Computational Intelligence Lab

Assignment 1

Szymon Berdzik ISI IO

1.1 Function

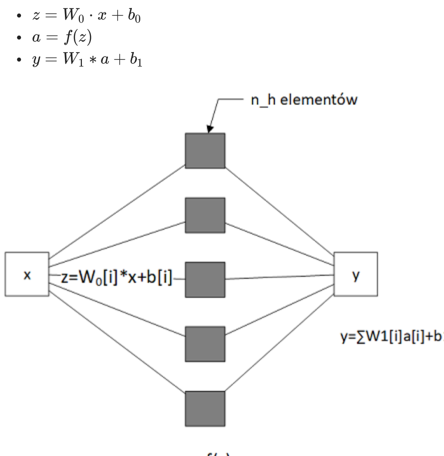
Consider the following Python class.

In [ ]:
```python
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)
y
```

Out[ ]:
```
array([0.        , 0.02820175, 0.0564035 , 0.08460526, 0.11280701,
       0.14100876, 0.16921051, 0.19741227, 0.22561402, 0.25381577,
       0.28201752, 0.31021928, 0.33842103, 0.36662278, 0.39482453,
       0.42302629, 0.45122804, 0.47942979, 0.50763154, 0.5358333 ,
       0.56403505, 0.5922368 , 0.62043855, 0.64864031, 0.67684206,
       0.70504381, 0.73324556, 0.76144732, 0.78964907, 0.81785082,
       0.84605257, 0.87425433, 0.90245608, 0.93065783, 0.95885958,
       0.98706134, 1.01526309, 1.04346484, 1.07166659, 1.09986835,
       1.1280701 , 1.15627185, 1.1844736 , 1.21267536, 1.24087711,
       1.26907886, 1.29728061, 1.32548237, 1.35368412, 1.38188587,
       1.41008762, 1.43828938, 1.46649113, 1.49469288, 1.52289463,
       1.55109639, 1.57929814, 1.60749989, 1.63570164, 1.6639034 ,
       1.69210515, 1.7203069 , 1.74850865, 1.77671041, 1.80491216,
       1.83311391, 1.86131566, 1.88951742, 1.91771917, 1.94592092,
       1.97412267, 2.00232443, 2.03052618, 2.05872793, 2.08692968,
       2.11513143, 2.14333319, 2.17153494, 2.19973669, 2.22793844,
       2.2561402 , 2.28434195, 2.3125437 , 2.34074545, 2.36894721,
       2.39714896, 2.42535071, 2.45355246, 2.48175422, 2.50995597,
       2.53815772, 2.56635947, 2.59456123, 2.62276298, 2.65096473,
       2.67916648, 2.70736824, 2.73556999, 2.76377174, 2.79197349])
```

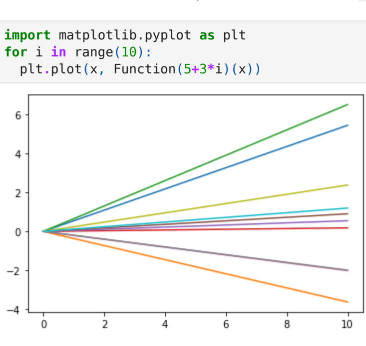
Operations placed in the function call operator can be expressed as:

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



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

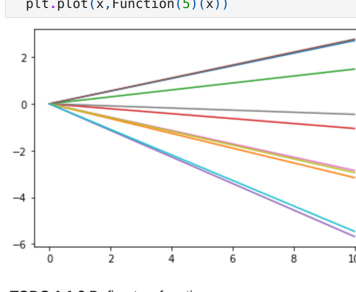
In [ ]:
```python
import matplotlib.pyplot as plt
for i in range(10):
  plt.plot(x, Function(5+3*i)(x))
```



Run the following code.

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

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



TODO 1.1.2 Define two functions:

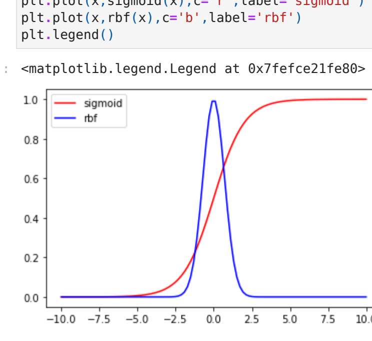
- $sigmoid(x) = \frac{1}{1+exp(-x)}$
- $rbf(x) = exp(-x^2)$

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

def rbf(z):
  return np.exp(-z**2)
```

then plot their graphs

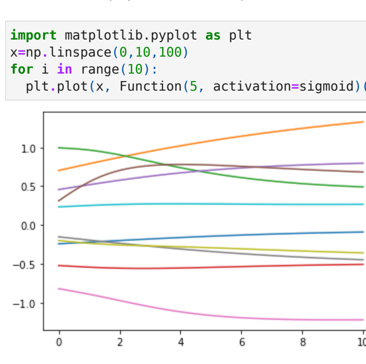
In [ ]:
```python
import matplotlib.pyplot as plt
x=np.linspace(-10,10,100)

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

Out[ ]: `<matplotlib.legend.Legend at 0x7fefce21fe80>`



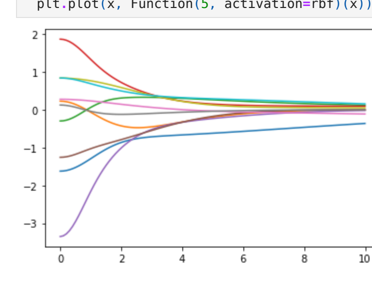
TODO 1.1.3 Display several function plots for activation = sigmoid

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

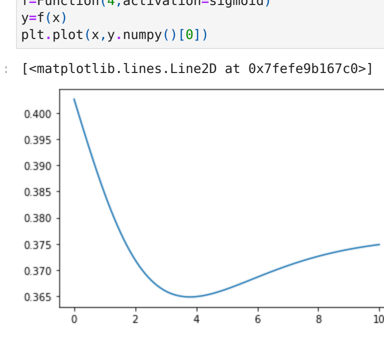
```
2.11.0
```

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

Out[ ]: `[<matplotlib.lines.Line2D at 0x7fefe9b167c0>]`



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 [ ]:
```python
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([0.01000001 0.00999998 0.00999998 0.00999998], shape=(4,), dtype=float32)
tf.Tensor(0.009999987, shape=(), dtype=float32)
```

In [ ]:
```python
def MSE(y_true,y_pred):
  e = (y_true - y_pred)**2
  return tf.math.reduce_sum(e)/e.shape[0]
```

TODO 1.2.2 Rewrite sigmoid and rbf functions using TensorFlow

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

def rbf(x):
  return tf.exp(-x**2)
```

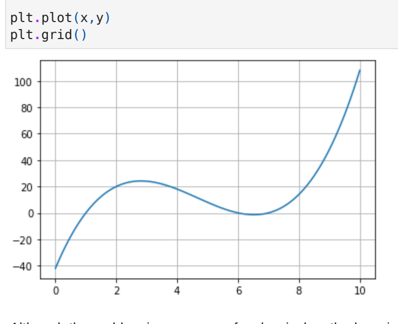
The fit method

- 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 [ ]:
```python
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:
        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 [ ]:
```python
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 [ ]:
```python
f=Function(n_h=50, activation=rbf)
f.fit(x,y, epochs=5000)

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

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

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 [ ]:
```python
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

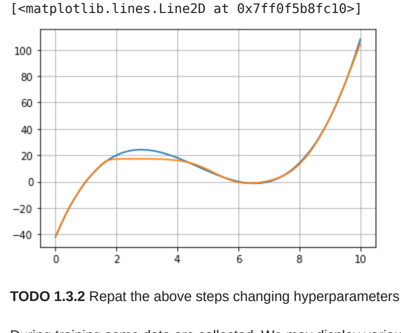
In [ ]:
```python
import numpy as np
x = np.linspace(0,10,100)
y = (x-1)*(x-6)*(x-7)

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

Check plots of original and fit curves

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

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

```
4/4 [==============================] - 0s 3ms/step
```

Out[ ]: `[<matplotlib.lines.Line2D at 0x7ff0f5b8fc10>]`

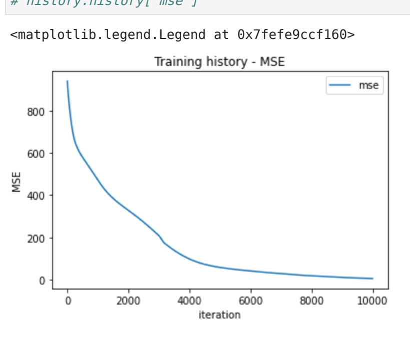


TODO 1.3.2 Repat the above steps changing hyperparameters to get a good fit

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

In [ ]:
```python
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[ ]: `<matplotlib.legend.Legend at 0x7fefe9ccf160>`



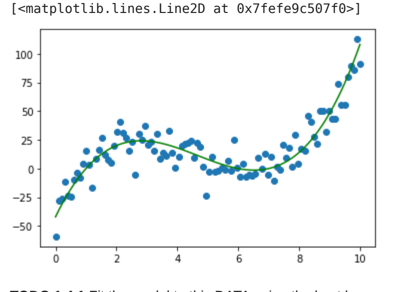
1.4 More realsitic model

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 [ ]:
```python
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[ ]: `[<matplotlib.lines.Line2D at 0x7fefe9c507f0>]`



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

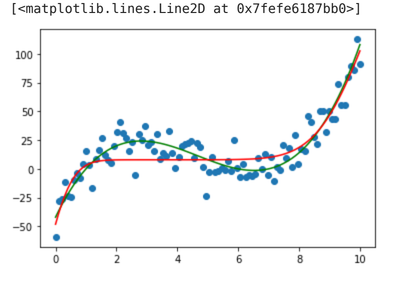
In [ ]:
```python
model = build_model(50)
history = model.fit(x,y, epochs=10000, batch_size=100, verbose=0)
```

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

In [ ]:
```python
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 2ms/step
```

Out[ ]: `[<matplotlib.lines.Line2D at 0x7fefe6187bb0>]`



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 [ ]:
```python
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, random_state=123)
```

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

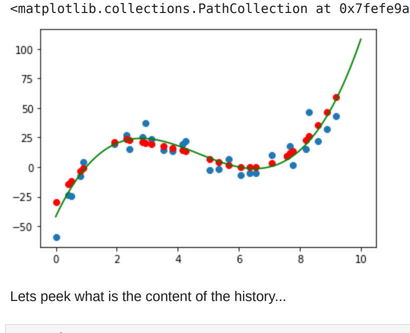
In [ ]:
```python
model = build_model(50)
history = model.fit(x_train,y_train, epochs=10000, batch_size=x_train.shape[0], verbose=0, validati
```

We will display true function, noisy data and predictions

In [ ]:
```python
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 68ms/step
```

Out[ ]: `<matplotlib.collections.PathCollection at 0x7fefe9a57eb0>`



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

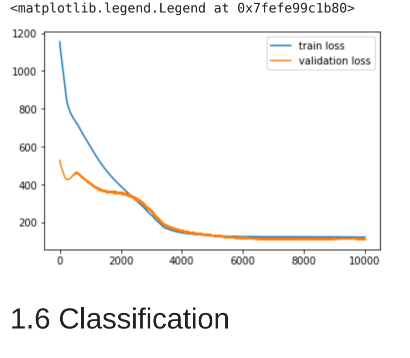
In [ ]:
```python
for k in history.history:
  print(k)
```

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

Out[ ]: `<matplotlib.legend.Legend at 0x7fefe99c1b80>`



1.6 Classification

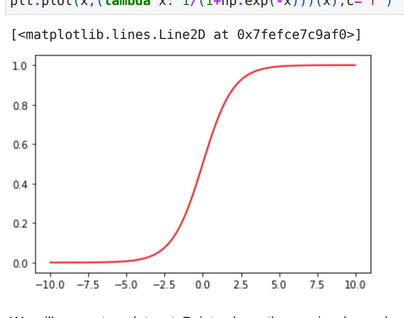
Function models can be used for classification, provided we constrain them to return probabilities, i.e. values from [0,1] interval.

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

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

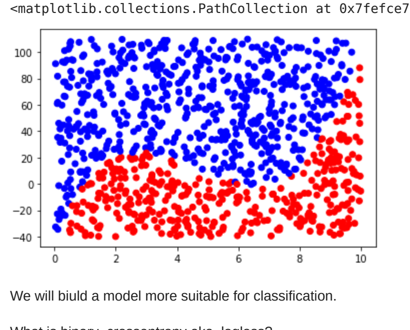
Out[ ]: `[<matplotlib.lines.Line2D at 0x7fefce7c9af0>]`



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

In [ ]:
```python
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[ ]: `<matplotlib.collections.PathCollection at 0x7fefce7b0af0>`



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 [ ]:
```python
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=['accuracy'])
  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 [ ]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=123)

model = build_classification_model(10)
history = model.fit(X_train,y_train, epochs=100, batch_size=100, verbose=1, validation_data=(X_test
```

```
Epoch 1/100
7/7 [==============================] - 1s 33ms/step - loss: 2.6268 - accuracy: 0.4243 - val_loss: 0.5760 - val_accuracy: 0.8200
Epoch 2/100
7/7 [==============================] - 0s 10ms/step - loss: 0.5125 - accuracy: 0.8486 - val_loss: 0.5076 - val_accuracy: 0.8200
Epoch 3/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4758 - accuracy: 0.8486 - val_loss: 0.4955 - val_accuracy: 0.8233
Epoch 4/100
7/7 [==============================] - 0s 9ms/step - loss: 0.4602 - accuracy: 0.8486 - val_loss: 0.4819 - val_accuracy: 0.8233
Epoch 5/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4450 - accuracy: 0.8486 - val_loss: 0.4687 - val_accuracy: 0.8233
Epoch 6/100
7/7 [==============================] - 0s 9ms/step - loss: 0.4288 - accuracy: 0.8500 - val_loss: 0.4517 - val_accuracy: 0.8233
Epoch 7/100
7/7 [==============================] - 0s 10ms/step - loss: 0.4131 - accuracy: 0.8486 - val_loss: 0.4337 - val_accuracy: 0.8233
Epoch 8/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3991 - accuracy: 0.8500 - val_loss: 0.4157 - val_accuracy: 0.8233
Epoch 9/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3854 - accuracy: 0.8500 - val_loss: 0.4151 - val_accuracy: 0.8233
Epoch 10/100
7/7 [==============================] - 0s 9ms/step - loss: 0.3688 - accuracy: 0.8514 - val_loss: 0.4082 - val_accuracy: 0.8233
Epoch 11/100
7/7 [==============================] - 0s 8ms/step - loss: 0.3567 - accuracy: 0.8514 - val_loss: 0.3724 - val_accuracy: 0.8233
Epoch 12/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3461 - accuracy: 0.8500 - val_loss: 0.3666 - val_accuracy: 0.8233
Epoch 13/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3323 - accuracy: 0.8486 - val_loss: 0.3544 - val_accuracy: 0.8233
Epoch 14/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3257 - accuracy: 0.8500 - val_loss: 0.3368 - val_accuracy: 0.8267
Epoch 15/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3237 - accuracy: 0.8443 - val_loss: 0.3304 - val_accuracy: 0.8333
Epoch 16/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3140 - accuracy: 0.8500 - val_loss: 0.3291 - val_accuracy: 0.8400
Epoch 17/100
7/7 [==============================] - 0s 7ms/step - loss: 0.3088 - accuracy: 0.8586 - val_loss: 0.3435 - val_accuracy: 0.8367
Epoch 18/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3140 - accuracy: 0.8529 - val_loss: 0.3226 - val_accuracy: 0.8667
Epoch 19/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2994 - accuracy: 0.8529 - val_loss: 0.3162 - val_accuracy: 0.8667
Epoch 20/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2979 - accuracy: 0.8457 - val_loss: 0.3155 - val_accuracy: 0.8700
Epoch 21/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2995 - accuracy: 0.8600 - val_loss: 0.3042 - val_accuracy: 0.8767
Epoch 22/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2943 - accuracy: 0.8514 - val_loss: 0.3068 - val_accuracy: 0.8800
Epoch 23/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2926 - accuracy: 0.8643 - val_loss: 0.2986 - val_accuracy: 0.8867
Epoch 24/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2863 - accuracy: 0.8571 - val_loss: 0.2897 - val_accuracy: 0.8900
Epoch 25/100
7/7 [==============================] - 0s 11ms/step - loss: 0.2839 - accuracy: 0.8500 - val_loss: 0.3101 - val_accuracy: 0.8867
Epoch 26/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2876 - accuracy: 0.8557 - val_loss: 0.3184 - val_accuracy: 0.8900
Epoch 27/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2900 - accuracy: 0.8586 - val_loss: 0.2880 - val_accuracy: 0.8933
Epoch 28/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2823 - accuracy: 0.8600 - val_loss: 0.2997 - val_accuracy: 0.8900
Epoch 29/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2843 - accuracy: 0.8600 - val_loss: 0.2862 - val_accuracy: 0.8933
Epoch 30/100
7/7 [==============================] - 0s 11ms/step - loss: 0.2817 - accuracy: 0.8614 - val_loss: 0.2781 - val_accuracy: 0.9033
Epoch 31/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2834 - accuracy: 0.8671 - val_loss: 0.2918 - val_accuracy: 0.8967
Epoch 32/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2772 - accuracy: 0.8643 - val_loss: 0.2931 - val_accuracy: 0.8933
Epoch 33/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2760 - accuracy: 0.8686 - val_loss: 0.3027 - val_accuracy: 0.8900
Epoch 34/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2771 - accuracy: 0.8671 - val_loss: 0.2871 - val_accuracy: 0.9000
Epoch 35/100
7/7 [==============================] - 0s 11ms/step - loss: 0.2775 - accuracy: 0.8671 - val_loss: 0.2779 - val_accuracy: 0.9000
Epoch 36/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2787 - accuracy: 0.8557 - val_loss: 0.2964 - val_accuracy: 0.8967
Epoch 37/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2732 - accuracy: 0.8686 - val_loss: 0.2987 - val_accuracy: 0.8967
Epoch 38/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2740 - accuracy: 0.8729 - val_loss: 0.2868 - val_accuracy: 0.9000
Epoch 39/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2707 - accuracy: 0.8757 - val_loss: 0.3187 - val_accuracy: 0.8933
Epoch 40/100
7/7 [==============================] - 0s 11ms/step - loss: 0.2727 - accuracy: 0.8743 - val_loss: 0.2837 - val_accuracy: 0.9000
Epoch 41/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2705 - accuracy: 0.8700 - val_loss: 0.2738 - val_accuracy: 0.9100
Epoch 42/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2664 - accuracy: 0.8657 - val_loss: 0.3013 - val_accuracy: 0.8967
Epoch 43/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2688 - accuracy: 0.8714 - val_loss: 0.3042 - val_accuracy: 0.8967
Epoch 44/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2704 - accuracy: 0.8771 - val_loss: 0.2785 - val_accuracy: 0.9067
Epoch 45/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2659 - accuracy: 0.8671 - val_loss: 0.2795 - val_accuracy: 0.9067
Epoch 46/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2673 - accuracy: 0.8757 - val_loss: 0.2650 - val_accuracy: 0.9133
Epoch 47/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2679 - accuracy: 0.8757 - val_loss: 0.2768 - val_accuracy: 0.9067
Epoch 48/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2622 - accuracy: 0.8729 - val_loss: 0.2989 - val_accuracy: 0.9067
Epoch 49/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2637 - accuracy: 0.8757 - val_loss: 0.2634 - val_accuracy: 0.9100
Epoch 50/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2646 - accuracy: 0.8743 - val_loss: 0.2640 - val_accuracy: 0.9067
Epoch 51/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2596 - accuracy: 0.8700 - val_loss: 0.2820 - val_accuracy: 0.9067
Epoch 52/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2622 - accuracy: 0.8757 - val_loss: 0.2683 - val_accuracy: 0.9033
Epoch 53/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2666 - accuracy: 0.8671 - val_loss: 0.2759 - val_accuracy: 0.9033
Epoch 54/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2615 - accuracy: 0.8714 - val_loss: 0.2897 - val_accuracy: 0.9033
Epoch 55/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2606 - accuracy: 0.8714 - val_loss: 0.2776 - val_accuracy: 0.9033
Epoch 56/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2641 - accuracy: 0.8700 - val_loss: 0.2816 - val_accuracy: 0.9033
Epoch 57/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2600 - accuracy: 0.8729 - val_loss: 0.2691 - val_accuracy: 0.9000
Epoch 58/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2603 - accuracy: 0.8700 - val_loss: 0.2584 - val_accuracy: 0.9067
Epoch 59/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2558 - accuracy: 0.8700 - val_loss: 0.2819 - val_accuracy: 0.9033
Epoch 60/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2573 - accuracy: 0.8757 - val_loss: 0.2771 - val_accuracy: 0.9033
Epoch 61/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2547 - accuracy: 0.8700 - val_loss: 0.2730 - val_accuracy: 0.9033
Epoch 62/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2566 - accuracy: 0.8757 - val_loss: 0.2629 - val_accuracy: 0.9033
Epoch 63/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2537 - accuracy: 0.8729 - val_loss: 0.2785 - val_accuracy: 0.9033
Epoch 64/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2528 - accuracy: 0.8700 - val_loss: 0.2705 - val_accuracy: 0.9000
Epoch 65/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2573 - accuracy: 0.8786 - val_loss: 0.2676 - val_accuracy: 0.9000
Epoch 66/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2498 - accuracy: 0.8700 - val_loss: 0.2882 - val_accuracy: 0.9067
Epoch 67/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2562 - accuracy: 0.8729 - val_loss: 0.2719 - val_accuracy: 0.9033
Epoch 68/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2482 - accuracy: 0.8757 - val_loss: 0.2952 - val_accuracy: 0.8900
Epoch 69/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2525 - accuracy: 0.8657 - val_loss: 0.2630 - val_accuracy: 0.9033
Epoch 70/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2498 - accuracy: 0.8700 - val_loss: 0.2694 - val_accuracy: 0.9033
Epoch 71/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2516 - accuracy: 0.8771 - val_loss: 0.2736 - val_accuracy: 0.9000
Epoch 72/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2447 - accuracy: 0.8729 - val_loss: 0.3048 - val_accuracy: 0.8767
Epoch 73/100
7/7 [==============================] - 0s 11ms/step - loss: 0.2529 - accuracy: 0.8671 - val_loss: 0.2534 - val_accuracy: 0.9067
Epoch 74/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2441 - accuracy: 0.8643 - val_loss: 0.2715 - val_accuracy: 0.8967
Epoch 75/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2507 - accuracy: 0.8600 - val_loss: 0.2595 - val_accuracy: 0.9000
Epoch 76/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2482 - accuracy: 0.8714 - val_loss: 0.2580 - val_accuracy: 0.9000
Epoch 77/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2442 - accuracy: 0.8757 - val_loss: 0.2645 - val_accuracy: 0.9000
Epoch 78/100
7/7 [==============================] - 0s 12ms/step - loss: 0.2451 - accuracy: 0.8643 - val_loss: 0.2595 - val_accuracy: 0.9000
Epoch 79/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2467 - accuracy: 0.8714 - val_loss: 0.2566 - val_accuracy: 0.9033
Epoch 80/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2420 - accuracy: 0.8686 - val_loss: 0.2518 - val_accuracy: 0.9067
Epoch 81/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2483 - accuracy: 0.8757 - val_loss: 0.2598 - val_accuracy: 0.9000
Epoch 82/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2408 - accuracy: 0.8714 - val_loss: 0.2454 - val_accuracy: 0.9067
Epoch 83/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2445 - accuracy: 0.8714 - val_loss: 0.2459 - val_accuracy: 0.9067
Epoch 84/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2385 - accuracy: 0.8700 - val_loss: 0.2524 - val_accuracy: 0.9067
Epoch 85/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2428 - accuracy: 0.8686 - val_loss: 0.2655 - val_accuracy: 0.8933
Epoch 86/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2491 - accuracy: 0.8643 - val_loss: 0.2688 - val_accuracy: 0.8867
Epoch 87/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2404 - accuracy: 0.8771 - val_loss: 0.2547 - val_accuracy: 0.8933
Epoch 88/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2360 - accuracy: 0.8743 - val_loss: 0.2436 - val_accuracy: 0.8933
Epoch 89/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2461 - accuracy: 0.8700 - val_loss: 0.2472 - val_accuracy: 0.9000
Epoch 90/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2394 - accuracy: 0.8657 - val_loss: 0.2601 - val_accuracy: 0.8933
Epoch 91/100
7/7 [==============================] - 0s 13ms/step - loss: 0.2407 - accuracy: 0.8686 - val_loss: 0.2505 - val_accuracy: 0.9033
Epoch 92/100
7/7 [==============================] - 0s 11ms/step - loss: 0.2354 - accuracy: 0.8771 - val_loss: 0.2451 - val_accuracy: 0.9000
Epoch 93/100
7/7 [==============================] - 0s 12ms/step - loss: 0.2363 - accuracy: 0.8714 - val_loss: 0.2422 - val_accuracy: 0.9000
Epoch 94/100
7/7 [==============================] - 0s 13ms/step - loss: 0.2456 - accuracy: 0.8614 - val_loss: 0.2637 - val_accuracy: 0.8933
Epoch 95/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2343 - accuracy: 0.8743 - val_loss: 0.2488 - val_accuracy: 0.9033
Epoch 96/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2387 - accuracy: 0.8714 - val_loss: 0.2417 - val_accuracy: 0.8900
Epoch 97/100
7/7 [==============================] - 0s 12ms/step - loss: 0.2361 - accuracy: 0.8671 - val_loss: 0.2613 - val_accuracy: 0.8933
Epoch 98/100
7/7 [==============================] - 0s 12ms/step - loss: 0.2367 - accuracy: 0.8771 - val_loss: 0.2532 - val_accuracy: 0.8933
Epoch 99/100
7/7 [==============================] - 0s 13ms/step - loss: 0.2330 - accuracy: 0.8700 - val_loss: 0.2608 - val_accuracy: 0.8933
Epoch 100/100
7/7 [==============================] - 0s 14ms/step - loss: 0.2346 - accuracy: 0.8743 - val_loss: 0.2457 - val_accuracy: 0.8933
```

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

In [ ]:

```
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 2ms/step
```

Out[ ]: [<matplotlib.lines.Line2D at 0x7fefc7899eb0>]