$Computational\,Intelligence\,Lab$

$Assignment\,1$

Karolina Kotlowska, lab czw. 9:30

1.1 Function

Consider the following Python class.

In [1]:
```
import numpy as np

class Function:
  def __init__(self,n_h,activation=lambda x : x):
    self.f=activation
    self.W0=np.random.randn(n_h,1)*np.sqrt(1/n_h)
    self.b0=np.zeros((n_h,1))
    self.W1=np.random.randn(1,n_h)*np.sqrt(1/n_h)
    self.b1=np.zeros((1,1))

  def __call__(self,x):
    z=self.W0*x+self.b0
    a  = self.f(z)
    y=np.dot(self.W1,a)+self.b1
    return y[0]

x=np.linspace(0,10,100)
f=Function(4)
y=f(x)
```

In [2]:
```
y
```

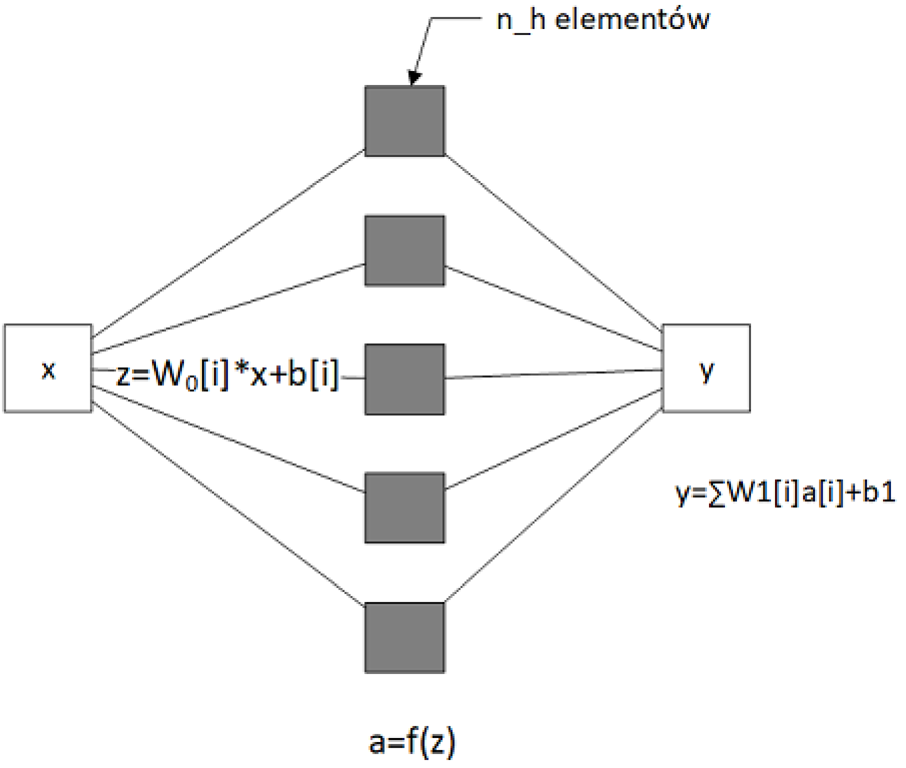
```
Out[2]: array([0.        , 0.0303176 , 0.0606352 , 0.0909528 , 0.1212704 ,
               0.151588  , 0.18190559, 0.21222319, 0.24254079, 0.27285839,
               0.30317599, 0.33349359, 0.36381119, 0.39412879, 0.42444639,
               0.45476399, 0.48508158, 0.51539918, 0.54571678, 0.57603438,
               0.60635198, 0.63666958, 0.66698718, 0.69730478, 0.72762238,
               0.75793998, 0.78825757, 0.81857517, 0.84889277, 0.87921037,
               0.90952797, 0.93984557, 0.97016317, 1.00048077, 1.03079837,
               1.06111597, 1.09143356, 1.12175116, 1.15206876, 1.18238636,
               1.21270396, 1.24302156, 1.27333916, 1.30365676, 1.33397436,
               1.36429196, 1.39460955, 1.42492715, 1.45524475, 1.48556235,
               1.51587995, 1.54619755, 1.57651515, 1.60683275, 1.63715035,
               1.66746795, 1.69778555, 1.72810314, 1.75842074, 1.78873834,
               1.81905594, 1.84937354, 1.87969114, 1.91000874, 1.94032634,
               1.97064394, 2.00096154, 2.03127913, 2.06159673, 2.09191433,
               2.12223193, 2.15254953, 2.18286713, 2.21318473, 2.24350233,
               2.27381993, 2.30413753, 2.33445512, 2.36477272, 2.39509032,
               2.42540792, 2.45572552, 2.48604312, 2.51636072, 2.54667832,
               2.57699592, 2.60731352, 2.63763111, 2.66794871, 2.69826631,
               2.72858391, 2.75890151, 2.78921911, 2.81953671, 2.84985431,
               2.88017191, 2.91048951, 2.9408071 , 2.9711247 , 3.0014423 ])
```

```
In [3]: x
```

```
Out[3]: array([ 0.        ,  0.1010101 ,  0.2020202 ,  0.3030303 ,  0.4040404 ,
                0.50505051,  0.60606061,  0.70707071,  0.80808081,  0.90909091,
                1.01010101,  1.11111111,  1.21212121,  1.31313131,  1.41414141,
                1.51515152,  1.61616162,  1.71717172,  1.81818182,  1.91919192,
                2.02020202,  2.12121212,  2.22222222,  2.32323232,  2.42424242,
                2.52525253,  2.62626263,  2.72727273,  2.82828283,  2.92929293,
                3.03030303,  3.13131313,  3.23232323,  3.33333333,  3.43434343,
                3.53535354,  3.63636364,  3.73737374,  3.83838384,  3.93939394,
                4.04040404,  4.14141414,  4.24242424,  4.34343434,  4.44444444,
                4.54545455,  4.64646465,  4.74747475,  4.84848485,  4.94949495,
                5.05050505,  5.15151515,  5.25252525,  5.35353535,  5.45454545,
                5.55555556,  5.65656566,  5.75757576,  5.85858586,  5.95959596,
                6.06060606,  6.16161616,  6.26262626,  6.36363636,  6.46464646,
                6.56565657,  6.66666667,  6.76767677,  6.86868687,  6.96969697,
                7.07070707,  7.17171717,  7.27272727,  7.37373737,  7.47474747,
                7.57575758,  7.67676768,  7.77777778,  7.87878788,  7.97979798,
                8.08080808,  8.18181818,  8.28282828,  8.38383838,  8.48484848,
                8.58585859,  8.68686869,  8.78787879,  8.88888889,  8.98989899,
                9.09090909,  9.19191919,  9.29292929,  9.39393939,  9.49494949,
                9.5959596 ,  9.6969697 ,  9.7979798 ,  9.8989899 , 10.        ])
```

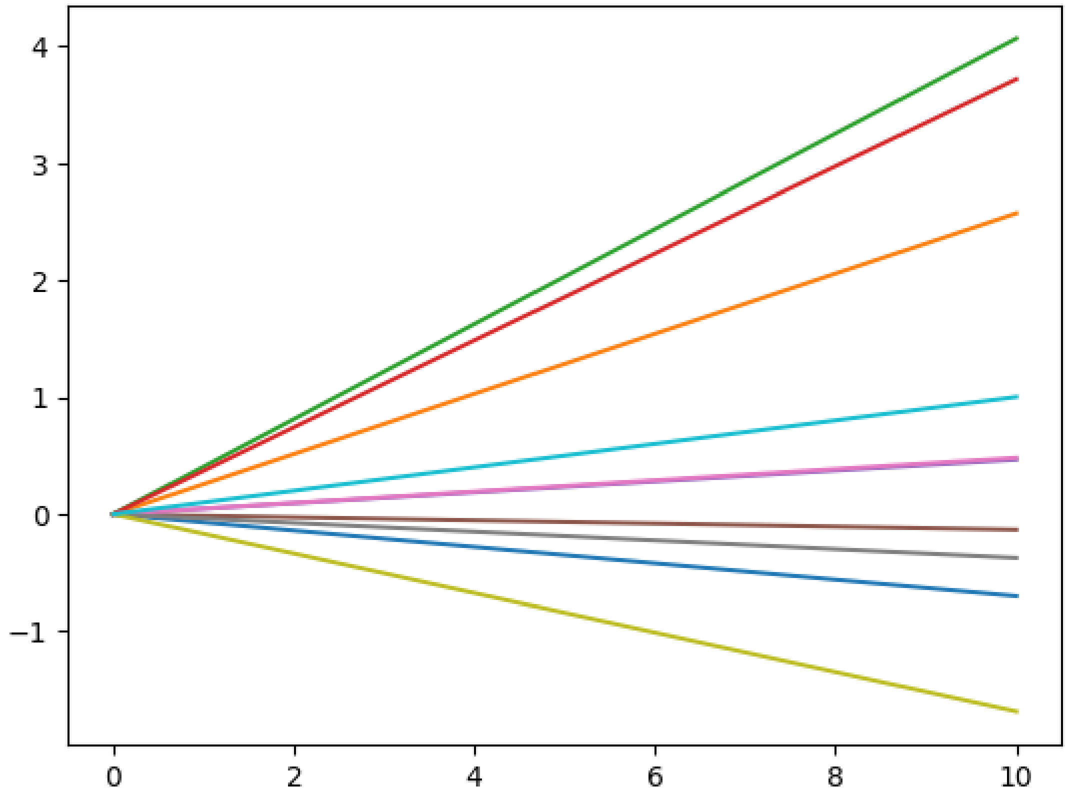
Operations placed in the function call operator can be expressed as:

- $z = W_0 \cdot x + b_0$
- $a = f(z)$
- $y = W_1 * a + b_1$



TODO 1.1.1 Create function objects for various values of `n_h` and display their shapes. Use a for loop

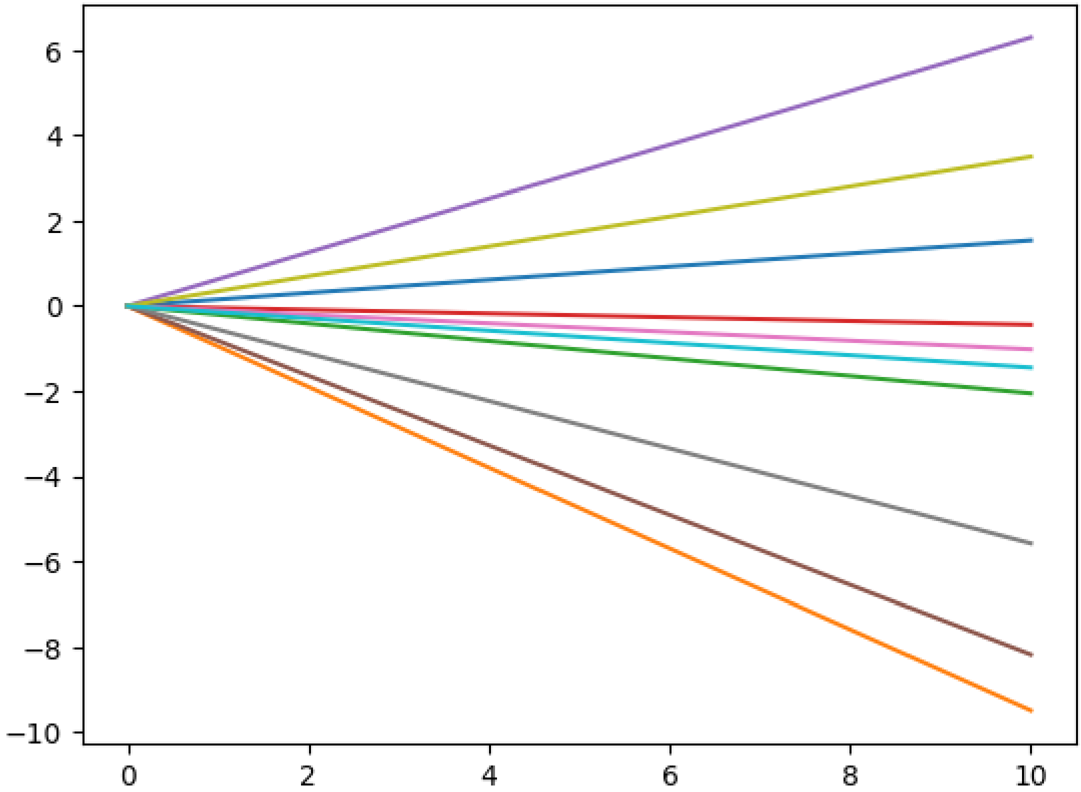
In [4]:
```
import matplotlib.pyplot as plt
for i in range(10):
  plt.plot(x, Function(5+3*i)(x))
```



Run the following code.

Question: What are the shapes of function graphs? Why there are multiple plots?

In [5]:
```
import matplotlib.pyplot as plt
for i in range(10):
  plt.plot(x,Function(5)(x))
```



TODO 1.1.2 Define two functions:

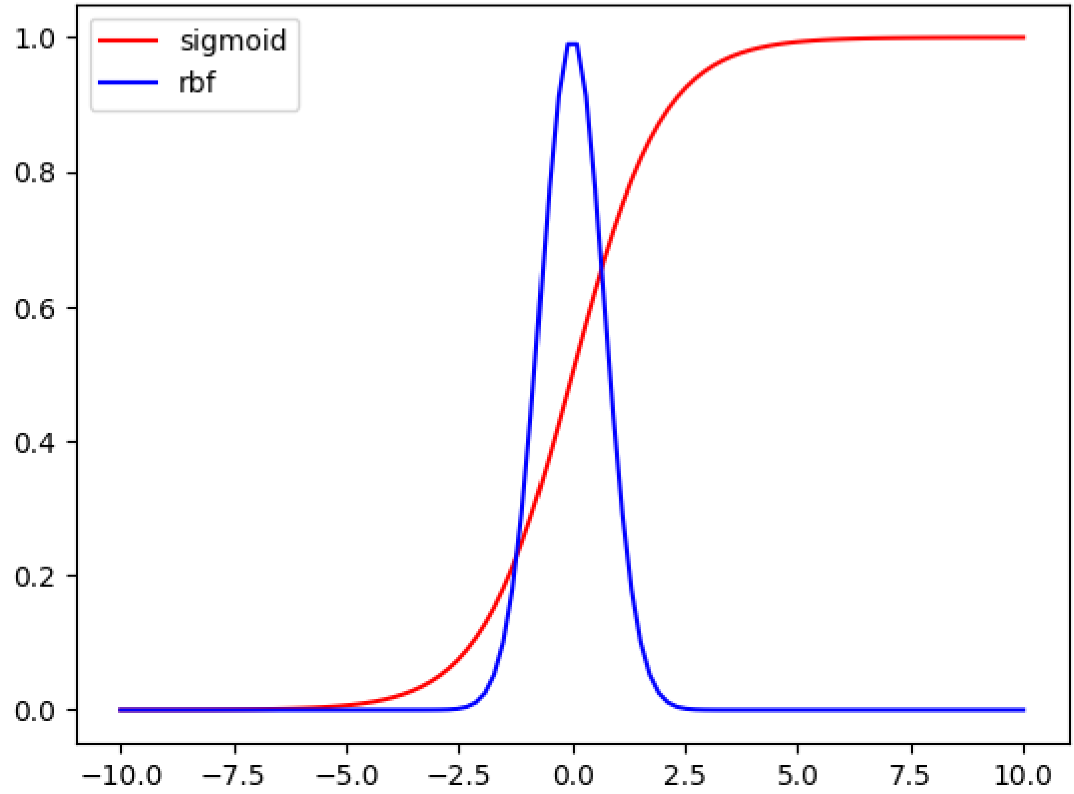
- $sigmoid(x) = \frac{1}{1+exp(-x)}$
- $rbf(x) = exp(-x^2)$

In [6]:
```
def sigmoid(z):
  return 1/(1 + np.exp(-z))

def rbf(z):
  return np.exp(-(z)*(z))
```

then plot their graphs

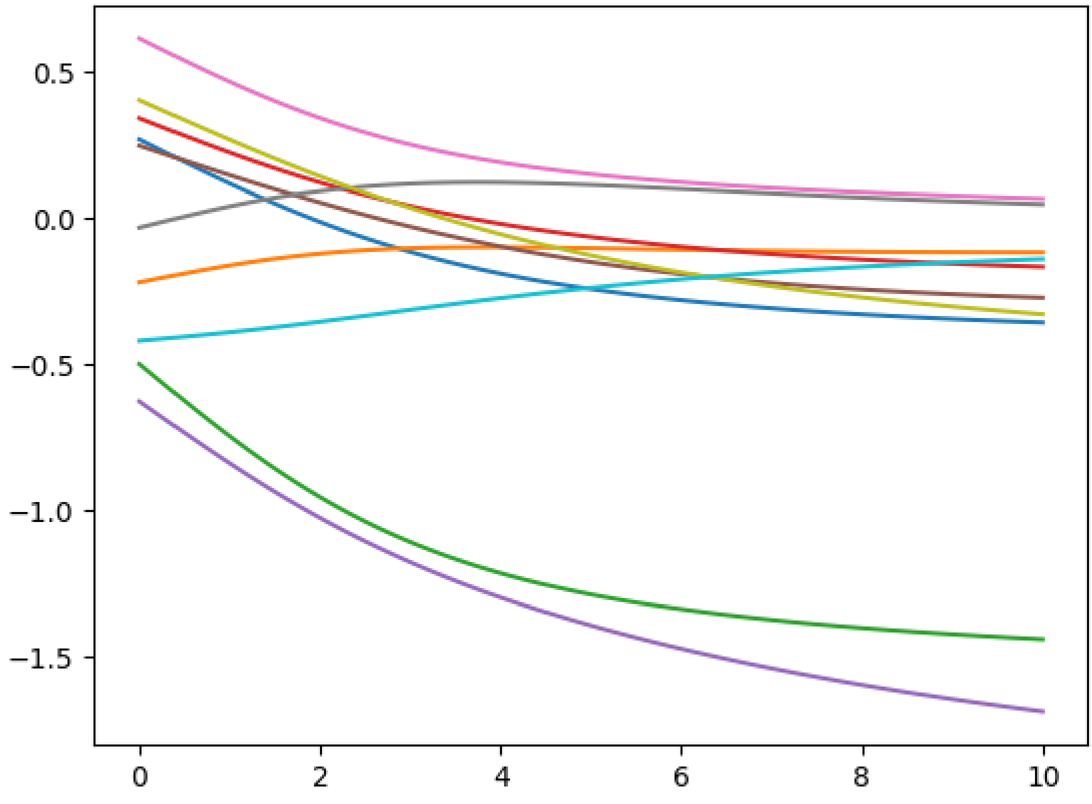
In [7]:
```
import matplotlib.pyplot as plt
x=np.linspace(-10,10,100)

plt.plot(x,sigmoid(x),c='r',label='sigmoid')
plt.plot(x,rbf(x),c='b',label='rbf')
plt.legend()
```

Out[7]: <matplotlib.legend.Legend at 0x1256c3be0>



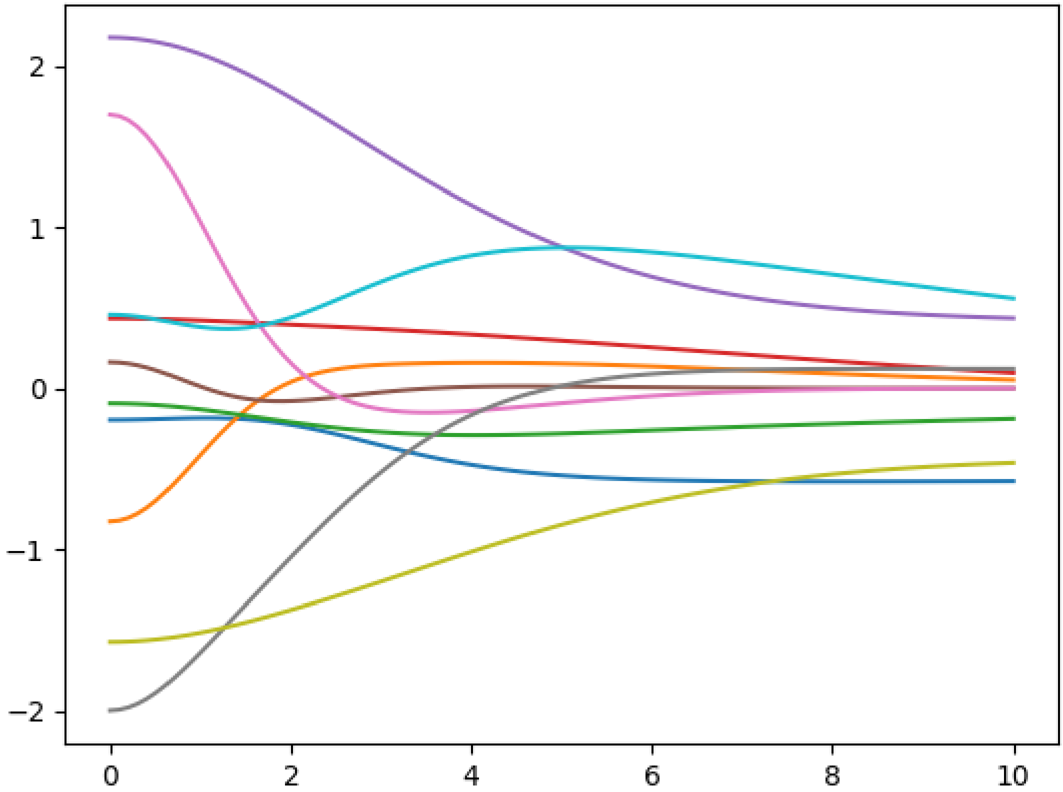
TODO 1.1.3 Display several function plots for activation = sigmoid

In [8]:
```
import matplotlib.pyplot as plt
x=np.linspace(0,10,100)
for i in range(10):
  plt.plot(x,Function(5, activation = sigmoid)(x))
```



TODO 1.1.4 Display several function plots for activation = rbf

In [9]:
```
import matplotlib.pyplot as plt
x=np.linspace(0,10,100)
for i in range(10):
  plt.plot(x,Function(5, activation = rbf)(x))
```



1.2 Implementation based on TensorFlow

In [10]:
```
import tensorflow as tf
print(tf.__version__)
```

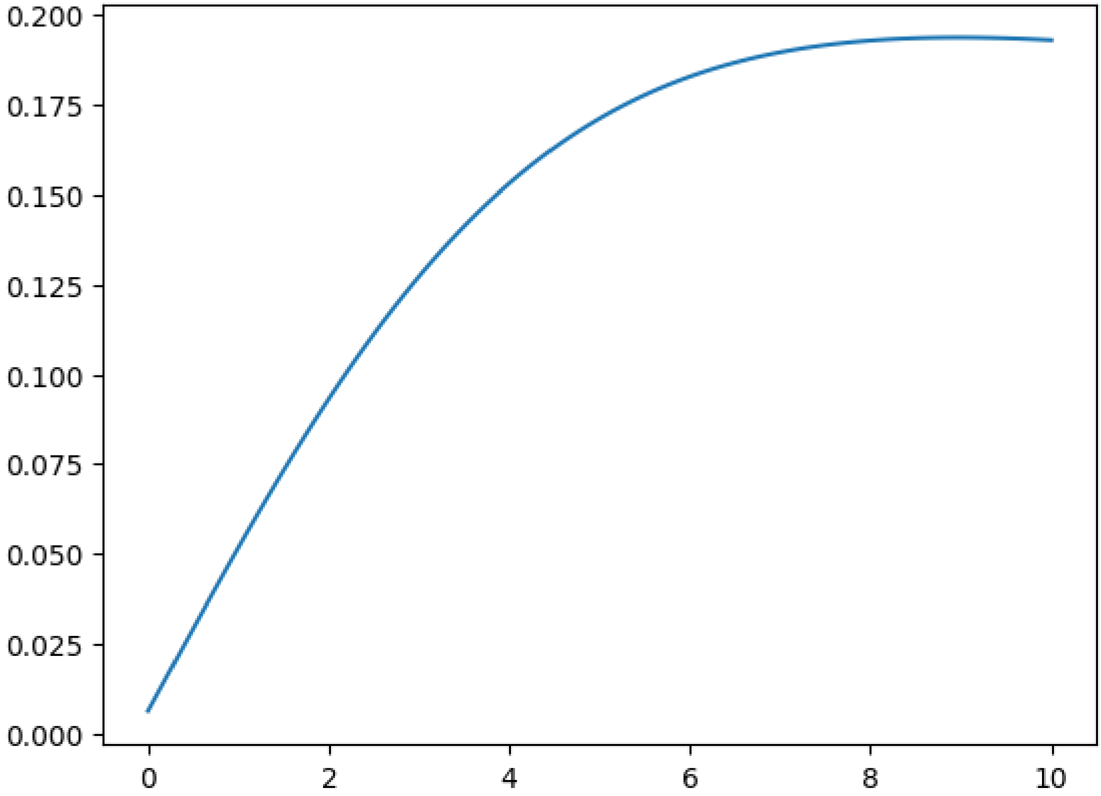
```
2.11.0
```

In [11]:
```
import tensorflow as tf


class Function:
  def __init__(self,n_h,activation = lambda x:x):
    self.f = activation
    self.W0=tf.Variable(np.random.randn(n_h,1)*np.sqrt(1/n_h))
    self.b0=tf.Variable(np.zeros((n_h,1)))
    self.W1=tf.Variable(np.random.randn(1,n_h)*np.sqrt(1/n_h))
    self.b1=tf.Variable(np.zeros((1,1)))


  def __call__(self,x):
    z=self.W0*x+self.b0
    a=self.f(z)
    y=tf.matmul(self.W1,a)+self.b1
    return y
```

Run the cell below several times. Each time the function shape changes.

In [12]:
```
x=np.linspace(0,10,100)
f=Function(4, activation = sigmoid)
y=f(x)
plt.plot(x,y.numpy()[0])
```

Out[12]: [<matplotlib.lines.Line2D at 0x295f37e20>]



How to fit the model to a given function?

We need

1. A measure to evaluate model fitness
2. A loss function to find the optimal model
3. Loss function may be identical to measure (but does not have to)
4. An optimization procedure that minimizes the loss

TODO 1.2.1 Analyze the code in the cell below and complete the code of MSE function. MSE means Mean Squared Error

In [13]:
```
a = tf.Variable([1.0,2.0,3.0,4.0])
b = tf.Variable([1.1,2.1,3.1,4.1])
e = (a-b)**2
print(e)
mse=tf.math.reduce_sum(e)/e.shape[0]
print(mse)
```

```
tf.Tensor([0.01       0.00999998 0.00999998 0.00999998], shape=(4,), dty
pe=float32)
tf.Tensor(0.009999987, shape=(), dtype=float32)
```

In [14]:
```
def MSE(y_true,y_pred):
  e = (y_true - y_pred)**2
  return tf.math.reduce_sum(e)/e.shape[0] #sum divided by number of eleme
```

TODO 1.2.2 Rewrite sigmoid and rbf functions using TensorFlow

In [15]:
```
def sigmoid(z):
  return 1/(1 + tf.exp(-z))

def rbf(z):
  return tf.exp(-(z)*(z))
```

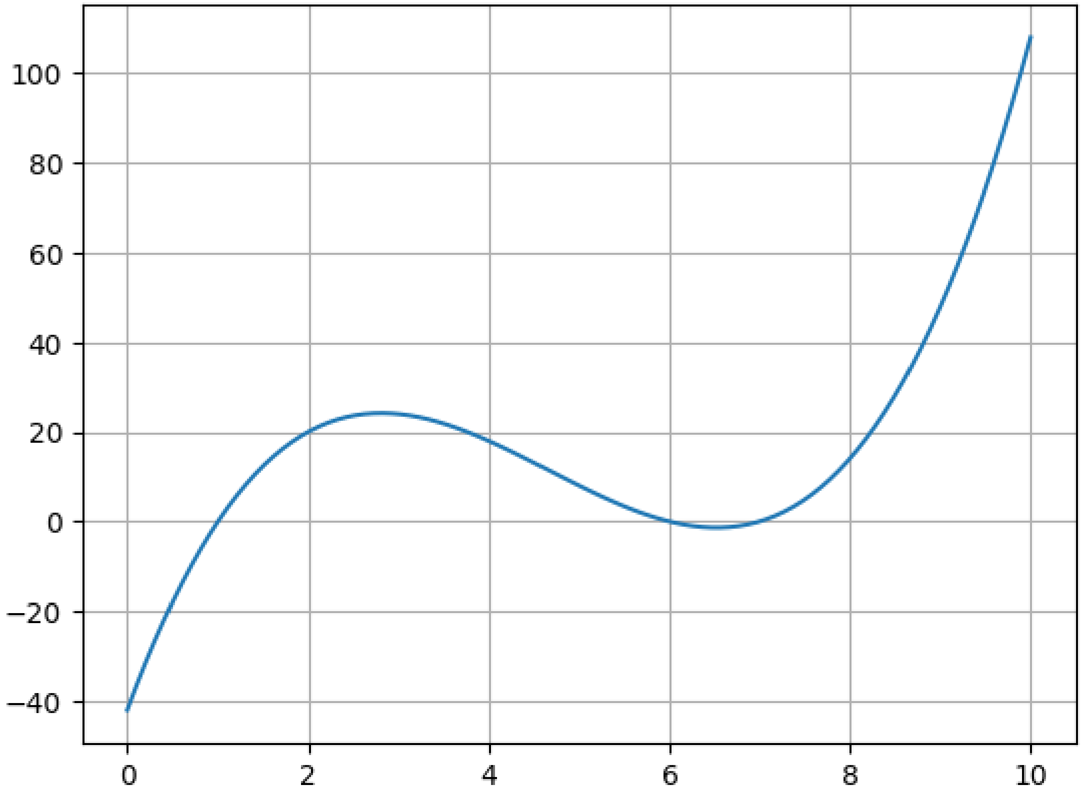
The fit method

- Input: x and y
- Iterates multiple times (parameter epoch)
- In each iteration
  - Calculates y_pred = model(x)
  - Computes loss function
  - Computes gradient of loss function with respect to weights
  - Updates weights, basicaly according to the formula $W = W - gradient * learning\_rate$.
  - Actually uses an optimizer that performs this in a smarter way

In [16]:
```
import tensorflow as tf


class Function:
  def __init__(self,n_h,activation = lambda x:x):
    self.f = activation
    self.W0=tf.Variable(np.random.randn(n_h,1)*0.01)
    self.b0=tf.Variable(np.zeros((n_h,1)))
    self.W1=tf.Variable(np.random.randn(1,n_h)*0.01)
    self.b1=tf.Variable(np.zeros((1,1)))


  def __call__(self,x):
    z=self.W0*x+self.b0
    a=self.f(z)
    y=tf.matmul(self.W1,a)+self.b1
    return y


  def fit(self,x,y,epochs=10,optimizer = tf.keras.optimizers.RMSprop()):
    for i in range(epochs):
      with tf.GradientTape() as tape: #keeps track of operations that are
        y_pred=self(x)
        loss = MSE(y_pred,y)
        # print(loss)
        variables=(self.W0,self.b0,self.W1,self.b1)
        gradients = tape.gradient(loss, variables)
        # print(gradients)
        optimizer.apply_gradients(zip(gradients, variables))
```

We will try to fit our model (Function class) to the polynomial function $y = (x-1)(x-6)(x-7)$

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0,10,100)
y = (x-1)*(x-6)*(x-7)

plt.plot(x,y)
plt.grid()
```

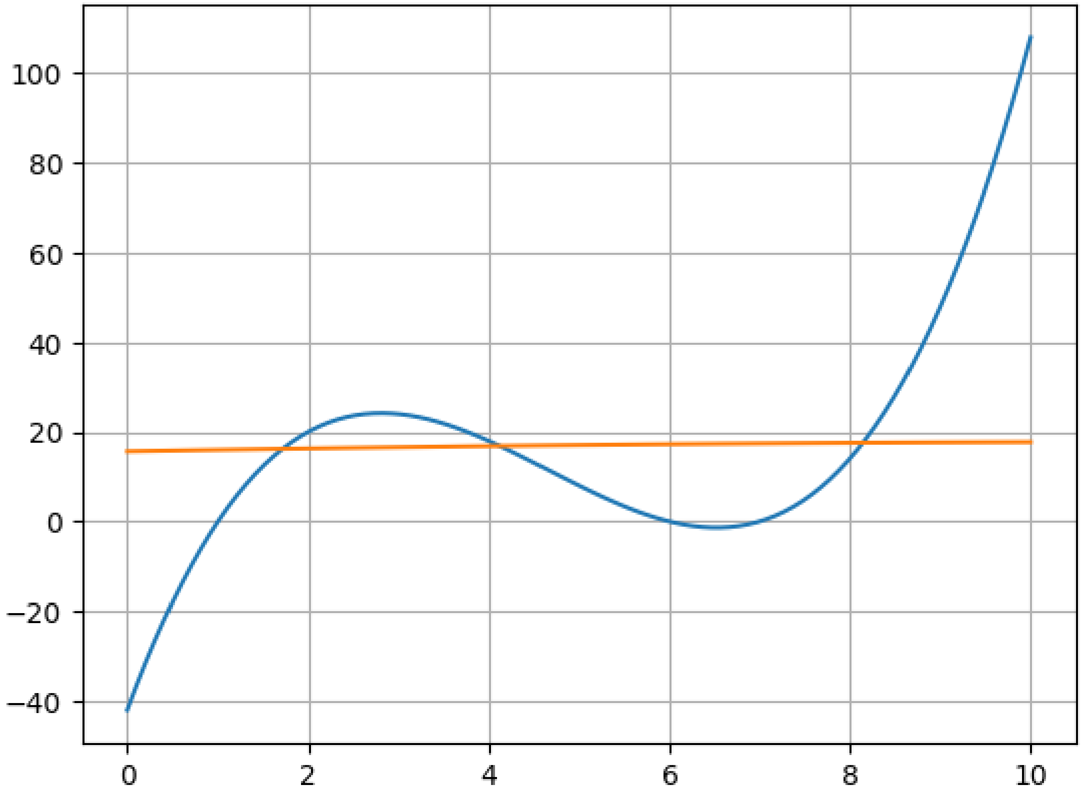


Although the problem is super-easy for classical methods, using this approach is a little bit hard. We need many hidden units and iterations... (execution abot 90 sec)

TODO 1.2.3 Create a model (Function) object passing as parameters 50 hidden units and rbf activation function. Fit the model setting number of iterations to 5000.

In [18]:
```
f=Function(n_h = 50, activation=rbf)
f.fit(x, y, epochs = 500)

plt.plot(x,y)
plt.grid()

y_pred=f(x)
plt.plot(x,y_pred.numpy()[0])
```

Out[18]: [<matplotlib.lines.Line2D at 0x296f40dc0>]



Hyperparameters

- n_h (number of hidden neurons) controls the model complexity
- activation function - influences the model performance
- epochs - controls number of iterations (influences the learning algorithm)

1.3 Neural network model

Analogous model can be built using components of keras library.

- **Advantage** the computations are converted to form a *computational graph* that can be executed much faster. Also on GPU. This is done with `compile` method.

In [19]:
```
from keras import models
from keras import layers

def build_model(n_h):
  model = models.Sequential()
  model.add(layers.Dense(n_h, activation=rbf, input_shape=(1,)))
  model.add(layers.Dense(1))
  model.compile(optimizer='rmsprop', loss='mse', metrics=['mse','mae'])
  return model
```

TODO 1.3.1 Create a model with 50 hidden units and call fit function setting number of epochs 5000 and batch_size (another hyperparameter) to 100

In [20]:
```
import numpy as np
x = np.linspace(0,10,100)
y = (x-1)*(x-6)*(x-7)
```

```
model = build_model(50)
history = model.fit(x, y, epochs = 10000, batch_size = 100, verbose=0)
```

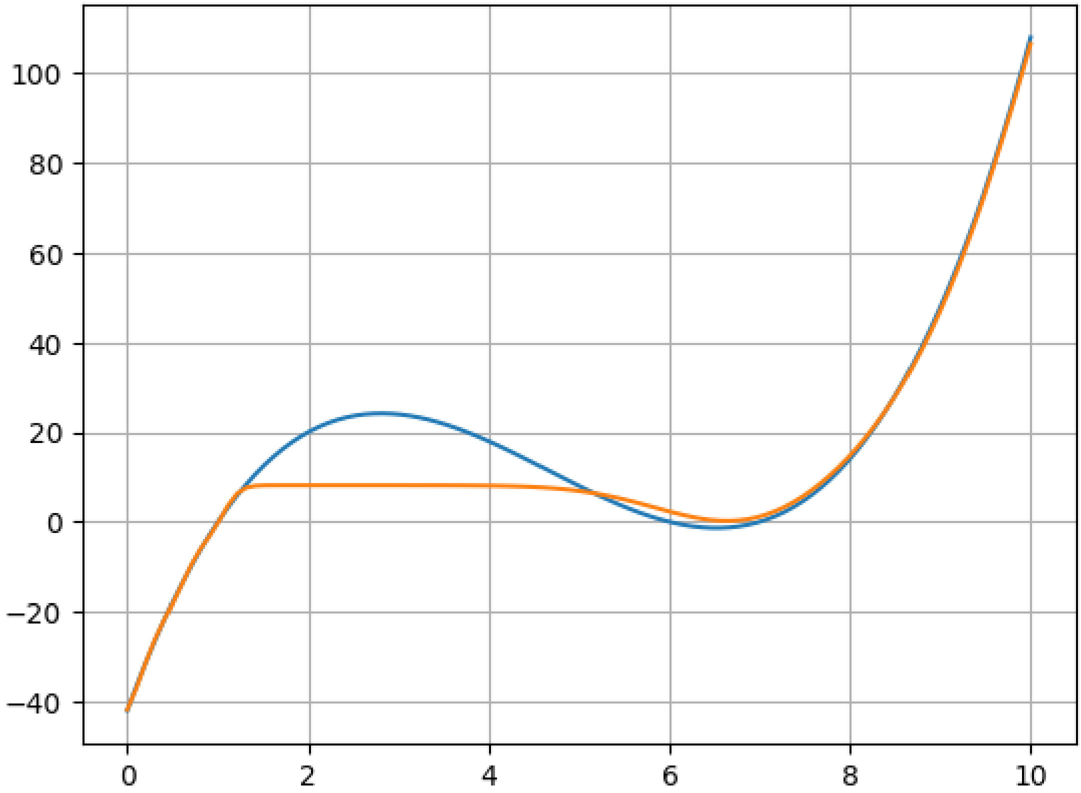
```
2023-03-11 10:52:45.027021: W tensorflow/tsl/platform/profile_utils/cpu_
utils.cc:128] Failed to get CPU frequency: 0 Hz
```

Check plots of original and fit curves

In [21]:
```
plt.plot(x,y)
plt.grid()

y_pred=model.predict(x)
plt.plot(x,y_pred)
```

```
4/4 [==============================] - 0s 669us/step
```

Out[21]: [<matplotlib.lines.Line2D at 0x298e075b0>]

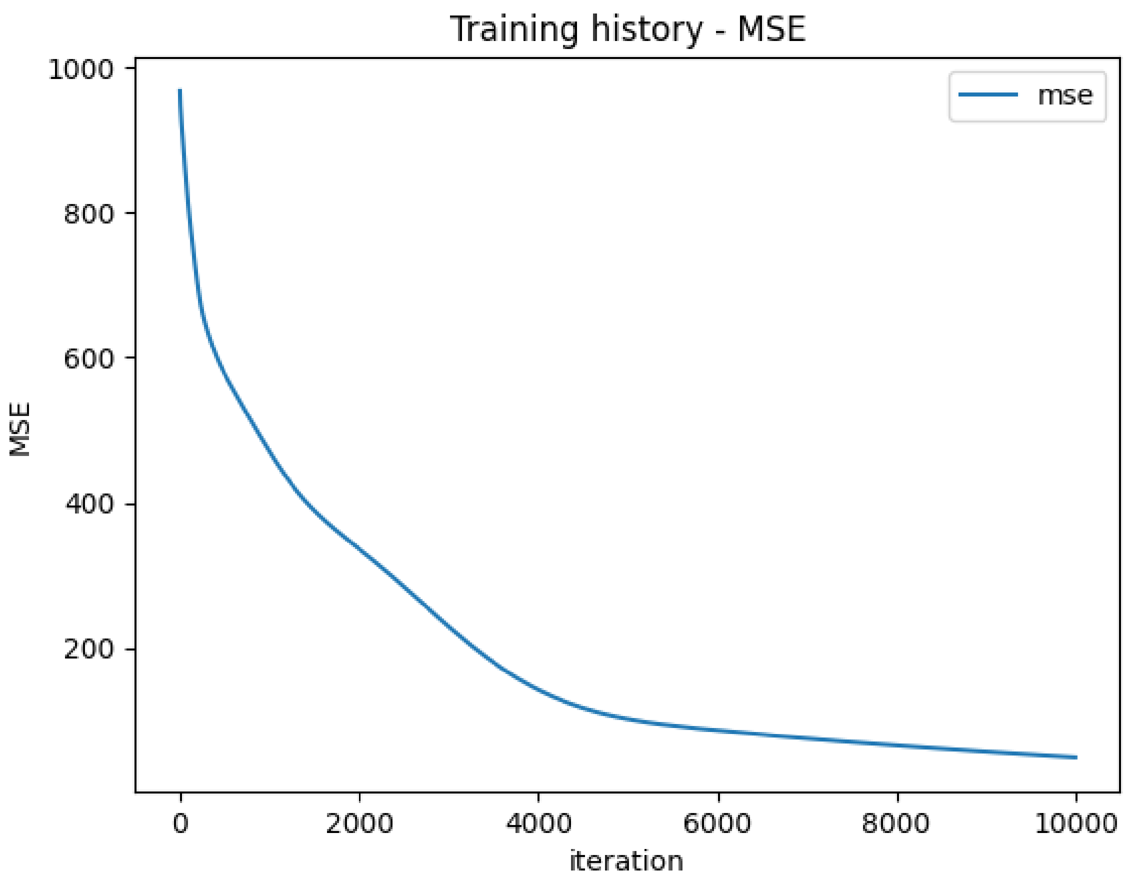


TODO 1.3.2 Repat the above steps changing hyperparameters to get a good fit

During training some data are collected. We may display various measures residing in the training history

In [22]:
```
import matplotlib.pyplot as plt

plt.title('Training history - MSE')
plt.plot(history.history['mse'],label='mse')
plt.xlabel('iteration')
plt.ylabel('MSE')
plt.legend()

# history.history['mse']
```

Out[22]: <matplotlib.legend.Legend at 0x298be6dd0>



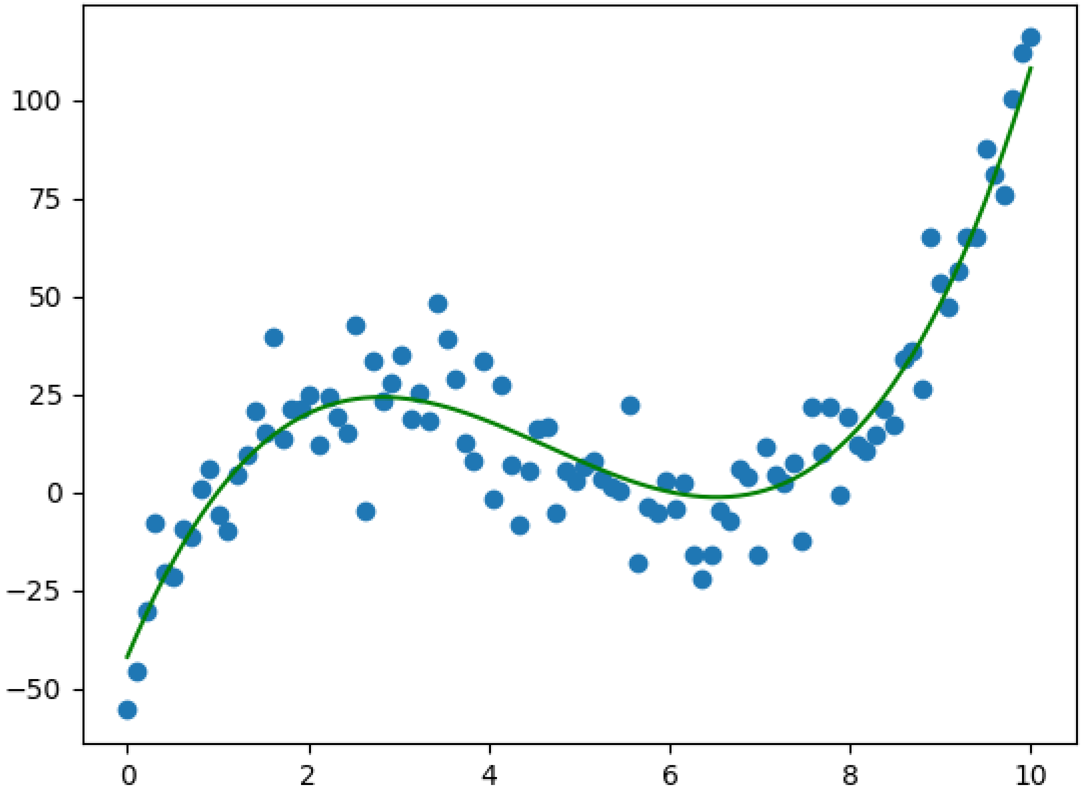
1.4 More realsitic model

The task of perfectly fitting a known function is very rare.

- It is rather assumed that we have data that originate from a true underlaying function with a noise $y = f(x) + \varepsilon$
- It is also often assumed that $\varepsilon \sim N(0, \sigma)$

In [23]:
```
from keras import models
from keras import layers
import numpy as np
import matplotlib.pyplot as plt

n_size=100
x = np.linspace(0,10,n_size)
y = (x-1)*(x-6)*(x-7)+np.random.normal(0,10,n_size)

plt.scatter(x,y)
plt.plot(x,(x-1)*(x-6)*(x-7),color='g')
```

Out[23]: [<matplotlib.lines.Line2D at 0x298c86e30>]



TODO 1.4.1 Fit the model to this DATA using the best hyperparameters obtained before

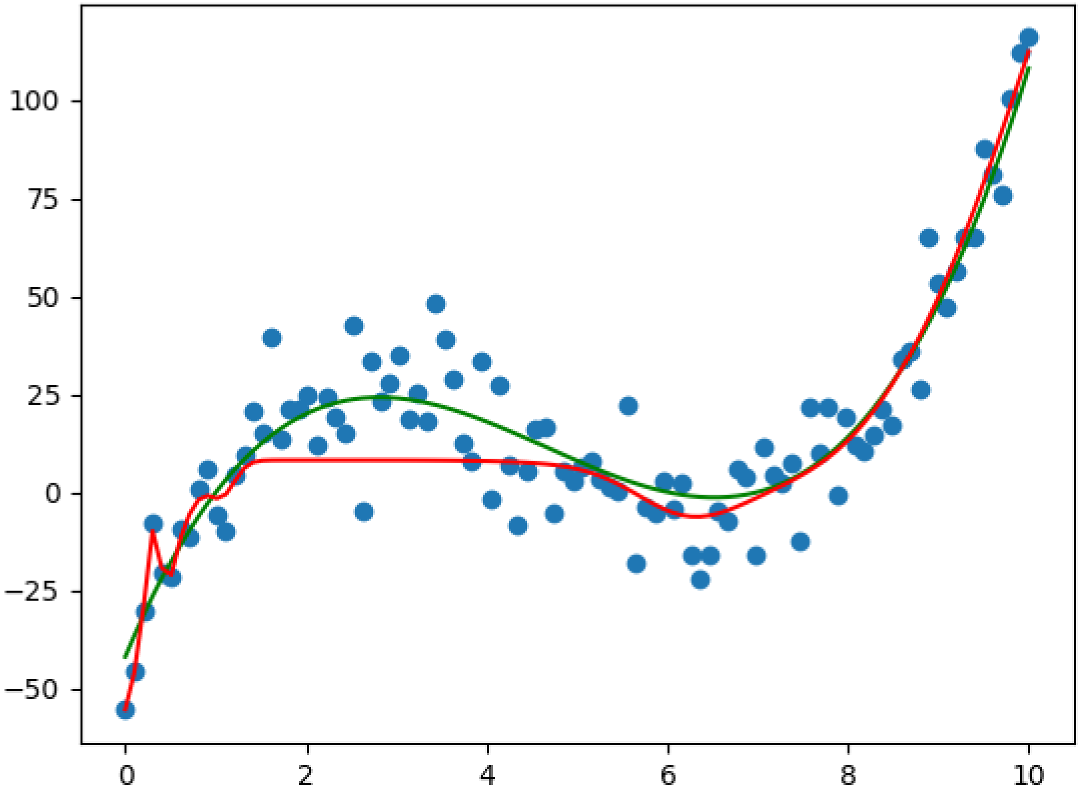
In [24]:
```
model = build_model(50)
history =  model.fit(x, y, epochs = 10000, batch_size = 100, verbose=0)
```

TODO 1.4.2 Plot the scattered data, true function in green and predictions in red

In [25]:
```
plt.scatter(x,y)
plt.plot(x,(x-1)*(x-6)*(x-7),color='g')
y_pred=model.predict(x)
plt.plot(x,y_pred,color='r')
```

```
4/4 [==============================] - 0s 633us/step
```

Out[25]: `[<matplotlib.lines.Line2D at 0x295be5780>]`



1.5 Validating model - training and testing

Typical ML workflow includes training the model and testing its performance on unseen data.

- **Why** - to control and assess generalization error which may result from
  - underfitting - the model is to simple or not trained enough
  - overfitting - the model is too complex, matches perfectly the training data (see part of the plot on the left)

We will split the data into two subsets

In [26]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
```

TODO 1.5.1 Fit the model using x_train and y_train, set the parameter `validation_data=(x_test, y_test)`

Warning: training lasts up to 250 sec

In [27]:
```
model = build_model(50)
history =  model.fit(x_train, y_train, epochs = 10000, batch_size = x_tra
```

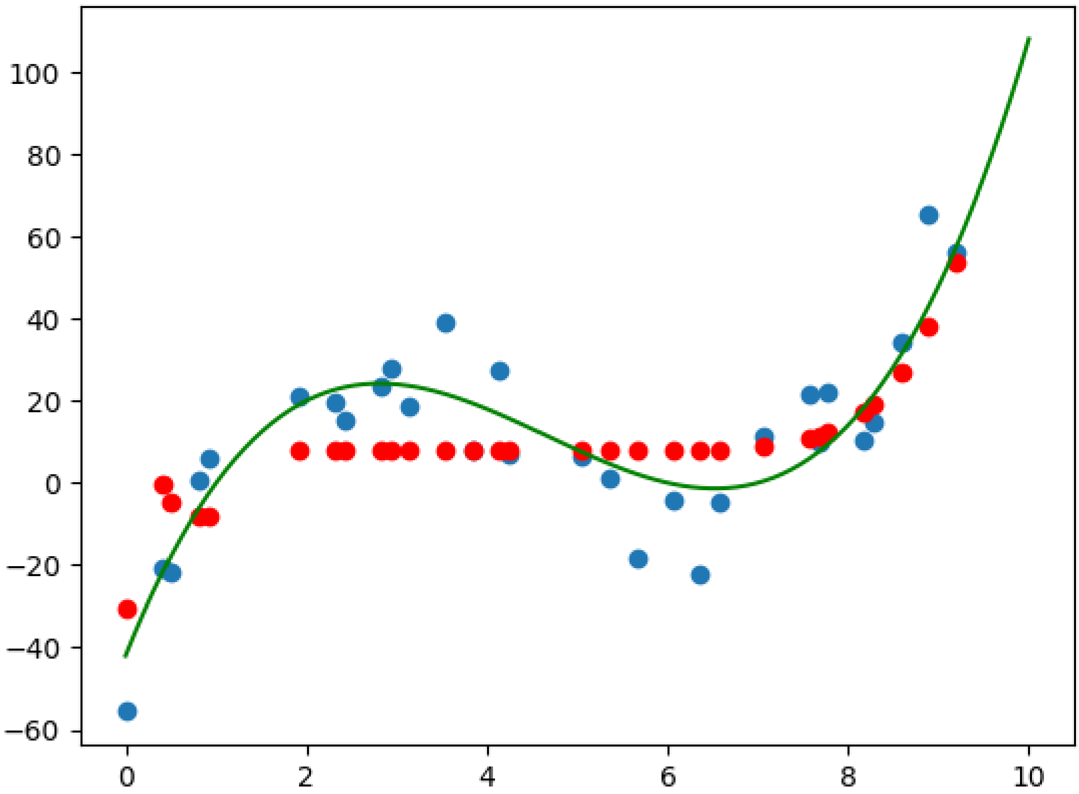
We will display true function, noisy data and predictions

In [28]:
```
plt.scatter(x_test,y_test)
plt.plot(x,(x-1)*(x-6)*(x-7),color='g')
```

```
y_pred=model.predict(x_test)
plt.scatter(x_test,y_pred,color='r')
```

```
1/1 [==============================] - 0s 30ms/step
```

Out[28]: <matplotlib.collections.PathCollection at 0x29c98f550>



Lets peek what is the content of the history...

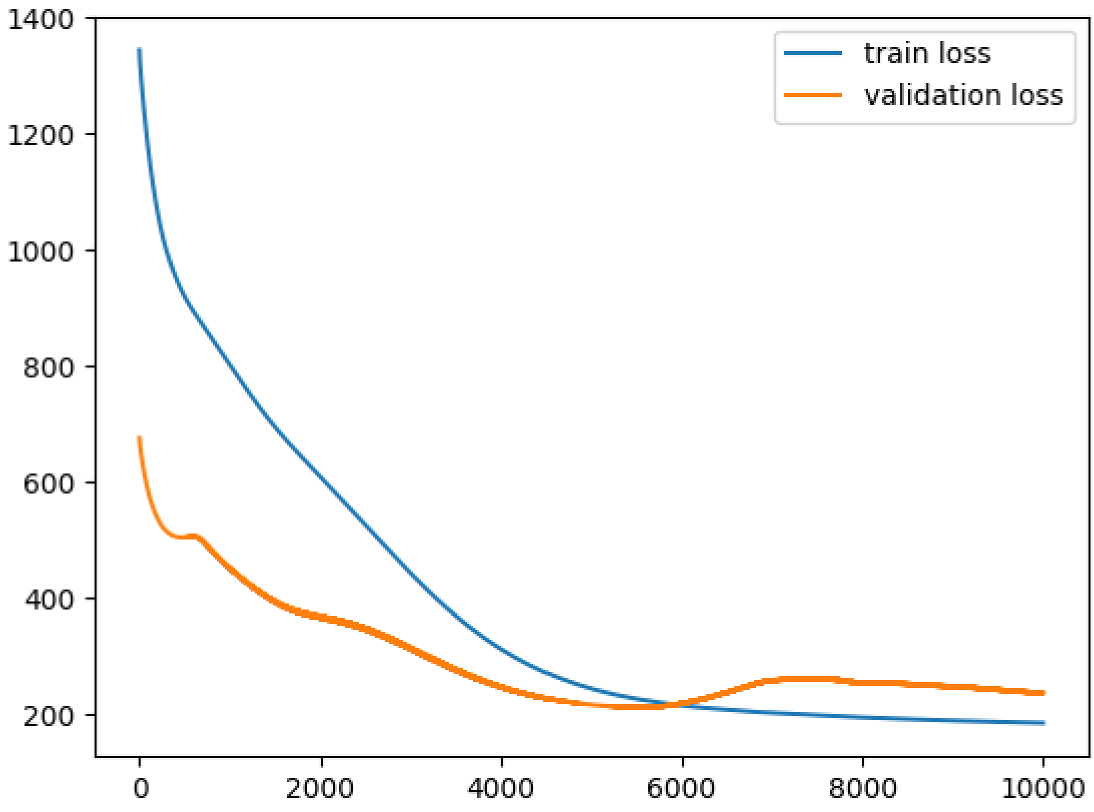
In [29]:
```
for k in history.history:
  print(k)
```

```
loss
mse
mae
val_loss
val_mse
val_mae
```

TODO 1.5.2 Display loss (training loss) and val_loss (validation loss on the test set)

In [30]:
```
plt.plot(history.history['loss'], label='train loss')
plt.plot(history.history['val_loss'], label='validation loss')
plt.legend()
```

Out[30]: <matplotlib.legend.Legend at 0x29b196fe0>



1.6 Classification

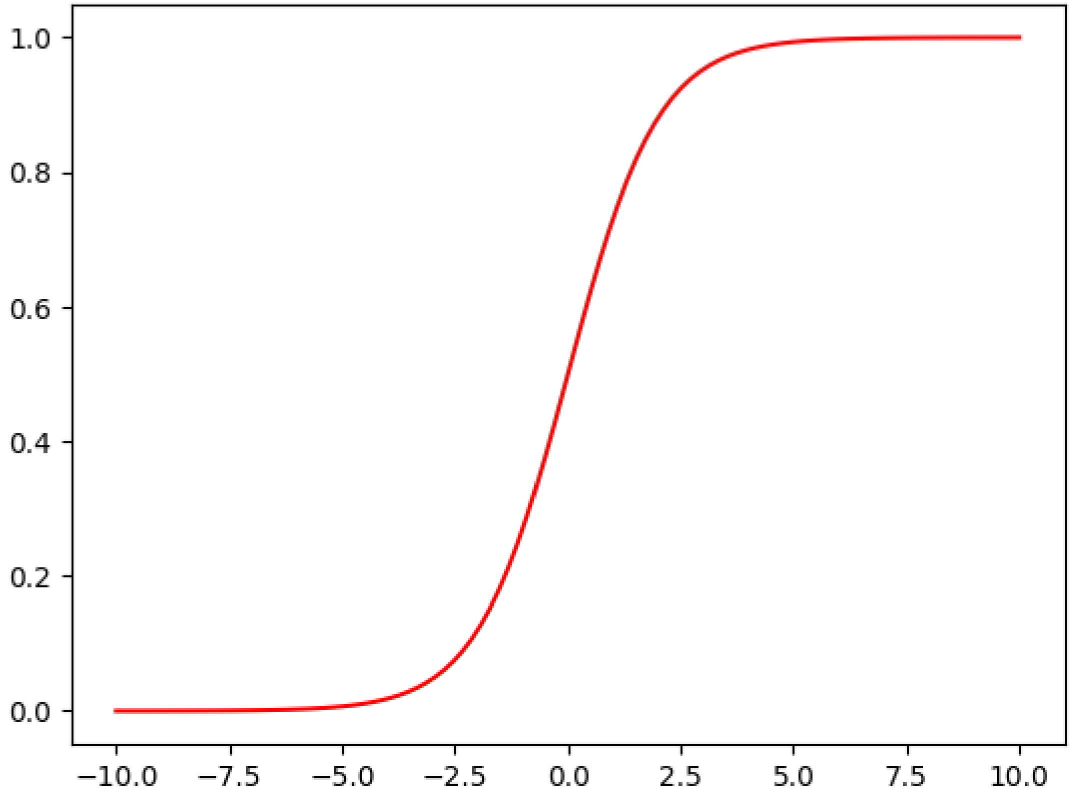
Function models can be used for classification, provided we constrain them to return probabilities, i.e. values from [0,1] interval.

- Function with one output may be used for binary classification:
  - Assign $label_0$ if $f(x) < 0.5$
  - Assign $label_1$ if $f(x) \geq 0.5$

TODO 1.6.1 Which function converts $R \rightarrow [0, 1]$? Answer the question

In [31]:
```
import matplotlib.pyplot as plt
x=np.linspace(-10,10,100)

plt.plot(x,(lambda x: 1/(1+np.exp(-x)))(x),c='r')
```

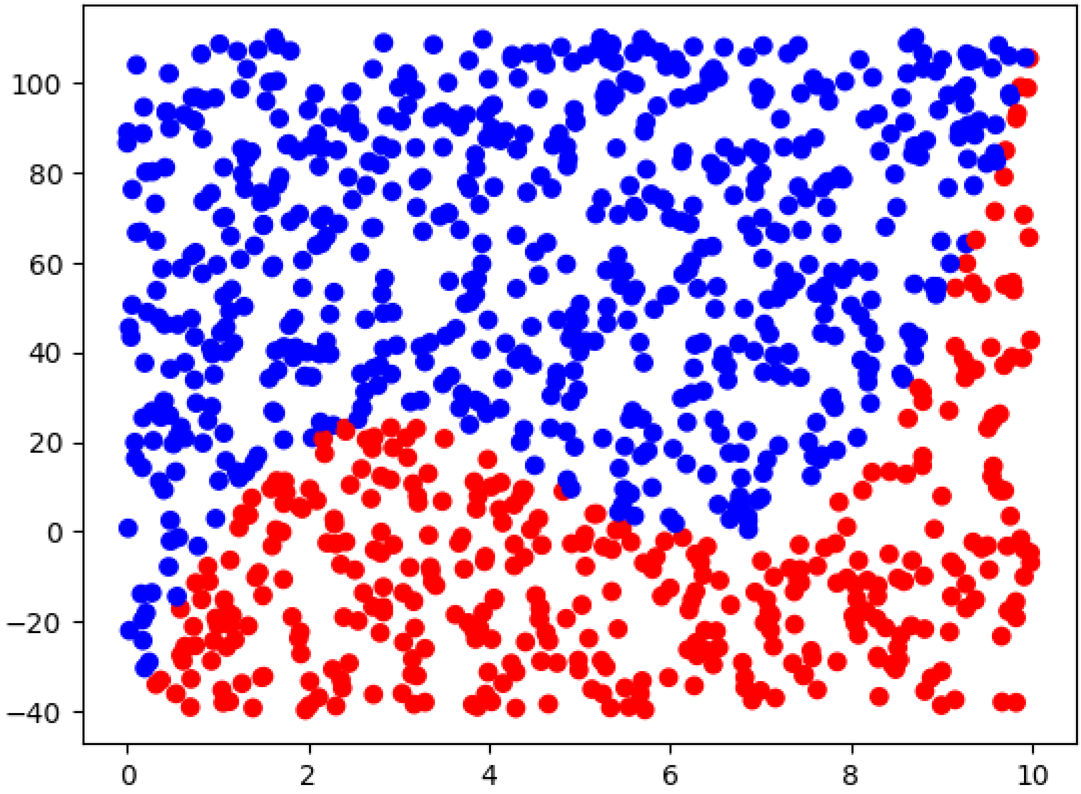
Out[31]:
```
[<matplotlib.lines.Line2D at 0x299da0bb0>]
```



We will generate a dataset. Points above the previously used polynomial will have blue label, the points below red.

In [32]:
```
X = np.random.rand(1000,2)*[10,150]-[0,40]
y = np.where(X[:,1]>(X[:,0]-1)*(X[:,0]-6)*(X[:,0]-7),1,0)
# y.shape

from matplotlib.colors import ListedColormap
cm = ListedColormap(['r', 'b'])
plt.scatter(X[:,0],X[:,1],c=y,cmap=cm)
```

Out[32]: <matplotlib.collections.PathCollection at 0x29b4d9b70>



We will biuld a model more suitable for classification.

What is binary_crossentropy aka. logloss?

$$loss_i = -[y_i \cdot ln(p_i) + (1 - y_i) \cdot ln(1 - p_i)]$$

You may google the term...

In [33]:
```
import tensorflow as tf


def build_classification_model(n_h):
  model = models.Sequential()
  model.add(layers.Dense(n_h, activation='relu', input_shape=(2,)))
  model.add(layers.Dense(n_h, activation='relu'))
  model.add(layers.Dense(1,activation='sigmoid'))
  model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=
  return model
```

TODO 1.6.2 fit the model using training data. Set about 100 epochs, use X_test and y_test as validation data.

In [34]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,

model = build_classification_model(10)
history =  model.fit(X_train, y_train, epochs = 100, batch_size = 100, ve
```

Display predictions and the polynomial curve which was used to separate class instances.

In [35]:
```
y_pred = model.predict(X_test)
plt.scatter(X_test[:,0],X_test[:,1],c=y_pred,cmap=cm)
```

```
x = np.linspace(0,10,100)
y = (x-1)*(x-6)*(x-7)
plt.plot(x,y,color='g')
```

```
10/10 [==============================] - 0s 553us/step
```

Out[35]: [<matplotlib.lines.Line2D at 0x29c9e0d60>]