Exercise 6.2

Interpolation

In this task, we implement a simple NN to learn a complicated function.

```
import numpy as np
from tensorflow import keras
import matplotlib.pyplot as plt

layers = keras.layers
```

Generation of data

```
def some_complicated_function(x):
    return (
        (np.abs(x)) ** 0.5
        + 0.1 * x
        + 0.01 * x ** 2
        + 1
        - np.sin(x)
        + 0.5 * np.exp(x / 10.0)
        ) / (0.5 + np.abs(np.cos(x)))
```

Let's simulate the train data

```
N_train = 10 ** 4  # number of training samples
# Note: "[:, np.newaxis]" reshapes array to (N,1) as required by our DNN (we input one f
xtrain = np.random.uniform(-10, 10, N_train)[:, np.newaxis]
ytrain = some_complicated_function(xtrain) + np.random.standard_normal(xtrain.shape)  #
```

```
print("xtrain.shape", xtrain.shape)
print("ytrain.shape", ytrain.shape)
```

```
xtrain.shape (10000, 1)
ytrain.shape (10000, 1)
```

Simulate test data

```
N_test = 10000  # number of testing samples
xtest = np.linspace(-10, 10, N_test)
ytest = some_complicated_function(xtest)
```

```
print("xtest.shape", xtest.shape)
print("ytest.shape", ytest.shape)
```

```
xtest.shape (10000,)
ytest.shape (10000,)
```

Define Model

Define the number of nodes, the number of layers, and choose an activation function. Use `keras.regularizers` to use parameter norm penalties or add a dropout layer via `layers.Dropout(fraction)`.

You may use the skeleton below:

```
nb_nodes = 150
nb_layers = 5
activation = "relu"

model = keras.models.Sequential(name="1Dfit")
regularizer=keras.regularizers.L2(l2=0.003)

model.add(layers.Dense(nb_nodes, activation=activation, input_dim=xtrain.shape[1])) #F
for _ in range(nb_layers):
  model.add(layers.Dense(nb_nodes, activation=activation, kernel_regularizer=regulariz

  model.add(layers.Dropout(0.01))

model.add(layers.Dense(1))  # final layer

print(model.summary())
```

Model: "1Dfit"

Layer (type)	Output Shape	Par
dense_16 (Dense)	(None, 150)	
dense_17 (Dense)	(None, 150)	22
dropout_12 (Dropout)	(None, 150)	
dense_18 (Dense)	(None, 150)	22
dropout_13 (Dropout)	(None, 150)	
dense_19 (Dense)	(None, 150)	22
dropout_14 (Dropout)	(None, 150)	
dense_20 (Dense)	(None, 150)	22
dropout_15 (Dropout)	(None, 150)	
dense_21 (Dense)	(None, 150)	22

dropout_16 (Dropout)	(None, 150)	
dense_22 (Dense)	(None, 1)	

```
 Total params: 113,701 (444.14 KB)
 Trainable params: 113,701 (444.14 KB)
 Non-trainable params: 0 (0.00 B)
None
```

Compile the model (set an objective and choose an optimizer)

Choose an optimizer from `keras.optimizers`, e.g., `adam = keras.optimizers.Adam(learning_rate=0.001)`.

Further, choose the correct objective (loss) for this **regression task**.

```
abe = keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss="mse", optimizer=abe)
```

Train the model

Train the network for a couple of epochs and save the model several times in between.

```
epochs = 100
save_period = 10   # after how many epochs the model should be saved?

chkpnt_saver = keras.callbacks.ModelCheckpoint("weights-{epoch:02d}.weights.h5", save_

results = model.fit(
    xtrain,
    ytrain,
    batch_size=64,
    epochs=epochs,
    verbose=1,
    callbacks=[chkpnt_saver]
    )
```

```
Epoch 1/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 2.5342
Epoch 2/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 2.3818
Epoch 3/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 2.2395
Epoch 4/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 2.0884
Epoch 5/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - loss: 1.9635
Epoch 6/100
```

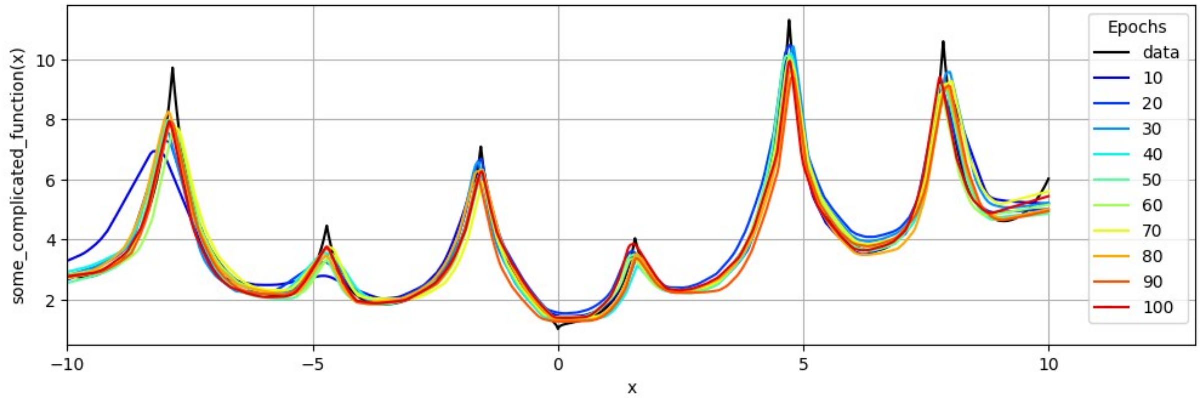
```
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.8646
Epoch 7/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.8113
Epoch 8/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.9227
Epoch 9/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.6126
Epoch 10/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.7196
Epoch 11/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.7697
Epoch 12/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.5807
Epoch 13/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.6415
Epoch 14/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.6836
Epoch 15/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.5595
Epoch 16/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - loss: 1.5804
Epoch 17/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.5400
Epoch 18/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 1.4982
Epoch 19/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.4839
Epoch 20/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.4597
Epoch 21/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.4425
Epoch 22/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.5050
Epoch 23/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.5120
Epoch 24/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.4765
Epoch 25/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.3900
Epoch 26/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 1.4986
Epoch 27/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.4460
Epoch 28/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 1.4472
Epoch 29/100
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 1.3696
```

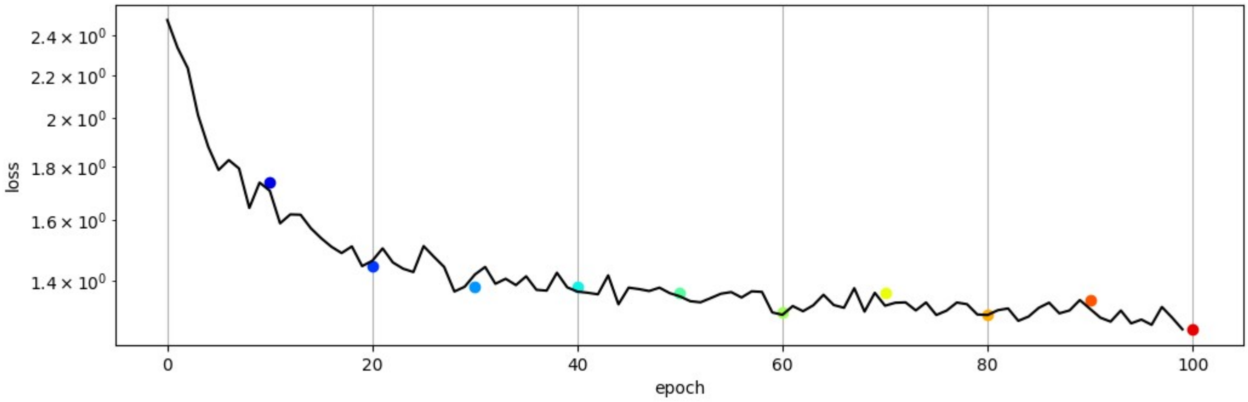
Compare the performance of the model during the training. You may use the skeleton below:

```
fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(12, 8))

ax1.plot(xtest, ytest, color="black", label="data")
saved_epochs = range(save_period, epochs + 1, save_period)

colors = [plt.cm.jet((i + 1) / float(len(saved_epochs) + 1)) for i in range(len(saved_
```

```
for i, epoch in enumerate(saved_epochs):
    model.load_weights("weights-{epoch:02d}.weights.h5".format(epoch=epoch))
    ypredict = model.predict(xtest).squeeze()
    ax1.plot(xtest.squeeze(), ypredict, color=colors[i], label=epoch)
    ax2.plot(epoch, results.history["loss"][epoch - 1], color=colors[i], marker="o")

ax1.set(xlabel="x", ylabel="some_complicated_function(x)", xlim=(-10, 13), title="")
ax1.grid(True)
ax1.legend(loc="upper right", title="Epochs")

ax2.plot(results.history["loss"], color="black")
ax2.set(xlabel="epoch", ylabel="loss")
ax2.grid(True)
ax2.semilogy()

plt.show()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
```





Note

Does not hit the very top of all peaks but that is fine. Loss decreases with more epochs but flattens out. There is still some training that can be done berfore overfit, but not much.