Booklet_2_Code_A1

August 8, 2020

1 Neuronale Netze - Beispielprogramm

1.0.1 Erweiterung des Beispielprogramms um einen Hidden Layer

1.0.2 1 - Preparations

1.1 - Imports

[2]:
```
import matplotlib.pyplot as plt
import numpy as np
import sklearn
import sklearn.datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import accuracy_score
import pandas as pd

# Display plots inline and change default figure size
import matplotlib
%matplotlib inline
matplotlib.rcParams['figure.figsize'] = (10.0, 8.0)
```

1.2 - Generating a dataset

[ ]:
```
# Generate a dataset and plot it
n          = 1000
data_seed  = 1337
split_seed = 42
test_size  = 0.25

np.random.seed(data_seed)
X, y = sklearn.datasets.make_moons(n, noise=0.25)
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral);
```

1.3 - Split the dataset

[4]:
```
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=test_size,
```

```
                                                random_state=split_seed)
```

1.0.3 2- Implementierung

2.1 - Aktivierungsfunktion (activation function) Als Aktivierungsfunktion des Output-Layers wird die logistische Funktion

$$\sigma\colon\ \mathbb{R} \to (0,1)\,,\ z \mapsto \frac{1}{1+\exp(-z)}$$

verwendet.

[5]:
```
def sigmoid(z, derivation = False):
    if derivation:
        return sigmoid(z)*(1-sigmoid(z))
    else:
        return 1/(1+np.exp(-z))

value = 1
print(sigmoid(value))
print(sigmoid(value,True))
```

```
0.7310585786300049
0.19661193324148185
```

2.2 Kostenfunktion (cost function) Als Kostenfunktion wird folgende quadratische Fehlerfunktion verwendet:

$$E(y,\widehat{\pi}) = \frac{1}{2}\sum_{k=1}^{n}(y_k - \widehat{\pi_k})^2$$

wobei $y = (y_1,\ldots,y_n)^T$ und $\widehat{\pi} = (\widehat{\pi_1},\ldots,\widehat{\pi_n})^T$.

2 Booklet Teil 2

hier startet unsere Implementierung:

[6]:
```
class NeuralNet:
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate=0.
 ↪1):
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes
        self.learning_rate = learning_rate

        # set random seed to get reproducable results
        np.random.seed(5)

        # Glorot initialisation
```

```
        self.W_input_hidden = np.random.normal(0.0,
                                               1/((self.input_nodes*self.
↪hidden_nodes+self.hidden_nodes)/2),
                                               (self.hidden_nodes, self.
↪input_nodes))

        self.W_hidden_output = np.random.normal(0.0,
                                               1/((self.hidden_nodes*self.
↪output_nodes+self.output_nodes)/2),
                                               (self.output_nodes, self.
↪hidden_nodes))

        self.bias_input = np.zeros((self.hidden_nodes, 1))

        self.bias_hidden = np.zeros((self.output_nodes, 1))

    def calculate_loss(self, inputs,labels):
        predictions = self.predict(inputs)

        # Berechnung des Kostenfunktionswertes
        cost = np.power(predictions-labels,2)
        cost = np.sum(cost)/2

        return cost


    def fit(self, inputs, label):
        label = np.array(label, ndmin=2).T
        output, hidden_outputs = self.predict(inputs, backprop=True)

        # Update weights from hidden to output layer
        output_error = output - label  #getting this from the derived squared␣
↪error loss function
        # note that "output_error*outputs*(1.0-outputs)" comes from the␣
↪derivate of sigmoid acrivation.
        gradient_weights_hidden_output = np.dot((output_error*output*(1.
↪0-output)), np.transpose(hidden_outputs))
        self.W_hidden_output += -self.learning_rate *␣
↪gradient_weights_hidden_output

        # update weights of bias_hidden
        gradient_weights_bias_hidden = output_error*output*(1.0-output)
        self.bias_hidden += - self.learning_rate * gradient_weights_bias_hidden

        # Update weights from input to hidden layer
```

```
        # first "propagate" the errors back to the hidden layer.
        hidden_errors = np.dot(self.W_hidden_output.T, (output_error*output*(1.
↪0-output)))

        # this step is the same as from the previous update. just one layer␣
↪closer to the input.
        gradient_weights_input_hidden = np.dot((hidden_errors * hidden_outputs␣
↪* (1.0 - hidden_outputs)),
                                               np.array(inputs, ndmin=2))
        self.W_input_hidden += -self.learning_rate *␣
↪gradient_weights_input_hidden

        # update weights of bias_input
        gradient_weights_bias_input = (hidden_errors * hidden_outputs * (1.0 -␣
↪hidden_outputs))
        self.bias_input = self.bias_input - self.learning_rate *␣
↪gradient_weights_bias_input

        pass

    def predict(self, inputs, backprop=False):

        inputs = np.array(inputs, ndmin=2).T

        # feedforward from input layer to hidden layer
        hidden_inputs = np.dot(self.W_input_hidden, inputs) + self.bias_input
        hidden_outputs = sigmoid(hidden_inputs)

        # feedforward from hidden layer to output layer
        output_inputs = np.dot(self.W_hidden_output, hidden_outputs) + self.
↪bias_hidden
        output_outputs = sigmoid(output_inputs)

        if backprop == True:
            # returning hidden outputs which we need in case of backpropagation
            return (output_outputs, hidden_outputs)

        return output_outputs
```

[7]:
```
# create nn
nn = NeuralNet(input_nodes=2, hidden_nodes=100, output_nodes=1, learning_rate=0.
↪1)
```

[8]:
```
# train nn
for epoch in range(15):
    for sample, sample_label, in zip(X_train, y_train):
```

```
        nn.fit(sample, sample_label)
```

[9]:
```
nn.calculate_loss(X_test, y_test)
```

[9]: 14.84704733671989

[10]:
```
# calculate the accuracy
predictions = nn.predict(X_test)
rounded_predictions = np.where(predictions > 0.5,1,0)
rounded_predictions[0]

accuracy = (rounded_predictions[0] == y_test).mean()
accuracy
```

[10]: 0.824