis=-
1)

# Load and normalise test data and labels
test_data_filename = 'test_data_stacked.fits'
test_hdul = fits.open(test_data_filename)
test_data = np.expand_dims(([hdu.data for hdu in test_hdul] - mean) / stdev, axis=-
1)
test_is_lens = ascii.read(table_file_name)['is_lens'][18001:]

# Tensorflow model
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(9, kernel_size=(3, 3), input_shape=(101, 101, 1),
activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1)
])

# Compile and fit
model.compile(
    loss=tf.losses.BinaryCrossentropy(from_logits=True),
    optimizer=tf.optimizers.Adam(),
    metrics=['accuracy']
)
model.fit(training_data, is_lens, epochs=10, validation_data=(test_data,
test_is_lens))

# Evaluation
test_acc = model.evaluate(test_data, test_is_lens, verbose=2)
print('\nTest accuracy: ', test_acc)
```

We tried to implement the AUC metric, but when passing in the argument `from_logits=True`, the compiler said there was no argument for from_logits. We will calculate the AUC score manualy next week by getting the model to make predictions.

This week, we added more layers to the model. We added a Conv2D layer. We had a lot of trouble getting the input shape to work, and realised we needed to expand the training data to be 3 dimensional by using the np.expand_dims() function. At the moment the image data is just stored as an array with its rows and columns specified (i.e. Its shape is only (101,101)) .The `axis=-1` paramenter adds an extra dimension at the end (effectively adds another set of square brackets around the entire array), to specify the number of channels to be one.

Week 11

This week we will calculate the AUC score and preprocess our data to reduce overfitting.

We have found out from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html#sklearn.metrics.roc_auc_score that this can be done using the sklearn library as follows:

```
from sklearn import metrics

#This calculates and gives the AUC score.
predictions = model.predict(x_test)
auc_score = metrics.roc_auc_score(y_test, predictions)
print('\nAUC score: ', auc_score)
```

Our model last week produced very high values for accuracy of training data, but low values of accuracy for the validation data. This was due to the overfitting that was occurring. Also, some of the labels didn't align with the images due to a mistake compiling the training and validation files.

```
Epoch 1/10
563/563 [==============================] - 28s 49ms/step - loss: 0.6178 - accuracy: 0.6808 - val_loss: 0.6137 -
val_accuracy: 0.6970
Epoch 2/10
563/563 [==============================] - 27s 48ms/step - loss: 0.6147 - accuracy: 0.6956 - val_loss: 0.6135 -
val_accuracy: 0.6970
Epoch 3/10
563/563 [==============================] - 27s 49ms/step - loss: 0.6143 - accuracy: 0.6973 - val_loss: 0.6140 -
val_accuracy: 0.6970
Epoch 4/10
563/563 [==============================] - 28s 49ms/step - loss: 0.6134 - accuracy: 0.6982 - val_loss: 0.6140 -
val_accuracy: 0.6970
Epoch 5/10
563/563 [==============================] - 27s 48ms/step - loss: 0.6135 - accuracy: 0.6986 - val_loss: 0.6137 -
val_accuracy: 0.6970
Epoch 6/10
563/563 [==============================] - 27s 47ms/step - loss: 0.6129 - accuracy: 0.6983 - val_loss: 0.6137 -
val_accuracy: 0.6970
Epoch 7/10
563/563 [==============================] - 26s 47ms/step - loss: 0.6131 - accuracy: 0.6976 - val_loss: 0.6151 -
val_accuracy: 0.6970
```

```
Epoch 8/10
563/563 [==============================] - 27s 47ms/step - loss: 0.6127 - accuracy: 0.6985 - val_loss: 0.6135 -
val_accuracy: 0.6970
Epoch 9/10
563/563 [==============================] - 27s 48ms/step - loss: 0.6127 - accuracy: 0.6986 - val_loss: 0.6156 -
val_accuracy: 0.6970
Epoch 10/10
563/563 [==============================] - 27s 48ms/step - loss: 0.6129 - accuracy: 0.6981 - val_loss: 0.6186 -
val_accuracy: 0.6650
63/63 - 1s - loss: 0.6186 - accuracy: 0.6650 - 576ms/epoch - 9ms/step
Validation accuracy:  [0.6185726523399353, 0.6650000214576721]
AUC score:  0.5088421144840453
```

The AUC score of 0.5088… being so close to 0.5 suggested that the model is entirely random. Additionally, the val_accuracy of 0.6970 is exactly the same as the proportion of images that are lenses in the validation data.

New model from paper

We took a model from the challenge paper and implemented it into Python to see how it fared. We also moved the code to Google Colab, since it was taking too much time to run on our laptops.

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(16, kernel_size=(4, 4), input_shape=(101, 101, 1),
activation='relu'),
    tf.keras.layers.Conv2D(16, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Conv2D(128, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Conv2D(128, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1024),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1024),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1024),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])
```

This is the LASTRO EPFL architecture. In the paper it produced an AUC score of 0.93.

Data augmentation

To prevent overfitting of the data, we implemented data augmentation; this randomly transforms the images, to create new training data from the existing training dataset. In the meeting we discussed that the best methods to use were rotations, reflections, and resizing, as these were the augmentations used by the most accurate models in the Gravitation Lens Challenge. We were told to

avoid shearing, as that changes the physics of the images, and so wouldn't be a good representation of real life.
We used the keras ImageDataGenerator:
(*https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator*)
This is our data generator:

```
# Data generator
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    horizontal_flip=True,
    vertical_flip=True,
    rotation_range=90,
    zoom_range=0.4
)

datagen.fit(x_train)
```

When passing it to the model, we now use this code:

```
model.fit(datagen.flow(x_train, y_train), epochs=NUM_EPOCHS,
validation_data=(x_test, y_test))
```

Note: I have changed the training data and labels from (training_data_normalised, is_lens) to (x_train, y_train) and the validation data and labels from (validation_data_normalised, validation_is_lens) to (x_test, y_test). This is to be more consistent with tutorials we found online.

History
Other improvements to the model we made were to make history plots of the test and validation accuracy against epochs, in order to help analyse when the model is most accurate for validation data. Information on the History object are given in:
https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/History
We did this with:

```
import matplotlib.pyplot as plt

history =model.fit(datagen.flow(x_train, y_train), epochs=NUM_EPOCHS,
validation_data=(x_test, y_test))

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
plt.show()
```

Progress
We discovered that when creating the large FITS file, we used this script:

```
# Open every file
hdul = fits.open(file_name.format(0))

for i in range(N):
    with fits.open(file_name.format(i)) as hdu:
        hdul.append(hdu[0])
```

This script opens image 0 and appends every image from 0 to N. This means image 0 was included twice, so the labels corresponded to the wrong image: label 1 corresponds with image 0, label 2 to image 1 and so on. We fixed this error by using this script instead:

```
# Open every file
hdul = fits.HDUList()

for i in range(N):
    with fits.open(file_name.format(i)) as hdu:
        hdul.append(hdu[0])
```

The output from the model is now:

```
Epoch 1/10
563/563 [==============================] - 44s 54ms/step - loss: 0.5924 - accuracy: 0.7002 - val_loss: 0.5447 -
val_accuracy: 0.7385
Epoch 2/10
563/563 [==============================] - 29s 52ms/step - loss: 0.4617 - accuracy: 0.7542 - val_loss: 0.5374 -
val_accuracy: 0.7900
Epoch 3/10
563/563 [==============================] - 29s 52ms/step - loss: 0.4020 - accuracy: 0.7781 - val_loss: 0.3714 -
val_accuracy: 0.8130
Epoch 4/10
563/563 [==============================] - 30s 53ms/step - loss: 0.3791 - accuracy: 0.7937 - val_loss: 0.3430 -
val_accuracy: 0.8035
Epoch 5/10
563/563 [==============================] - 29s 52ms/step - loss: 0.3599 - accuracy: 0.8022 - val_loss: 0.3378 -
val_accuracy: 0.8280
Epoch 6/10
563/563 [==============================] - 29s 52ms/step - loss: 0.3503 - accuracy: 0.8107 - val_loss: 0.3360 -
val_accuracy: 0.8310
Epoch 7/10
563/563 [==============================] - 29s 52ms/step - loss: 0.3425 - accuracy: 0.8116 - val_loss: 0.3780 -
val_accuracy: 0.8185
Epoch 8/10
563/563 [==============================] - 30s 53ms/step - loss: 0.3396 - accuracy: 0.8179 - val_loss: 0.3457 -
val_accuracy: 0.8205
Epoch 9/10
563/563 [==============================] - 30s 53ms/step - loss: 0.3310 - accuracy: 0.8180 - val_loss: 0.3015 -
val_accuracy: 0.8450
Epoch 10/10
563/563 [==============================] - 30s 54ms/step - loss: 0.3238 - accuracy: 0.8296 - val_loss: 0.3876 -
val_accuracy: 0.8250
63/63 - 1s - loss: 0.3876 - accuracy: 0.8250 - 751ms/epoch - 12ms/step
Validation accuracy:  [0.387556791305542, 0.824999988079071]
AUC score:  0.9199066248088223
```

The AUC score is roughly 0.92. This is a dramatic improvement and comparable to the AUC score of 0.93 provided by the paper. We ran the code a few times and got AUC scores ranging from 0.91 to 0.95 for 10 epochs.

Next week, we will try implementing the EfficientNet neural network. This will require us to change the image to be 3 channels and we will have to resize the images.

Meeting
Implement mobile net or efficient net

Efficientnet : include_top = false
Model.trainable = false
Follow article https://keras.io/examples/vision/image_classification_efficientnet_fine_tuning/
Try with one channel and see if it works

Week 12

This week we have been tasked to update the neural network to use the EfficientNetB0 model rather than the ASTRO EPFL model we used last week. The ASTRO EPFL model was used in 2017. Since then, machine learning technology has advanced. EfficientNet is a more up-to-date model which will hopefully give a higher AUC score than the ASTRO EPFL model did.

Resizing data
To implement this model, we must adapt the data to fit it. This means resizing it from 101x101 to 224x224 pixels and changing the number of channels from 1 to 3. We implemented some code to do this:

```python
import numpy as np
import tensorflow as tf
from astropy.io import fits, ascii
from skimage import transform

FILE_NAME = 'training_data_stacked_2021_12_01.fits'
train_hdul = fits.open(FILE_NAME)

# Get data from each image
training_data_unnormalised = np.array([hdu.data for hdu in train_hdul])

# Normalisation
mean = np.mean(training_data_unnormalised)
stdev = np.std(training_data_unnormalised)
training_data_normalised = (training_data_unnormalised - mean) / stdev

# Resize data
training_data_normalised_resized \
    = np.array([transform.resize(image, (224, 224)) for image in
training_data_normalised])

# Delete variables to save memory
train_hdul.close()
del train_hdul
del training_data_unnormalised
del training_data_normalised

# Make 1 channel image into 3 channel image for EfficientNet model
x_train = np.empty(shape=(18000, 224, 224, 3))
x_train[:, :, :, 0] = training_data_normalised_resized
x_train[:, :, :, 1] = training_data_normalised_resized
x_train[:, :, :, 2] = training_data_normalised_resized
del training_data_normalised_resized

# Make numpy array to HDUList to write to a file
hdul_xtrain = fits.HDUList()
for img in x_train:
    hdul_xtrain.append(fits.ImageHDU(data=img))

# Write file
```

```
HDUL_XTRAIN_RESIZED_PATH = 'training_data_stacked_resized_2021_12_06.fits'
hdul_xtrain.writeto(HDUL_XTRAIN_RESIZED_PATH, overwrite=True)
hdul_xtrain.close()
del hdul_xtrain
```

We did this for the training and validation data.

Having multiple variables with the data in (training_data_unnormalised, training_data_normalised, x_train, etc.) took up a lot of memory. Therefore, the first few times I tried to run it, my computer ran out of memory, so I had to delete variables in the code as it was going along.

The data files came to over 23GB in total and didn't fit on Google Drive. Additionally, Colab would have run out of RAM. Therefore, we upgraded to Colab Pro and 200GB storage for a month.

More problems with Colab

We ran out of RAM again. To fix this we used the original data again, with shape (101, 101, 1). In the code we changed it to be 3 channels so it would work with the model. This is the model we used:

```
# Base model. We will add layers to this.
base_model = tf.keras.applications.EfficientNetB0(
                      include_top=False,
                      weights=None,
                      input_shape=(101, 101, 3),
                      pooling='max'
            )
# Add layers to base model
model = tf.keras.Sequential()
model.add(base_model)
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dropout(0.2, name='dropout_out'))
model.add(tf.keras.layers.Dense(256, activation='relu', name='fc1'))
model.add(tf.keras.layers.Dense(128, activation='relu', name='fc2'))
model.add(tf.keras.layers.Dense(1, activation='relu', name='fc_out'))
# Compile the model
model.compile(
    loss=tf.losses.BinaryCrossentropy(),
    optimizer=tf.optimizers.Adam(),
    metrics=['accuracy']
)
```

On the first time running it with 10 epochs, the AUC score was ~0.902 which is lower than the model from the paper.

```
Epoch 1/10 563/563 [==============================] - 54s 82ms/step - loss: 0.6277 - accuracy: 0.7063 - val_loss:
0.7675 - val_accuracy: 0.5390

Epoch 2/10 563/563 [==============================] - 45s 79ms/step - loss: 0.5076 - accuracy: 0.7542 - val_loss:
1.6599 - val_accuracy: 0.6970

Epoch 3/10 563/563 [==============================] - 45s 79ms/step - loss: 0.4400 - accuracy: 0.7826 - val_loss:
0.4289 - val_accuracy: 0.7955

Epoch 4/10 563/563 [==============================] - 45s 80ms/step - loss: 0.4182 - accuracy: 0.7947 - val_loss:
0.3943 - val_accuracy: 0.7770

Epoch 5/10 563/563 [==============================] - 45s 79ms/step - loss: 0.4028 - accuracy: 0.8000 - val_loss:
0.4181 - val_accuracy: 0.7455
```

```
Epoch 6/10 563/563 [==============================] - 45s 80ms/step - loss: 0.3962 - accuracy: 0.8072 - val_loss:
0.4391 - val_accuracy: 0.7605

Epoch 7/10 563/563 [==============================] - 45s 79ms/step - loss: 0.3783 - accuracy: 0.8152 - val_loss:
0.3562 - val_accuracy: 0.8180

Epoch 8/10 563/563 [==============================] - 44s 79ms/step - loss: 0.3716 - accuracy: 0.8137 - val_loss:
0.3892 - val_accuracy: 0.7885

Epoch 9/10 563/563 [==============================] - 45s 80ms/step - loss: 0.3679 - accuracy: 0.8161 - val_loss:
0.6910 - val_accuracy: 0.7485

Epoch 10/10 563/563 [==============================] - 45s 79ms/step - loss: 0.3575 - accuracy: 0.8223 - val_loss:
0.3794 - val_accuracy: 0.8160
63/63 - 1s - loss: 0.3794 - accuracy: 0.8160 - 928ms/epoch - 15ms/step

Validation accuracy: [0.3793826401233673, 0.8159999847412109]

AUC score: 0.9023484665539725
```

The training loss is decreasing steadily each epoch, but the validation loss goes up and down seemingly randomly. This suggests the model is overfitting to the training data.

Update
The model actually isn't overfitting. We ran the model again for 120 epochs and got an AUC score of 0.96. We ran it again with early stopping, where the model stops after the validation loss stops decreasing. It stopped at epoch 35 and gave a value of 0.95.

```
Epoch 1/120 563/563 [==============================] - 55s 82ms/step - loss: 0.6244 - accuracy: 0.7003 - val_loss:
0.8728 - val_accuracy: 0.3720
Epoch 2/120 563/563 [==============================] - 45s 79ms/step - loss: 0.4812 - accuracy: 0.7604 - val_loss:
0.4361 - val_accuracy: 0.7705
Epoch 3/120 563/563 [==============================] - 45s 79ms/step - loss: 0.4291 - accuracy: 0.7876 - val_loss:
0.4218 - val_accuracy: 0.7940
Epoch 4/120 563/563 [==============================] - 45s 79ms/step - loss: 0.4015 - accuracy: 0.7994 - val_loss:
0.3840 - val_accuracy: 0.8090
Epoch 5/120 563/563 [==============================] - 45s 80ms/step - loss: 0.3804 - accuracy: 0.8117 - val_loss:
0.3504 - val_accuracy: 0.8050
Epoch 6/120 563/563 [==============================] - 45s 80ms/step - loss: 0.3846 - accuracy: 0.8107 - val_loss:
0.3951 - val_accuracy: 0.8010
Epoch 7/120 563/563 [==============================] - 45s 79ms/step - loss: 0.3841 - accuracy: 0.8133 - val_loss:
0.9388 - val_accuracy: 0.7405
Epoch 8/120 563/563 [==============================] - 45s 80ms/step - loss: 0.3617 - accuracy: 0.8239 - val_loss:
0.3798 - val_accuracy: 0.7975
Epoch 9/120 563/563 [==============================] - 45s 80ms/step - loss: 0.3532 - accuracy: 0.8258 - val_loss:
0.3343 - val_accuracy: 0.8295
Epoch 10/120 563/563 [==============================] - 45s 79ms/step - loss: 0.3507 - accuracy: 0.8281 - val_loss:
0.4041 - val_accuracy: 0.8060
Epoch 11/120 563/563 [==============================] - 45s 79ms/step - loss: 0.3387 - accuracy: 0.8353 - val_loss:
0.4378 - val_accuracy: 0.8125
Epoch 12/120 563/563 [==============================] - 45s 79ms/step - loss: 0.3375 - accuracy: 0.8358 - val_loss:
0.3629 - val_accuracy: 0.8250
Epoch 13/120 563/563 [==============================] - 45s 79ms/step - loss: 0.3307 - accuracy: 0.8396 - val_loss:
0.3261 - val_accuracy: 0.8390
Epoch 14/120 563/563 [==============================] - 45s 80ms/step - loss: 0.3233 - accuracy: 0.8404 - val_loss:
0.3019 - val_accuracy: 0.8590
Epoch 15/120 563/563 [==============================] - 44s 79ms/step - loss: 0.3197 - accuracy: 0.8422 - val_loss:
0.4332 - val_accuracy: 0.7655
Epoch 16/120 563/563 [==============================] - 45s 79ms/step - loss: 0.3142 - accuracy: 0.8483 - val_loss:
0.3003 - val_accuracy: 0.8480
```

```
Epoch 17/120 563/563 [==============================] - 44s 78ms/step - loss: 0.3105 - accuracy: 0.8460 - val_loss:
0.2711 - val_accuracy: 0.8680
Epoch 18/120 563/563 [==============================] - 44s 79ms/step - loss: 0.3048 - accuracy: 0.8511 - val_loss:
0.3431 - val_accuracy: 0.8260
Epoch 19/120 563/563 [==============================] - 45s 79ms/step - loss: 0.3012 - accuracy: 0.8511 - val_loss:
0.2662 - val_accuracy: 0.8685
Epoch 20/120 563/563 [==============================] - 44s 78ms/step - loss: 0.2947 - accuracy: 0.8587 - val_loss:
0.3604 - val_accuracy: 0.8130
Epoch 21/120 563/563 [==============================] - 44s 79ms/step - loss: 0.2930 - accuracy: 0.8572 - val_loss:
0.3530 - val_accuracy: 0.8325
Epoch 22/120 563/563 [==============================] - 44s 78ms/step - loss: 0.2905 - accuracy: 0.8571 - val_loss:
0.2688 - val_accuracy: 0.8795
Epoch 23/120 563/563 [==============================] - 45s 79ms/step - loss: 0.2877 - accuracy: 0.8579 - val_loss:
0.2847 - val_accuracy: 0.8810
Epoch 24/120 563/563 [==============================] - 44s 78ms/step - loss: 0.2866 - accuracy: 0.8598 - val_loss:
0.2617 - val_accuracy: 0.8820
Epoch 25/120 563/563 [==============================] - 44s 78ms/step - loss: 0.2802 - accuracy: 0.8629 - val_loss:
0.3624 - val_accuracy: 0.8475
Epoch 26/120 563/563 [==============================] - 44s 79ms/step - loss: 0.2838 - accuracy: 0.8609 - val_loss:
0.2767 - val_accuracy: 0.8705
Epoch 27/120 563/563 [==============================] - 44s 78ms/step - loss: 0.2719 - accuracy: 0.8656 - val_loss:
0.2714 - val_accuracy: 0.8835
Epoch 28/120 563/563 [==============================] - 44s 79ms/step - loss: 0.2706 - accuracy: 0.8689 - val_loss:
0.2555 - val_accuracy: 0.8740
Epoch 29/120 563/563 [==============================] - 44s 78ms/step - loss: 0.2727 - accuracy: 0.8650 - val_loss:
0.3503 - val_accuracy: 0.8030
Epoch 30/120 563/563 [==============================] - 44s 78ms/step - loss: 0.2686 - accuracy: 0.8694 - val_loss:
0.2413 - val_accuracy: 0.8895
Epoch 31/120 563/563 [==============================] - 44s 78ms/step - loss: 0.2662 - accuracy: 0.8718 - val_loss:
0.2745 - val_accuracy: 0.8735
Epoch 32/120 563/563 [==============================] - 44s 78ms/step - loss: 0.2641 - accuracy: 0.8697 - val_loss:
0.2881 - val_accuracy: 0.8730
Epoch 33/120 563/563 [==============================] - 45s 79ms/step - loss: 0.2638 - accuracy: 0.8709 - val_loss:
0.2904 - val_accuracy: 0.8650
Epoch 34/120 563/563 [==============================] - 45s 80ms/step - loss: 0.2623 - accuracy: 0.8731 - val_loss:
0.2647 - val_accuracy: 0.8840
Epoch 35/120 563/563 [==============================] - 45s 79ms/step - loss: 0.2618 - accuracy: 0.8734 - val_loss:
0.3463 - val_accuracy: 0.8495

63/63 - 1s - loss: 0.3463 - accuracy: 0.8495 - 908ms/epoch - 14ms/step

Validation accuracy: [0.3463481366634369, 0.8495000004768372]

AUC score: 0.9515024314483099
```

Friday 7 January

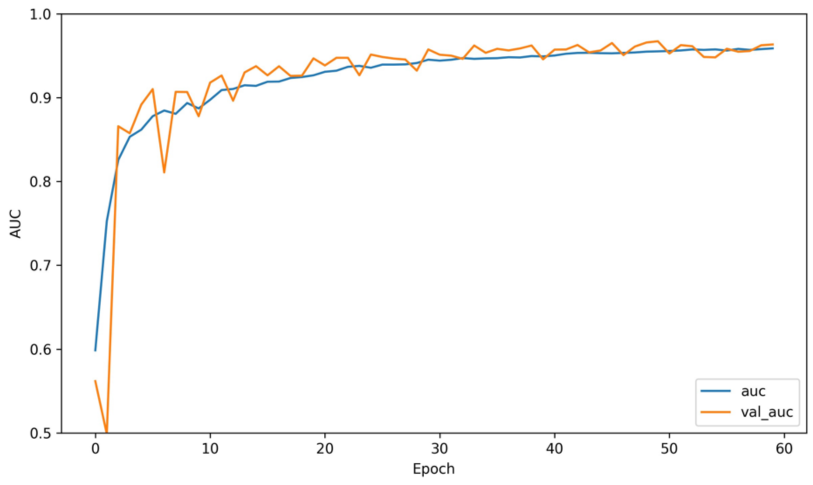
Today we will train models other than EfficientNetB0 and plot the graphs of their validation loss over time, compared to EfficientNetB0. This will help us to see if other models perform better.

```
Epoch 1/100
563/563 [==============================] - 43s 72ms/step - loss: 0.6031 - accuracy: 0.6937 - val_loss: 0.7307 -
val_accuracy: 0.3030
Epoch 2/100
563/563 [==============================] - 40s 70ms/step - loss: 0.4925 - accuracy: 0.7556 - val_loss: 0.6569 -
val_accuracy: 0.7755
Epoch 3/100
563/563 [==============================] - 40s 71ms/step - loss: 0.4409 - accuracy: 0.7766 - val_loss: 0.6154 -
```

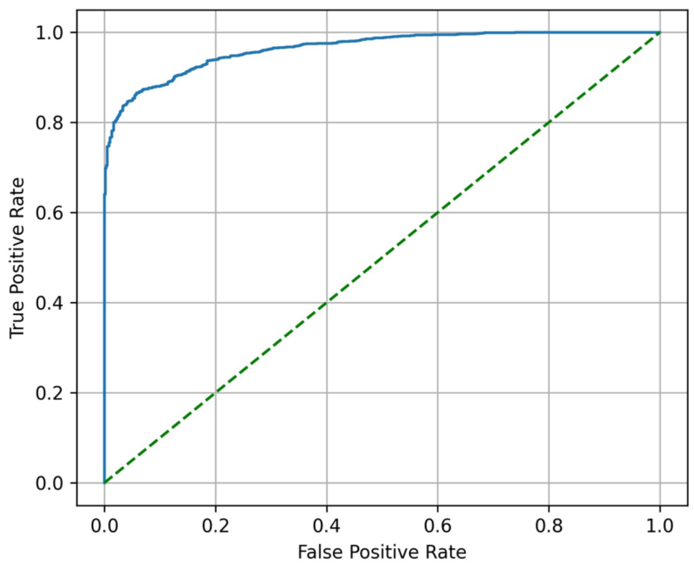
```
val_accuracy: 0.7170
Epoch 4/100
563/563 [==============================] - 40s 71ms/step - loss: 0.4153 - accuracy: 0.7922 - val_loss: 0.7038 -
val_accuracy: 0.5885
Epoch 5/100
563/563 [==============================] - 40s 71ms/step - loss: 0.3980 - accuracy: 0.7999 - val_loss: 0.3817 -
val_accuracy: 0.8090
Epoch 6/100
563/563 [==============================] - 40s 70ms/step - loss: 0.3894 - accuracy: 0.8044 - val_loss: 0.4888 -
val_accuracy: 0.7655
Epoch 7/100
563/563 [==============================] - 40s 71ms/step - loss: 0.3812 - accuracy: 0.8123 - val_loss: 0.3479 -
val_accuracy: 0.8135
Epoch 8/100
563/563 [==============================] - 40s 71ms/step - loss: 0.3703 - accuracy: 0.8137 - val_loss: 0.5654 -
val_accuracy: 0.6125
Epoch 9/100
563/563 [==============================] - 40s 70ms/step - loss: 0.3648 - accuracy: 0.8193 - val_loss: 0.3241 -
val_accuracy: 0.8435
Epoch 10/100
563/563 [==============================] - 40s 70ms/step - loss: 0.3530 - accuracy: 0.8276 - val_loss: 0.3983 -
val_accuracy: 0.7755
Epoch 11/100
563/563 [==============================] - 39s 70ms/step - loss: 0.3560 - accuracy: 0.8263 - val_loss: 0.3921 -
val_accuracy: 0.8040
Epoch 12/100
563/563 [==============================] - 40s 70ms/step - loss: 0.3555 - accuracy: 0.8262 - val_loss: 0.3147 -
val_accuracy: 0.8470
Epoch 13/100
563/563 [==============================] - 40s 71ms/step - loss: 0.3393 - accuracy: 0.8339 - val_loss: 0.3109 -
val_accuracy: 0.8535
Epoch 14/100
563/563 [==============================] - 40s 71ms/step - loss: 0.3359 - accuracy: 0.8338 - val_loss: 0.3357 -
val_accuracy: 0.8230
Epoch 15/100
563/563 [==============================] - 41s 72ms/step - loss: 0.3305 - accuracy: 0.8359 - val_loss: 0.2969 -
val_accuracy: 0.8600
Epoch 16/100
563/563 [==============================] - 40s 70ms/step - loss: 0.3228 - accuracy: 0.8421 - val_loss: 0.2918 -
val_accuracy: 0.8600
Epoch 17/100
563/563 [==============================] - 40s 70ms/step - loss: 0.3228 - accuracy: 0.8415 - val_loss: 0.6531 -
val_accuracy: 0.6895
Epoch 18/100
563/563 [==============================] - 40s 70ms/step - loss: 0.3151 - accuracy: 0.8451 - val_loss: 0.3955 -
val_accuracy: 0.7845
Epoch 19/100
563/563 [==============================] - 39s 70ms/step - loss: 0.3064 - accuracy: 0.8503 - val_loss: 0.3305 -
val_accuracy: 0.8335
Epoch 20/100
563/563 [==============================] - 39s 70ms/step - loss: 0.3095 - accuracy: 0.8501 - val_loss: 0.3176 -
val_accuracy: 0.8425
Epoch 21/100
563/563 [==============================] - 39s 70ms/step - loss: 0.3066 - accuracy: 0.8497 - val_loss: 0.3493 -
val_accuracy: 0.8210
Epoch 22/100
563/563 [==============================] - 40s 70ms/step - loss: 0.3009 - accuracy: 0.8528 - val_loss: 0.2854 -
val_accuracy: 0.8590
Epoch 23/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2951 - accuracy: 0.8558 - val_loss: 0.3221 -
val_accuracy: 0.8545
Epoch 24/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2950 - accuracy: 0.8558 - val_loss: 0.2953 -
```

```
val_accuracy: 0.8515
Epoch 25/100
563/563 [==============================] - 40s 71ms/step - loss: 0.2942 - accuracy: 0.8567 - val_loss: 0.3108 -
val_accuracy: 0.8440
Epoch 26/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2914 - accuracy: 0.8586 - val_loss: 0.3713 -
val_accuracy: 0.7950
Epoch 27/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2873 - accuracy: 0.8601 - val_loss: 0.3318 -
val_accuracy: 0.8205
Epoch 28/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2866 - accuracy: 0.8611 - val_loss: 0.2869 -
val_accuracy: 0.8540
Epoch 29/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2837 - accuracy: 0.8644 - val_loss: 0.2787 -
val_accuracy: 0.8645
Epoch 30/100
563/563 [==============================] - 40s 71ms/step - loss: 0.2842 - accuracy: 0.8605 - val_loss: 0.2579 -
val_accuracy: 0.8775
Epoch 31/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2747 - accuracy: 0.8621 - val_loss: 0.2588 -
val_accuracy: 0.8815
Epoch 32/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2755 - accuracy: 0.8670 - val_loss: 0.2514 -
val_accuracy: 0.8710
Epoch 33/100
563/563 [==============================] - 40s 71ms/step - loss: 0.2752 - accuracy: 0.8653 - val_loss: 0.2549 -
val_accuracy: 0.8755
Epoch 34/100
563/563 [==============================] - 40s 71ms/step - loss: 0.2721 - accuracy: 0.8692 - val_loss: 0.2750 -
val_accuracy: 0.8780
Epoch 35/100
563/563 [==============================] - 40s 71ms/step - loss: 0.2710 - accuracy: 0.8711 - val_loss: 0.3038 -
val_accuracy: 0.8570
Epoch 36/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2732 - accuracy: 0.8690 - val_loss: 0.3905 -
val_accuracy: 0.7850
Epoch 37/100
563/563 [==============================] - 40s 71ms/step - loss: 0.2654 - accuracy: 0.8713 - val_loss: 0.4265 -
val_accuracy: 0.8445
Epoch 38/100
563/563 [==============================] - 40s 71ms/step - loss: 0.2691 - accuracy: 0.8693 - val_loss: 0.3341 -
val_accuracy: 0.8530
Epoch 39/100
563/563 [==============================] - 40s 71ms/step - loss: 0.2671 - accuracy: 0.8667 - val_loss: 0.3188 -
val_accuracy: 0.8505
Epoch 40/100
563/563 [==============================] - 40s 71ms/step - loss: 0.2674 - accuracy: 0.8700 - val_loss: 0.3056 -
val_accuracy: 0.8400
Epoch 41/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2695 - accuracy: 0.8691 - val_loss: 0.3308 -
val_accuracy: 0.8345
Epoch 42/100
563/563 [==============================] - 40s 70ms/step - loss: 0.2670 - accuracy: 0.8714 - val_loss: 0.3489 -
val_accuracy: 0.8260
63/63 - 0s - loss: 0.3489 - accuracy: 0.8260 - 480ms/epoch - 8ms/step
Validation accuracy:  [0.34886592626571655, 0.8259999752044678]
AUC score:  0.94328
```

Here is a graph from when we ran the program with early stopping with a patience of 10. The program stopped at 60 epochs.



And the AUC curve:



Saturday 8 January

Today we will introduce the confusion matrix. This shows how often the model wrongly categorises images which are gravitational lenses vs images which aren't.

Here is the confusion matrix with a threshold of 0.5.

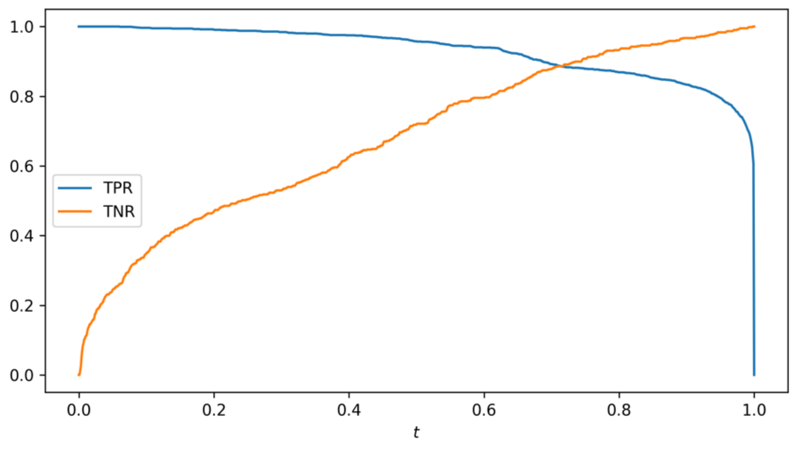
	Predicted lens	Predicted not lens
Actual not lens	436	170
Actual lens	59	1335

Tuesday 11 January

Code to plot graph of how true positive and true negative rates change with the threshold:

```
# predictions = model.predict(x_test)
tpr = []
tnr = []
thresholds = np.linspace(0, 1, 1001)
for t in thresholds:
    threshold_predictions = predictions > t
    confusion_matrix = tf.math.confusion_matrix(
        y_test, threshold_predictions, num_classes=None, weights=None,
dtype=tf.dtypes.int32,
        name=None
    )
    tpr.append(confusion_matrix.numpy()[1, 1] / (confusion_matrix.numpy()[1, 1] +
confusion_matrix.numpy()[1, 0]))
    tnr.append(confusion_matrix.numpy()[0, 0] / (confusion_matrix.numpy()[0, 0] +
confusion_matrix.numpy()[0, 1]))
plt.figure(figsize=(8.5, 4.5))
plt.plot(thresholds, tpr, label='TPR')
plt.plot(thresholds, tnr, label='TNR')
plt.xlabel('$t$')
plt.legend()
plt.show()
```

Output:


$t$
TPR
TNR