In [1]:
```
import torch, numpy as np, pandas as pd
```

Creating a Linear Model, Neural Net and Deep Learning Model from Scratch using Tabular Da

Introduction

In this notebook, I will be demonstrating my learning in creating neural networks (NNs) from scra be building on my knowledge by going through distinct steps:

. Build **linear model** from scratch.
. Build simple **NN** from scratch.
. Build a **deep learning** (DL) model from scratch.

While a similar task was previously completed for image classification using the MNIST dataset, notebook will focus on the Titanic dataset, aiming to build a model that can predict the chance of

Data Extraction and Cleaning

The data is contained in a csv file which we can open with Pandas.

In [2]:
```
df = pd.read_csv('train.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

We can see that some of the columns contain NaN values, which we will be unable to multiply b coefficients.

Let's replace the missing values with something - normally the mode is a good place to start.

In [3]:
```
modes = df.mode().iloc[0] # we use .iloc to take first row, as it will re
df.fillna(modes, inplace=True)
```

Now we need to make sure that our data will be appropriate to feed through a model. A good pla
start is with `.describe()` to see a summary, selecting numeric columns only to start with.

In [4]:
```
df.describe(include=(np.number))
```

Out[4]:

	PassengerId	Survived	Pclass	Age
count	.	.	.	.
mean	.	.	.	.
std	.	.	.	.
min	.	.	.	.
%	.	.	.	.
%	.	.	.	.
%	.	.	.	.
max	.	.	.	.

We need to do a bit of feature engineering to make our data fit for purpose.

Fare has many values between and but some massive values too. This will skew
model, so we take the `log` to bring all values to a sensible range. This is generally a good tech
continuous variables involving *money* or *popn*.

In [5]:
```
df['LogFare'] = np.log(df['Fare'] + 1) # we add 1 to avoid log(0)!
```

Clearly, we still have some issues - notably, the strings cannot be multiplied by coefficients! Let's
them with numbers.

Pandas allows us to create new columns containing **dummmy variables** which is a column that
a `1` where a particular column contains a particular value or `0` otherwise. This is very easy in
using the `get_dummies` function - by default, this provides a n columns for n categories (even
technically we only need n- columns as you can derive the final column). However, this is us
means we do not need to worry about adding a constant term anymore as we don't need a sepa
intercept term to cover rows that aren't otherwise part of a column.

We will do this for all categorical variables, even `Pclass`, because as shown below, this has
distinct values.

In [6]:
```
# only 3 distinct values in Pclass
pclasses = sorted(df.Pclass.unique())
pclasses
```

Out[6]:
```
[1, 2, 3]
```

In [7]:
```
categorical = ['Pclass', 'Sex', 'Embarked']
df = pd.get_dummies(df, columns=categorical)
```

```python
# get_dummies will automatically remove the original colummns.
df.head()
```

Out[7]:

PassengerId	Survived	Name	Age	SibSp	Parch	Ticket	Far
		Braund, Mr. Owen Harris	.			A/	.
		Cumings, Mrs. John Bradley (Florence Briggs Th...	.			PC	.
		Heikkinen, Miss. Laina	.			STON/O .	.
		Futrelle, Mrs. Jacques Heath (Lily May Peel)	.				.
		Allen, Mr. William Henry	.				.

Now, we have engineered both the independent and dependent variables we will use in the mod will discard some of the remaining columns for the purpose of this notebook (which is focusing o creating a NN from scratch rather than effective data transformation).

However, it is worth noting that a lot can be done with the remaining columns. The best Kaggle r actually used *only* the name column to predict chance of survival!

In [8]:
```python
df.drop(columns=["Fare", "PassengerId", "Name", "Ticket", "Cabin"], inpla

df.head()
```

Out[8]:

Survived	Age	SibSp	Parch	LogFare	Pclass_	Pclass_	Pclass_	Se
	.			.	False	False	True	
	.			.	True	False	False	
	.			.	False	False	True	
	.			.	True	False	False	
	.			.	False	False	True	

The final step we want to do is **normalize** our data so that all the columns contain numbers from
`1`. We do this by diciging each column by its maximum value. This will prevent the model by be
dominated by larger values such as *age*.

It is worth noting that I am practicing using Pandas here for data manipulation, but in practice (pa
if working with more data) this would be much more efficient to perform in PyTorch, as we could
broadcasting to rapidly perform divisions. However, this allows me to have all the data enginee
together so I can focus on creating the models.

Also note, we are converting our boolean columns to floats, as this will enable matrix multiplicati
PyTorch later.

In [9]:
```python
# PyTorch expects floats to perform matrix multiplications
indep_cols = df.columns[df.columns != "Survived"]
df[indep_cols] = df[indep_cols].astype(float)
```

In [10]:
```python
for col in df.columns:
  df[col] = df[col]/df[col].max()
df.head()
```

Out[10]:

Survived	Age	SibSp	Parch	LogFare	Pclass_	Pclass_	Pclass_
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	

Now, we can start building models!

Linear Model

Complile Data

We will be using PyTorch to build our model, as PyTorch tensors can make use of the GPU to m
fast calculations.

We start by turning our independent variables (predictors) and dependent variables (targets) into

In [11]:
```python
from torch import tensor

dep = tensor(df["Survived"])
indep = tensor(df[indep_cols].values)

indep.shape, dep.shape
```

Out[11]:
```
(torch.Size([891, 12]), torch.Size([891]))
```

At this point, it is also important to split the data into **training** and **validation** sets, a topic I have
in-depth about previously.

We can use fastai's `RandomSplitter` for this task as it will return distinct random selection of
both training and validation sets.

In [12]:
```
from fastai.data.transforms import RandomSplitter
train, val = RandomSplitter(seed=42)(df)
train, val
```

Out[12]:
```
((#713)
[788,525,821,253,374,98,215,313,281,305,701,812,76,50,387,47,516,564,434,117
 (#178)
[303,778,531,385,134,476,691,443,386,128,579,65,869,359,202,187,456,880,705,
)
```

In [13]:
```
trn_dep, val_dep = dep[train], dep[val]
trn_indep, val_indep = indep[train], indep[val]

trn_indep.shape, val_indep.shape, trn_dep.shape, val_dep.shape
```

Out[13]:
```
(torch.Size([713, 12]),
 torch.Size([178, 12]),
 torch.Size([713]),
 torch.Size([178]))
```

Finally, we need to turn our dependent variable into a column vector (rank-   tensor) which we
by indexing the column dimension (which doesn't currently exist) with the special value `None` w
tells PyTorch to add a new dimension here.

We are doing this as we will be using matrix multiplication and the predictions will be returned as
rank-   column vector.

In [14]:
```
trn_dep = trn_dep[:,None]
val_dep = val_dep[:,None]
```

Initialise Coefficients

Now that we have our variables, we need to generate our (initially) random coefficients.

We need one coefficient for each independent variable and we will pick random numbers in the r
(- . , .).

When we perform matrix multiplication between the coefficients and independent variables, we r
`coeffs` to be a rank-   column vector, hence, we add an extra dimension through the secon
argument of `torch.rand()`.

In [15]:
```
def init_coeffs(n_coeff):
  return (torch.rand(n_coeff, 1, dtype=torch.float64) * 0.1).requires_gra
  # I use dtype as I need to make sure the coeffs are the same data type
```

Calculating Predictions

Our predictions will be calculated by multiplying each row by the coefficients and adding them up

The function `calc_preds` will use **broadcasting** to multiply each row of independent variable
vector of coefficients. The sum of row of independent variables will be calculated and this will rep
the prediction for these predictors.

But what about the `sigmoid` function? Well, this is a cool function that basically limits our pred
be between `0` and `1` (since `0` means died and `1` means survived). Essentially whenever w
performing binary classification, we will use the sigmoid function as it improves the accuracy of t
substantially.

In [16]:
```
def calc_preds(coeffs, indeps):
  return torch.sigmoid((indeps*coeffs).sum(axis=1))
```

This function looks great! However, we can actually improve it further!

Multiplying elements together, then adding across rows is identical to doing a matrix-vector prod
can use the Python `@` operator to perform PyTorch optimised matrix products.

In [17]:
```
def calc_preds(coeffs, indeps):
  return torch.sigmoid(indeps@coeffs)
```

Calculating Loss

Once the predictions are made (initially on random coefficients) for each row of independent vari
we need to calculate the **loss**. The loss will allow us to update the coefficients (we will now refer
as **parameters**).

`calc_loss` calls the `calc_preds` function and uses the **mean absolute error** for loss.

In [18]:
```
def calc_loss(coeffs, indeps, deps):
  return torch.abs(calc_preds(coeffs, indeps) - deps).mean()
```

Update Parameters

At this point, we have made predictions and calculated the loss. Now we need to use the loss ar
stochastic gradient descent (SGD) to update the parameters.

`update_coeffs` uses the gradient (calculated by PyTorch as we used `requires_grad_` w
initialising the paramters) of each coefficient to determine how to adjust it. It is adjusted by the **le**
rate an important hyperparameter when designing a model.

We zero the coefficients to prevent the gradients from accumulating (the default behaviour of Py

In [19]:
```
def update_coeffs(coeffs, lr):
  coeffs.sub_(coeffs.grad * lr)
  coeffs.grad.zero_()
```

Training the Model

Now we have all of the important functions to:

. Initialise Coefficients

. Calculate Predictions

. Calculate Loss

. Update Parameters

We will put this all together using functions.

The first called `epoch` represents a single **epoch** which a full pass through all training data anc our previously defined functions.

The second function called `train_model` will initialise the coefficients then call `epoch` for ea epoch we want to perform.

In [20]:
```
def epoch(coeffs, lr):
  loss = calc_loss(coeffs, trn_indep, trn_dep)
  loss.backward()
  with torch.no_grad():
    update_coeffs(coeffs, lr)
  print(f"{loss:.3f}")
```

In [21]:
```
def train_model(epochs, lr):
  torch.manual_seed(442)
  coeffs = init_coeffs(indep.shape[1])
  for i in range(epochs):
    epoch(coeffs, lr=lr)
  return coeffs
```

In [22]:
```
coeffs = train_model(15, 100)
```

```
0.515
0.323
0.288
0.204
0.200
0.198
0.197
0.197
0.196
0.196
0.196
0.195
0.195
0.195
0.195
```

Analyse Coefficients

We can write a quick function to see all of our coefficients for each independent variable. This wi an idea of how our function works.

For example, we can see that a higher *age* is a strong predictor of death and a a higher *class* (a or) is a strong predictor of life by looking at the coefficients.

In [23]:
```
def show_coeffs():
  return dict(zip(indep_cols, coeffs.requires_grad_(False)))
show_coeffs()
```

```
Out[23]: {'Age': tensor([-1.1208], dtype=torch.float64),
          'SibSp': tensor([-0.8208], dtype=torch.float64),
          'Parch': tensor([-0.3355], dtype=torch.float64),
          'LogFare': tensor([0.4991], dtype=torch.float64),
          'Pclass_1': tensor([3.3172], dtype=torch.float64),
          'Pclass_2': tensor([1.2995], dtype=torch.float64),
          'Pclass_3': tensor([-6.3529], dtype=torch.float64),
          'Sex_female': tensor([8.2192], dtype=torch.float64),
          'Sex_male': tensor([-10.0180], dtype=torch.float64),
          'Embarked_C': tensor([1.2405], dtype=torch.float64),
          'Embarked_Q': tensor([1.4383], dtype=torch.float64),
          'Embarked_S': tensor([-4.3675], dtype=torch.float64)}
```

Calculating Metrics

Now we just need a **metric** to determine the quality of the model. We can see that the loss is go
down, but loss is not suitable for evaluating how **accurate** the model is.

We will define a prediction of *death* as any value <= . and a prediction of *life* as any valu
. .

We have our coeffients but we haven't used our validation set to determine how effective the mo
We will use the validation set to calculate accuracy.

```
In [24]: def accuracy(coeffs):
           return (val_dep.bool()==(calc_preds(coeffs, val_indep)>0.5)).float().me
         accuracy(coeffs)
```

```
Out[24]: tensor(0.8258)
```

Summary

We've now built a linear model that is performing very well! This model is not yet a NN but **it is tl
for creating a layer of a NN**, which we will be working on in the next section!

In reality, beacuse this is a simple task and there is minimal data, a NN is unlikely to improve our
performance, as an accuracy of . % is very good. **However, for the purpose of learning
be beneficial to see how our linear model integrates in a full NN.**

Neural Network

Now we will be creating a NN that will have **layers**. The first layer will take the independen
variables as inputs and create `n` **activations** or outputs (after passing through an activation fu
These `n` activations will be the input for the second layer which will output exactly value,
representing our prediction of survival or not.

Initialise Coefficients

Because our NN will have layers, it will rank- tensors of coefficients. We will rede
`init_coeffs` function for the NN.

By default, our `init_coeffs` function will produce hidden units in the first layer, whic mapped to a single output in the second layer. If we increase the hidden units the network will be flexible but slower and harder to train, so this is an important hyperparameter.

`layer1` will be a x tensor and when we matrix multiply our data by layer v output x outputs. We divide each value by the number of hidden units as we want of the coefficients to be inversely proportional to the number of hidden units. This prevents the w from growing out of control.

`layer2` will be a x tensor and we matrix multiply the activations from `layer1` output a single prediction.

`const` is the bias for the final output layer. layer has 'bias' already factored into the extra independent variables (discussed above).

Activations are the final output from layer after passing through the activation function.

In [25]:
```
def init_coeffs(n_coeff, n_hidden=20):
  layer1 = (torch.rand(n_coeff, n_hidden, dtype=torch.float64) - 0.5)/n_h
  layer2 = torch.rand(n_hidden, 1, dtype=torch.float64) - 0.2
  const = torch.rand(1, dtype=torch.float64)[0]
  return layer1.requires_grad_(),layer2.requires_grad_(),const.requires_g
```

Calculate Predictions

We import `torch.nn.functional` so we can access the `relu` function from PyTorch.

Now we can really see the NN. Our `coeffs` are unpacked into their relevant variables.

We update `res` at each layer, initially multiplying our independent variables by layer , we th employ the **activation function**, ReLU.

Then, the second layer takes `res` and multiplies by layer adding the constant term. This our hidden activations to a single prediction which we return after employing the `sigmo` function (to map it to a value between and).

In [26]:
```
import torch.nn.functional as F

def calc_preds(coeffs, indeps):
  l1, l2, const = coeffs
  res = F.relu(indeps@l1)
  res = res@l2 + const
  return torch.sigmoid(res)
```

Update Coefficients

Once we have our prediction, we can update our coefficients. However, we need to update the coefficients in every layer, so we use a for loop.

In [27]:
```
def update_coeffs(coeffs, lr):
  for layer in coeffs:
    layer.sub_(layer.grad * lr)
    layer.grad.zero_()
```

Train Model

Finally, the model can be trained, using many of the same functions used in the linear model. Th
`calc_loss` function remains the same as does the `epoch` and `train_model` functions.

In [28]:
```
coeffs = train_model(30, 20)
```

```
0.545
0.419
0.235
0.213
0.207
0.208
0.207
0.216
0.203
0.201
0.199
0.196
0.194
0.194
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
```

In [29]:
```
accuracy(coeffs)
```

Out[29]:
```
tensor(0.8258)
```

Summary

We see minimal improvemeny by employing a NN in this situation; however, we can easily see h
linear model easily translates to a NN simply by adding layers and activation funcitons.

Deep Learning

Initialise Coefficients

Because the NN above only uses a single hidden layer, it does not count as "deep learning". Ho
we can easily extend the logic form above to add an arbitrary number of layers.

As we can see, `sizes` starts with a dimension of n_coeff, representing the number of coefficie
need for each independent variable. It takes our defined `hidden` layer dimensions and maps t
single output (our prediction).

Next, we define our `layers` and `consts`.

To help demonstrate my understanding we can run through what `layers` looks like. It is a list
tensors. The first tensor has random values and shape n_coeff x . The second tensor is
as it takes the outputs from layer and maps it to outputs. The third
x as it takes the outputs from layer and maps it to a single prediction.

`consts` consists of tensors too, all with random constant term.

In [30]:
```python
def init_coeffs(n_coeff):
  hiddens = [10, 10] # this will represent the size of each hidden layer
  sizes = [n_coeff] + hiddens + [1]
  n = len(sizes)

  layers = [(torch.rand(sizes[i], sizes[i + 1], dtype=torch.float64) - 0.
  consts = [(torch.rand(1, dtype=torch.float64)[0] - 0.5) * 0.1 for i in

  for l in layers + consts:
    l.requires_grad_()

  return layers, consts
```

Calculate Predictions

Now we can alter our `calc_preds` function to be suitable for deep learning.

The main difference is we loop through each layer performing the matrix multiplication and activa
key to notice that we use the ReLU function after each linear transformation *unless* we are at the
layer, in which case we use the sigmoid function to ensure our final prediction falls between
.

In [31]:
```python
def calc_preds(coeffs, indeps):
  layers, consts = coeffs

  n = len(layers)

  res = indeps

  for i, l in enumerate(layers):
    res = res@l + consts[i]
    if i != n - 1:
      res = F.relu(res)

  return torch.sigmoid(res)
```

Update Coefficients

Again, only a minor change is required here, ensuring we perform SGD on each parameter (both
and bias).

In [32]:
```
def update_coeffs(coeffs, lr):
  layers, consts = coeffs
  for l in layers + consts:
    l.sub_(l.grad * lr)
    l.grad.zero_()
```

Train Model

Finally, we can train our model and check its accuracy!

In [33]:
```
coeffs = train_model(20, 4)
```

```
0.554
0.484
0.407
0.345
0.314
0.293
0.212
0.201
0.218
0.213
0.194
0.194
0.193
0.193
0.193
0.193
0.193
0.193
0.193
0.193
```

In [34]:
```
accuracy(coeffs)
```

Out[34]:
```
tensor(0.8258)
```

Conclusion

In this notebook, we have seen the gradual build of a deep learning neural network from a primit
model. Other than understanding how neural networks really work (which is pretty cool!) there's
other takeaways from this notebook:

. **Sometimes simple solutions work**. We actually saw no improvement from our initial linea
While it was great to demonstrate my understanding of NNs, they're not always necessary
depending on the data available.
. **There's no need to build NNs from scractch**. By using pretrained models and curated
architectures we can get better results much easier. Throughout this notebook, particularly
initialising random coefficients, we had to multiply or divide our paramters by arbitrary numb
is not ideal, and their are more evidenced-based approaches to selecting appropriate initial
parameters, that other architectures have researched and put in place.

However, overall, this notebook did not intend to produce the best results but to demonstrate hov
are built and how DL is conducted on a basic, yet low-level. As such, I feel I have a deep unders
of how NNs work.