Exercise 5: Practice with MLP Network (3)

Assignment 4 Author: David Nogales Pérez

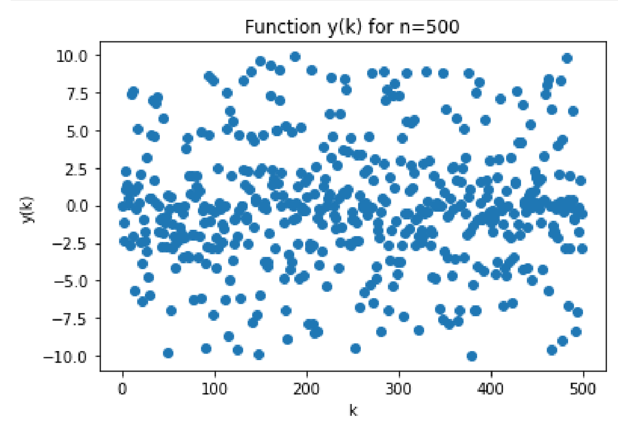
In [ ]:
```python
import numpy as np
import pandas as pd
import seaborn as sns
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.activations import sigmoid
from tensorflow.keras.losses import MeanSquaredError

from sklearn.model_selection import RepeatedKFold, cross_val_score
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor
from tensorflow.keras.metrics import RootMeanSquaredError
from keras import losses
from keras import regularizers
import keras
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

Generate training data

In [ ]:
```python
def y1(k,y,u):
  PI = np.pi
  e = np.e
  u_k = u[k]
  y_k= y[k]
  exp = -(u_k**2)-(y_k**2)
  return 2.5*y_k*np.sin(PI*np.power(e,exp))
def y2(k,y,u):
  u_k=u[k]
  return u_k*(1+u_k**2)

#Generates n samples for the function y(k) described in the problem statement
def generate_data(n=10,seed=12345):
  data = np.zeros(n)
  rng = np.random.default_rng(seed)
  u = rng.uniform(-2,2,n)
  for i in range(1,n):
    yk= y1(i-1,data,u)+y2(i-1,data,u)
    data[i]=yk
  return np.arange(n),data

n = 500 #training size
X_train,y_train = generate_data(n)
X_train.shape,y_train.shape
```

Out[ ]:
```
((500,), (500,))
```

In [ ]:
```python
plt.scatter(X_train,y_train)
plt.title(f'Function y(k) for n={n}')
plt.ylabel("y(k)")
plt.xlabel("k")
plt.show()
```



Plot of 500 samples of the function:

$$y(k) = y_1(k-1) + y_2(k-1)$$

where

$$y_1(k) = 2.5y(k)sin(\pi\epsilon^{-u^2(k)-y^2(k)})$$
$$y_2(k) = u(k)(1+u^2(k))$$

Build MLP

In [ ]:
```python
#Generates the loss plot for the given history generated by the model
def plot_loss(history):
    plt.plot(history.history['loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()

#Generates a scatter plot of the input data and the prediction of the given model
def plot_predicted(model,X,y):
  plt.scatter(X,y,label="real")
  plt.scatter(X,model.predict(X),label="predicted")
  plt.legend(loc="upper right")
  plt.xlabel("k")
  plt.ylabel("Value")


#Returns a function which generates a 3-hidden layer MLP with the given regularization and activation function
def buildmodel(act='sigmoid',reg = regularizers.l1_l2()):
    def build():
      model = Sequential()
      model.add(Dense(1, input_dim=1, activation=act, kernel_regularizer=reg))
      model.add(Dense(50, activation=act, kernel_regularizer=reg))
      model.add(Dense(30, activation=act, kernel_regularizer=reg))
      model.add(Dense(15, activation=act, kernel_regularizer=reg))
      model.add(Dense(1,activation = 'linear'))
      model.compile(optimizer="Adam", loss=losses.mean_squared_error,metrics=[RootMeanSquaredError()])
      return model
    return build
```

Applying cross validation to the MLP

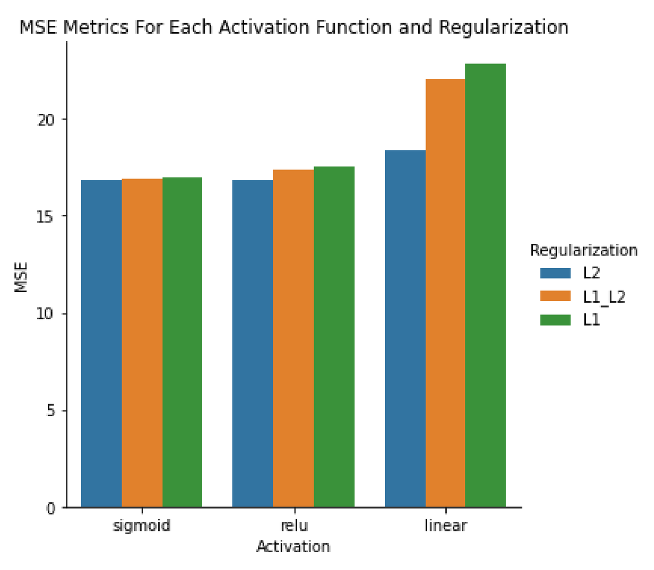
In [ ]:
```python
parameters = {"activation":["sigmoid","linear","relu"],
              "regularizer":[("L1",regularizers.l1()),("L2",regularizers.l2()),("L1_L2",regularizers.l1_l2())]
              }

results = {
}
for act in parameters["activation"]:
  results[act]={}
  for reg in parameters["regularizer"]:
    build = buildmodel(act,reg[1])
    estimator= KerasRegressor(build_fn=build, epochs=50, batch_size=25, verbose=0)
    kfold= RepeatedKFold(n_splits=5, n_repeats=5)
    result = cross_val_score(estimator, X_train, y_train, cv=kfold, n_jobs=1)
    mean = result.mean()
    print(f"Cross validation mean of MSE: {mean}")
    results[act][reg[0]] = mean
```

```
Cross validation mean of MSE: -16.952189750671387
Cross validation mean of MSE: -16.827399063110352
Cross validation mean of MSE: -16.93697811126709
Cross validation mean of MSE: -22.822959213256837
Cross validation mean of MSE: -18.354305610656738
Cross validation mean of MSE: -22.08200454711914
Cross validation mean of MSE: -17.559741706848143
Cross validation mean of MSE: -16.844078407287597
Cross validation mean of MSE: -17.384937133789062
```

In [ ]:
```python
df = pd.DataFrame(results)
df = df.rename_axis('Regularization').reset_index()
df = pd.melt(df,id_vars=["Regularization"],value_vars=['sigmoid',"linear","relu"],var_name='Activation',value_name='MSE').sort_values('MSE',ascending=False).reset_index(drop=True)
df['MSE']=abs(df['MSE'])
ax =  sns.catplot(data=df,x='Activation',y='MSE',hue='Regularization',kind='bar').set(title = 'MSE Metrics For Each Activation Function and Regularization')
df
```

Out[ ]:

	Regularization	Activation	MSE
0	L2	sigmoid	16.827399
1	L2	relu	16.844078
2	L1_L2	sigmoid	16.936978
3	L1	sigmoid	16.952190
4	L1_L2	relu	17.384937
5	L1	relu	17.559742
6	L2	linear	18.354306
7	L1_L2	linear	22.082005
8	L1	linear	22.822959

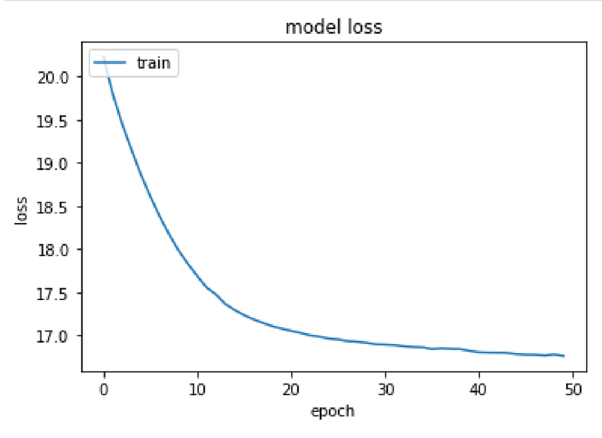


As seen in the previous graphic the best results are obtained using a Sigmoid activation function with any kind of regularization but L1+L2 regularization will be used from now on.

In [ ]:
```python
model = buildmodel()()
history = model.fit(X_train, y_train, epochs=50, batch_size=25, verbose=1)
```

```
Epoch 1/50
20/20 [==============================] - 1s 2ms/step - loss: 20.2313 - root_mean_squared_error: 4.0947
Epoch 2/50
20/20 [==============================] - 0s 2ms/step - loss: 19.7954 - root_mean_squared_error: 4.0811
Epoch 3/50
20/20 [==============================] - 0s 2ms/step - loss: 19.4505 - root_mean_squared_error: 4.0776
Epoch 4/50
20/20 [==============================] - 0s 2ms/step - loss: 19.1481 - root_mean_squared_error: 4.0771
Epoch 5/50
20/20 [==============================] - 0s 2ms/step - loss: 18.8636 - root_mean_squared_error: 4.0764
Epoch 6/50
20/20 [==============================] - 0s 2ms/step - loss: 18.6112 - root_mean_squared_error: 4.0767
Epoch 7/50
20/20 [==============================] - 0s 2ms/step - loss: 18.3772 - root_mean_squared_error: 4.0761
Epoch 8/50
20/20 [==============================] - 0s 2ms/step - loss: 18.1737 - root_mean_squared_error: 4.0767
Epoch 9/50
20/20 [==============================] - 0s 2ms/step - loss: 17.9868 - root_mean_squared_error: 4.0764
Epoch 10/50
20/20 [==============================] - 0s 2ms/step - loss: 17.8301 - root_mean_squared_error: 4.0770
Epoch 11/50
20/20 [==============================] - 0s 2ms/step - loss: 17.6890 - root_mean_squared_error: 4.0775
Epoch 12/50
20/20 [==============================] - 0s 2ms/step - loss: 17.5566 - root_mean_squared_error: 4.0767
Epoch 13/50
20/20 [==============================] - 0s 2ms/step - loss: 17.4756 - root_mean_squared_error: 4.0788
Epoch 14/50
20/20 [==============================] - 0s 2ms/step - loss: 17.3640 - root_mean_squared_error: 4.0760
Epoch 15/50
20/20 [==============================] - 0s 2ms/step - loss: 17.2929 - root_mean_squared_error: 4.0763
Epoch 16/50
20/20 [==============================] - 0s 2ms/step - loss: 17.2348 - root_mean_squared_error: 4.0762
Epoch 17/50
20/20 [==============================] - 0s 2ms/step - loss: 17.1856 - root_mean_squared_error: 4.0760
Epoch 18/50
20/20 [==============================] - 0s 2ms/step - loss: 17.1455 - root_mean_squared_error: 4.0764
Epoch 19/50
20/20 [==============================] - 0s 2ms/step - loss: 17.1065 - root_mean_squared_error: 4.0760
Epoch 20/50
20/20 [==============================] - 0s 2ms/step - loss: 17.0780 - root_mean_squared_error: 4.0764
Epoch 21/50
20/20 [==============================] - 0s 2ms/step - loss: 17.0525 - root_mean_squared_error: 4.0764
Epoch 22/50
20/20 [==============================] - 0s 2ms/step - loss: 17.0300 - root_mean_squared_error: 4.0766
Epoch 23/50
20/20 [==============================] - 0s 2ms/step - loss: 17.0010 - root_mean_squared_error: 4.0757
Epoch 24/50
20/20 [==============================] - 0s 2ms/step - loss: 16.9856 - root_mean_squared_error: 4.0764
Epoch 25/50
20/20 [==============================] - 0s 2ms/step - loss: 16.9640 - root_mean_squared_error: 4.0758
Epoch 26/50
20/20 [==============================] - 0s 2ms/step - loss: 16.9537 - root_mean_squared_error: 4.0763
Epoch 27/50
20/20 [==============================] - 0s 2ms/step - loss: 16.9360 - root_mean_squared_error: 4.0759
Epoch 28/50
20/20 [==============================] - 0s 2ms/step - loss: 16.9286 - root_mean_squared_error: 4.0766
Epoch 29/50
20/20 [==============================] - 0s 2ms/step - loss: 16.9162 - root_mean_squared_error: 4.0764
Epoch 30/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8990 - root_mean_squared_error: 4.0758
Epoch 31/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8949 - root_mean_squared_error: 4.0764
Epoch 32/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8891 - root_mean_squared_error: 4.0769
Epoch 33/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8755 - root_mean_squared_error: 4.0765
Epoch 34/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8672 - root_mean_squared_error: 4.0766
Epoch 35/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8631 - root_mean_squared_error: 4.0771
Epoch 36/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8439 - root_mean_squared_error: 4.0759
Epoch 37/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8511 - root_mean_squared_error: 4.0773
Epoch 38/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8461 - root_mean_squared_error: 4.0775
Epoch 39/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8436 - root_mean_squared_error: 4.0778
Epoch 40/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8231 - root_mean_squared_error: 4.0766
Epoch 41/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8064 - root_mean_squared_error: 4.0757
Epoch 42/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8026 - root_mean_squared_error: 4.0761
Epoch 43/50
20/20 [==============================] - 0s 2ms/step - loss: 16.8018 - root_mean_squared_error: 4.0766
Epoch 44/50
20/20 [==============================] - 0s 2ms/step - loss: 16.7991 - root_mean_squared_error: 4.0767
Epoch 45/50
20/20 [==============================] - 0s 2ms/step - loss: 16.7844 - root_mean_squared_error: 4.0758
Epoch 46/50
20/20 [==============================] - 0s 2ms/step - loss: 16.7790 - root_mean_squared_error: 4.0759
Epoch 47/50
20/20 [==============================] - 0s 2ms/step - loss: 16.7776 - root_mean_squared_error: 4.0763
Epoch 48/50
20/20 [==============================] - 0s 2ms/step - loss: 16.7693 - root_mean_squared_error: 4.0759
Epoch 49/50
20/20 [==============================] - 0s 2ms/step - loss: 16.7808 - root_mean_squared_error: 4.0776
Epoch 50/50
20/20 [==============================] - 0s 2ms/step - loss: 16.7632 - root_mean_squared_error: 4.0762
```

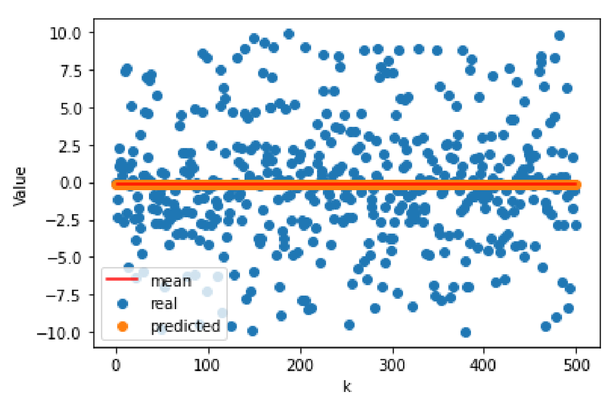
In [ ]:
```python
plot_loss(history)
```



Observing the loss curve we can see that the NN is starting to overfit slightly so no extra epoch are required to fully train the NN.

In [ ]:
```python
plot_predicted(model,X_train,y_train)
plt.plot(np.arange(0,n)+1,np.zeros(n)+y_train.mean(),'-r',label="mean")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f4777299e50>
```



In [ ]:
```python
y_train.mean()
```

Out[ ]:
```
-0.10853680066900842
```

In this plot we can see how the MLP predicts a steady line avoiding the noise introduced by the randomness of *u(k)* in the system. Which is really close to the mean of the image values of the function.