Solutions 1

Jumping Rivers

Building a first model

The **jrpytensorflow** package has some concentric circle data which can be loaded with

```
import jrpytensorflow

X, y = jrpytensorflow.datasets.load_circles()
```

- Create an exploratory visualisation of the data

```
import matplotlib.pyplot as plt
plt.figure()
plt.scatter(X[:,0],X[:,1],c=y)
plt.show()
```

- Create a logistic regression model for this problem

```
import tensorflow as tf
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(2, activation='sigmoid'),
    tf.keras.layers.Dense(1,  activation='sigmoid')
])
```

- Compile and run a simple training routine to fit this model.

```
model.compile(optimizer = 'sgd',
     loss = 'binary_crossentropy')
history = model.fit(X, y, epochs =200)
```

- How many predictions did you get correct?

```
sum((model(X).numpy().ravel() > 0.5) == y)
```

- Assuming that your model object is called `model`, you can visualise the predicted probability space with the following code.

```
import numpy as np
import matplotlib.pyplot as plt
x1 = np.linspace(-1.5,1.5,100)
grid = np.array([(x,y) for x in x1 for y in x1])

output = np.array(model(grid))
plt.figure()
plt.pcolormesh(x1, x1,output.reshape(100,100))
```

```
plt.scatter(X[:,0],X[:,1],c=y, edgecolor = 'black')
plt.xlim([-1.5,1.5])
plt.ylim([-1.5,1.5])
plt.show()
```

- It is very hard to classify this dataset with logistic regression using only the input variables provided. Can you improve performance by introducing some additional features.

```
X_new = np.hstack([X,X*X])

model2 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(4, activation='sigmoid'),
    tf.keras.layers.Dense(1,  activation='sigmoid')
])

model2.compile(optimizer = 'sgd',
     loss = 'binary_crossentropy')
history = model2.fit(X_new, y, epochs =200)

x1 = np.linspace(-1.5,1.5,100)
grid = np.array([(x,y) for x in x1 for y in x1])
grid = np.hstack([grid,grid*grid])

output = np.array(model2(grid))
plt.figure()
plt.pcolormesh(x1, x1,output.reshape(100,100))
plt.scatter(X[:,0],X[:,1],c=y, edgecolor = 'black')
plt.xlim([-1.5,1.5])
plt.ylim([-1.5,1.5])
plt.show()

sum((model2(X_new).numpy()[:,0] > 0.5) == y)
```

Optional

- Create a flexible logistic regression class that could be integrated with a `sklearn.Pipeline` for predicting a binary output

```
import inspect
from sklearn.metrics import accuracy_score

def regModel(inputDim, outputDim):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(inputDim,
                              activation='sigmoid'),
```

```
        tf.keras.layers.Dense(outputDim,
                              activation='sigmoid')
    ])
    return model

class tfRegression:
    def __init__(self, inputDim = 2,
                 outputDim = 1, numEpochs = 200):
        self.model = None

        # this allows us to access what is in the above
        # brackets through self.
        args, _, _, values = inspect.getargvalues(
          inspect.currentframe()
        )
        values.pop('self')
        for arg, val in values.items():
          setattr(self,arg,val)

    def _build_model(self):
        self.model = regModel(
            self.inputDim,
            self.outputDim
            )

    def _train_model(self, X, y):
        self.model.compile(optimizer = 'sgd',
             loss = 'binary_crossentropy')
        history = self.model.fit(X, y,
                              epochs = self.numEpochs)

    def fit(self,X, y):
        self._build_model()
        self._train_model(X, y)
        return self

    def predict(self,X,y = None):
        return self.model(X).numpy().ravel()

    def score(self,X,y,sample_weight = None):
        y_pred = self.predict(X)
        return accuracy_score(y, (y_pred > 0.5).astype(int))

from sklearn.pipeline import Pipeline
```

```
pipe = Pipeline([
  ('reg', tfRegression(2,1, 200))
])
```