Solutions 2

Jumping Rivers

Concentric circles: revisited

- Fit a multi layer percepton model to the concentric circles data from earlier. You should be able to get pretty good performance without having to do any manual feature engineering.

```
import jrpytensorflow
import tensorflow as tf
from tensorflow import keras

X, y = jrpytensorflow.datasets.load_circles()

def circleModel(opt, loss):
  model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(50, input_shape=(2,),
                                  activation='relu'),
    tf.keras.layers.Dense(25, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
  ])

  model.compile(optimizer=opt,
              loss=loss,
              metrics=['accuracy'])

  return model

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

It can be difficult to get an idea of what your network predictions look like, especially in high dimensional spaces. Fortunately visualisation in two dimension is trivial. One of the advantages of being able to watch your prediction space update in real time is the ability to gain intuition on how changes to the model structure or additional hyper-parameters affects the model fitting process and final output. To that end the **jrpytensorflow** package has a `vis_binary_classification` classification function. This function is really only intended for visualisation with a 2 dimensional input space mapping to a binary label as visualisation of larger dimensions becomes tricky.

- Try fitting your model to the concentric circles using the visualiser, for example

```
import jrpytensorflow
X,y = jrpytensorflow.datasets.load_circles()
model = myModel('sgd', 'binary_crossentropy')
jrpytensorflow.vis_binary_classification(model, X, y, epochs=200)
```

If you are using a Jupyter notebook, you may need to use matplotlib magic to properly register the backend to view the plots updating as we train. To do so, add

```
%matplotlib notebook
```

at the *top* of your notebook along with your matplotlib import.

- Explore changing your model architecture and the optimiser hyperparameters, how do they affect fit and convergence.

Boston housing

The code below will set you up with a training and test set for the boston housing data.

```
from sklearn.datasets import load_boston
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

boston = load_boston()
X, y = boston.data, boston.target.reshape(-1,1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

- Try fitting a multi layered perceptron model to the boston housing example, how does changing the number and size of layers affect the ability to find a good model.
- Investigate whether different activation functions have much bearing on the outcome
- Try training a model, saving it, and loading it back using one of the methods discussed in the notes.

```
import pandas as pd

def bostonModel(opt, loss):
  model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(10, input_shape=(13,),
                                  activation='tanh'),
```

```
    tf.keras.layers.Dense(1, activation='softplus')
  ])

  model.compile(optimizer=opt,
                loss=loss)

  return model

def trainer(model, X_train,X_test,y_train,y_test, epochs):
    history = model.fit(X_train, y_train, epochs = epochs,
                        validation_data=[X_test,y_test],
                        verbose = 0)
    return pd.Series(history.history['loss'])

model = bostonModel('sgd','mse')
losses = trainer(model, X_train,X_test,y_train,y_test, 50)

losses.plot()
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```