Statistical Machine Learning 2018
Assignment 4
Deadline: 11th of January 2019

Christoph Schmidl
s4226887
c.schmidl@student.ru.nl

Mark Beijer
s4354834
mbeijer@science.ru.nl

January 6, 2019

Exercise 1 - Logistic regression (weight 5)

Part 1 - The IRLS algorithm

Many machine learning problems require minimizing / maximizing some function $f(x)$. For this, an alternative to the familiar gradient descent technique, is the so called Newton-Raphson iterative method:

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \mathbf{H}^{-1}\nabla f(\mathbf{x}^{(n)}) \tag{1}$$

where $\mathbf{H}$ represents the Hessian matrix of second derivatives of $f(\mathbf{x})$, see Bishop, 4.3.3.

1.1.1

Derive an expression for the minimization / maximization of the function $f(x) = \sin(x)$, using the Newton-Raphson iterative optimization scheme (1), and verify (using Matlab, just up to, e.g., five iterations) how quickly it converges when starting from $x^{(0)} = 1$. What happens when you start from $x^{(0)} = -1$?
Hint: The Hessian of a 1-dimensional function $f(x)$ is just the second derivative f''. So, the Newton-Raphson iterative method reduces in 1-d to

$$x^{(n+1)} = x^{(n)} - \frac{f'(x^{(n)})}{f''(x^{(n)})} \tag{2}$$

Answer:

Expression for the minimization / maximization of the function $f(x) = \sin(x)$:

$$\begin{aligned} x^{(x+1)} &= x^{(n)} - \frac{\sin'(x^{(n)})}{\sin''(x^{(n)})} \\ &= x^{(n)} - \frac{\cos\left(x^{(n)}\right)}{-\sin\left(x^{(n)}\right)} \\ &= x^{(n)} + \frac{\cos\left(x^{(n)}\right)}{\sin\left(x^{(n)}\right)} \end{aligned}$$

```python
import numpy as np
import sympy as sp

# Exercise 1.1.1

## See also: https://docs.sympy.org/latest/tutorial/calculus.html

```

```python
def f(x):
    return np.sin(x)

def iterative_optimization(x):
    """ Using Newton-Raphson iterative optimization scheme """
    return x + (np.cos(x)/np.sin(x))

x_t = 1
n_iterations = 5

print("Starting the optimization process with x_0: {}".format(x_t))

for i in range(n_iterations):
    x_t = iterative_optimization(x_t)
    print("Step {}: {}".format(i, x_t))

# Starting the optimization process with x_0: 1
# Step 0: 1.6420926159343308
# Step 1: 1.5706752771612507
# Step 2: 1.5707963267954879
# Step 3: 1.5707963267948966
# Step 4: 1.5707963267948966

# Starting the optimization process with x_0: -1
# Step 0: -1.6420926159343308
# Step 1: -1.5706752771612507
# Step 2: -1.5707963267954879
# Step 3: -1.5707963267948966
# Step 4: -1.5707963267948966
```

If we take $x_0 = 1$, then the algorithm converges after 4 steps towards 1.5707963267948966 which is an approximation of $\frac{\pi}{2}$.
If we take $x_0 = -1$, then the algorithm converges after 4 steps towards -1.5707963267948966 which is an approximation of $-\frac{\pi}{2}$.

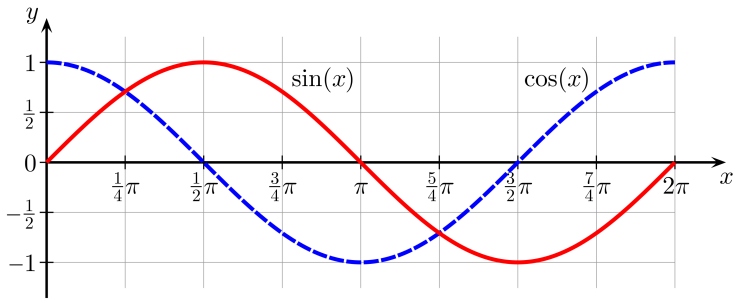


Figure 1: Sine and Cosine

When we take a look at figure 1 then we can see that the maximum and minimum of the sine function is indeed at $(x = \frac{\pi}{2}, f(x) = 1)$ and $(x = -\frac{\pi}{2}, f(x) = -1)$ with possible shiftings to the right and left based on the periodic nature of the sine function.

Note: Also see `https://en.wikipedia.org/wiki/Newton%27s_method#Applications`

1.1.2

We want to apply this method to the logistic regression model for classification (see Bishop, 4.3.2):

$$p(\mathcal{C}_1|\phi) = y(\phi) = \sigma(w^T\phi) \tag{3}$$

For a data set $\{\phi_n, t_n\}_{n=1}^N$, with $t_n \in \{0, 1\}$, using $y_n = p(\mathcal{C}_1|\phi_n)$ the corresponding cross entropy error function to minimize is

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\{t_n \ln y_n + (1 - t_n)\ln(1 - y_n)\} \tag{4}$$

With one basis function ϕ and the dummy basis function 1, the feature vector in (3) becomes $\phi = [1, \phi]^T$. The weight vector including the bias term is then also two dimensional, $\mathbf{w} = [w_0, w_1]^T$. Expressions for

the gradient $\nabla E(\mathbf{w})$ and Hessian $\mathbf{H}$ in terms of the data set are given in Bishop, eq.4.96-98. As both are implicitly dependent on the weights $\mathbf{w}$, they have to be recalculated after each step: hence this is known as the 'Iterative Reweighted Least Squares' algorithm.

Consider the following data set: $\{\phi_1, t_1\} = \{0.3, 1\}, \{\phi_2, t_2\} = \{0.44, 0\}, \{\phi_3, t_3\} = \{0.46, 1\}$ and $\{\phi_4, t_4\} = \{0.6, 0\}$, and initial weight vector $\mathbf{w}^{(0)} = [1.0, 1.0]^T$.

Show using e.g. a Matlab implementation that for this situation the IRLS algorithm converges in a few iterations to the optimal solution $\hat{\mathbf{w}}^T \approx [9.8, -21.7]$, and show that this solution corresponding to a decision boundary $\phi = 0.45$ in the logistic regression model. (The IRLS algorithm should take about five lines of Matlab code inside a loop + initialization).

Answer:

$$p(\mathcal{C}_1|\phi) = y(\phi) = \sigma(w^T\phi) = \frac{1}{1 + \exp(-(w^T\phi))}$$

As stated in Bishop, page 208, eq 4.99 and 4.100: The Newton-Raphson update formula for the logistic regression model becomes

$$\begin{aligned}\mathbf{w}^{new} &= \mathbf{w}^{old} - (\mathbf{\Phi}^T\mathbf{R}\mathbf{\Phi})^{-1}\mathbf{\Phi}^T(\mathbf{y} - \mathbf{t}) \\ &= (\mathbf{\Phi}^T\mathbf{R}\mathbf{\Phi})^{-1}\{\mathbf{\Phi}^T\mathbf{R}\mathbf{\Phi}\mathbf{w}^{(old)} - \mathbf{\Phi}^T(\mathbf{y} - \mathbf{t})\} \\ &= (\mathbf{\Phi}^T\mathbf{R}\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{R}\mathbf{z}\end{aligned}$$

where $\mathbf{z}$ is an N-dimensional vector with elements

$$\mathbf{z} = \mathbf{\Phi}\mathbf{w}^{old} - \mathbf{R}^{-1}(\mathbf{y} - \mathbf{t})$$

The following Python code converges after 6 iterations towards $\hat{\mathbf{w}}^T = [9.78227684, -21.73839298]$ which is pretty close to the optimal solution.

```
# Exercise 1.1.2

def sigmoid(x):
    """ The standard logistic function. np.exp also accepts arrays"""
    return 1.0 / (1 + np.exp(-x))

def gradient_of_error(phi, y, t):
    """ Gradient (first-order derivatives) of the error function, see Bishop page 207, eq.
    4.96 """
    return np.dot(phi.T, y - t)

def hessian_of_error(phi, y):
    """ Hessian (second-order derivatives) of the error function, see Bishop page 207, eq.
    4.97 """
    R = np.diag(np.ravel(y * (1 - y)))
    return np.dot(phi.T, np.dot(R, phi))

def cross_entropy_error(y, t):
    # Implementation of cross entropy error = E(w)
    result = 0
    for n in range(0, t.shape[0]):
        result += t[n] * np.log(y[n]) + (1 - t[n]) * np.log(1 - y[n])
    return -result

w = np.array([ [1.0], [1.0] ]) # two-dimensional weight vector
x = np.array([0.3, 0.44, 0.46, 0.6])
t = np.array([ [1], [0], [1], [0] ]) # targets
phi = np.array([ [1, x_element] for x_element in x ]) # feature vector

for i in range(10):
    y = sigmoid(np.dot(phi, w)) # class estimates

    current_gradient = gradient_of_error(phi, y, t)
    current_hessian = hessian_of_error(phi, y)
```

```
33
34      w = w - np.dot(np.linalg.inv(current_hessian), current_gradient)
35
36      print("Iteration {}: \nphi={} \ny={}, \ncurrent_gradient={}, \ncurrent_hessian={}, \nw
        ={}\n".format(
37          i, phi, y, current_gradient, current_hessian, w))
38
39  # Converges after 6 iterations
40  # w=[[  9.78227684][-21.73839298]]
```

In order to show that $\phi = 0.45$ is indeed the decision boundary, we show that the probability of a data point with $\phi = 0.45$ is the same for both classes. That means that both classes have a probability of 0.5 since $p(C_1|\phi) + p(C_2|\phi) = 1$.

$$p(C_1|\phi) = y(\phi) = \sigma(\boldsymbol{w}^T\phi) = \sigma(\begin{pmatrix} 9.78227684 \\ -21.73839298 \end{pmatrix}^T \begin{pmatrix} 1 \\ 0.45 \end{pmatrix}) \approx 0.50$$

We can also prove that in a slightly different way using the optimal solution:

$$\begin{aligned}
p(C_1|\phi) &= \frac{1}{1+\exp(-w^T\phi)} \\
0.5 &= \frac{1}{1+\exp(-[9.8,-21.7]^T[1,\phi])} \\
0.5 &= \frac{1}{1+\exp(-9.8+21.7\phi)} \\
2 &= 1+\exp\{-9.8+21.7\phi\} \\
1 &= \exp(-9.8+21.7\phi) \\
\ln(1) &= -9.8+21.7\phi \\
0 &= -9.8+21.7\phi \\
21.7\phi &= 9.8 \\
\phi &= \frac{9.8}{21.7} \approx 0.45
\end{aligned}$$

Useful internet resources which helped solving this exercise:

- Second Order Optimization - The Math of Intelligence 2: `https://www.youtube.com/watch?v=UIFMLK2nj_w`
- Logistic Regression - The Math of Intelligence (Week 2): `https://www.youtube.com/watch?v=D8alok2P468`
- `https://thelaziestprogrammer.com/sharrington/math-of-machine-learning/solving-logreg-newtons-method`
- `http://cs229.stanford.edu/notes/cs229-notes1.pdf`
- `https://www.stat.cmu.edu/~cshalizi/350/lectures/26/lecture-26.pdf`
- `https://www.stat.cmu.edu/~cshalizi/402/lectures/14-logistic-regression/lecture-14.pdf`
- `https://www.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/quadratic-approximations/a/the-hessian`
- `https://web.stanford.edu/group/sisl/k12/optimization/#!index.md`
- `http://openclassroom.stanford.edu/MainFolder/DocumentPage.php?course=MachineLearning&doc=exercises/ex4/ex4.html`

Part 2 - Two-class classification using logistic regression

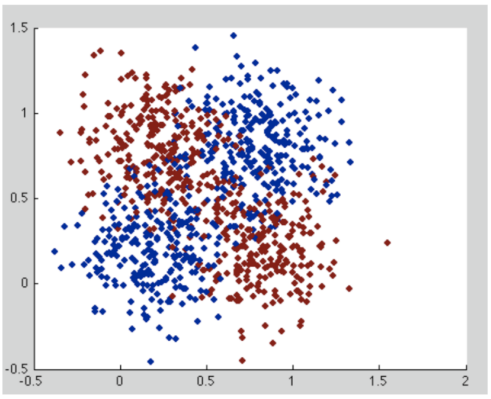


Figure 2: Two class data for logistic regression

Two-class classification using logistic regression in the IRLS algorith,. Data consists of 1000 pairs $\{x_1, x_2\}$ with corresponding class labels $\mathcal{C}_1 = 0$ or $\mathcal{C}_2 = 1$. Load it into Matlab using

```
data = load('a010_irlsdata.txt', '-ASCII');
X = data(:,1:2); Y = data(:,3);
```

1.2.1

Make a scatter plot of the data, similar to Figure 2. (Have a look at Matlab file `a010plotideas.m` in Brightspace for some ideas to make such a scatter plot and the plots later on.) Do you think logistic regression can be a good approach to classification for this type of data? Explain why.

Answer:

The following Python code generates figure 3:

```python
# Exercise 1.2.1

data = np.loadtxt("data/a010_irlsdata.txt")

print("Shape of loaded data: {}".format(data.shape))
X = data[:,0:2] # feature vector
print("Shape of feature vector: {}".format(X.shape))

Y = data[:,2:3] # label vector
print("Shape of label vector: {}".format(Y.shape))

# Plot
plt.scatter(X[:,0], X[:,1], c=Y.ravel())
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('figure_1_2_1.png')
plt.show()
```

We do not think that Logistic Regression is a good approach to classify this type of data because the data does not seem to be linearly separable.
The decision boundary drawn by Logistic regression is a linear combination of features and weights denoted by $w_0 + \sum_i w_i X_i$ and therefore is only able to draw a hyperplane through the space to predict one class of each side respectively. The presented data cannot be divided by such a plane except when it would be possible to apply kernel methods like in Support Vector Machines which maps the problem into a new space.

See also: `https://homes.cs.washington.edu/~marcotcr/blog/linear-classifiers/`

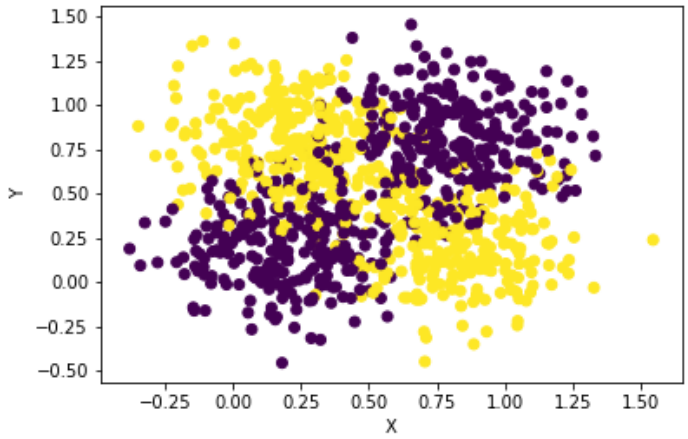


Figure 3: Scatter plot of two class data of exercise 1 - Part 2

1.2.2

Modify the Iterative Reweighted Least Squares algorithm from part 1 to calculate the optimal weights for this data. Use again a dummy basis function. Initialize with the weight vector $\mathbf{w}^T = [0, 0, 0]$. With these initial weights, what are the class probabilities according to the logistic regression model (i.e., before optimization)?

Answer:

$$
\begin{aligned}
p(C_1|\phi) &= \frac{1}{1 + \exp(-w^T \phi)} \\
&= \frac{1}{1 + \exp(-[0.0, 0.0, 0.0]^T [1, \phi, \phi])} \\
&= \frac{1}{1 + \exp([0.0, 0.0, 0.0])} \\
&= \frac{1}{1 + \exp(0)} \\
&= \frac{1}{1 + 1} \\
&= \frac{1}{2}
\end{aligned}
$$

According to the logistic regression model with the weight vector $\mathbf{w}^T = [0, 0, 0]$, both classes have a probability of 0.5.

1.2.3

Run the algorithm. Make a scatter plot of the data, similar to figure 2, but now with color that represent the data point probabilities $P(C = 1, X_n)$ according to the model after optimization. Compare the cross entropy error with the initial value. Did it improve? Much? Explain your findings.

Answer:

The following Python code generates figure 4:

```
# Exercise 1.2.3

w = np.array([[0.0], [0.0], [0.0]]) # three-dimensional weight vector
x = X # data
t = Y # targets/labels, has to look like np.array([ [1], [0], [1], [0] ])

print("Shape of x: {}".format(x.shape[0]))
print("Shape of t: {}".format(t.shape[0]))

```

```
phi = np.array([ [1, x_element[0], x_element[1]] for x_element in x ]) # feature vector

y = sigmoid(np.dot(phi, w))
print("Cross entropy error before optimization: {}".format(cross_entropy_error(y, t)))

for i in range(0, 1000):
    y = sigmoid(np.dot(phi, w)) # class estimates
    current_gradient = gradient_of_error(phi, y, t)
    current_hessian = hessian_of_error(phi, y)

    w = w - np.dot(np.linalg.inv(current_hessian), current_gradient)

print("Cross entropy error after optimization: {}".format(cross_entropy_error(y, t)))
print("Weights: {}".format(w))

# Normalised [0,1]
y = (y - np.min(y))/np.ptp(y)

cmap = plt.get_cmap('Blues')
plt.scatter(x[:, 0], x[:, 1], c = np.squeeze(cmap(y)))
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('figure_1_2_3.png')
plt.show()

# Cross entropy error before optimization: [693.14718056]
# Cross entropy error after optimization: [692.96935948]
# Weights: [[ 0.00440664]
# [-0.02139153]
# [-0.04930069]]
```

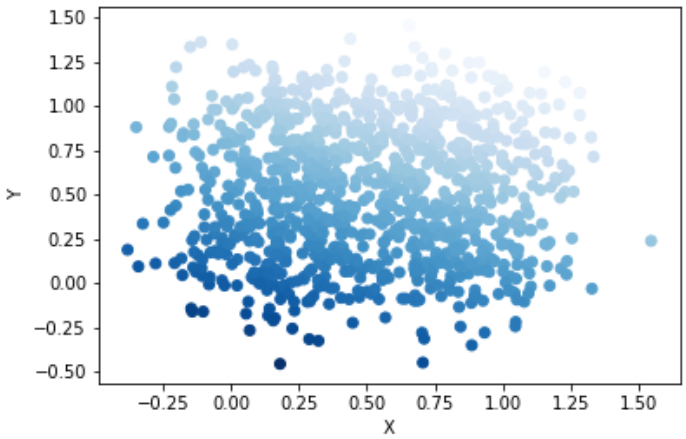


Figure 4: Scatter plot of two class data with class probabilities

The cross entropy error only improved slightly from 693.14718056 to 692.96935948.

We think the problem lies within the nature of the data that it is not linearly separable which we can also see in the transparency/probabilities in figure 4. The decision boundary seems to be drawn through the middle (linearly) although the data is not seperable that way. Based on the fact that only linear basis functions are used for this non linearly separable data explains that outcome. A non-linear basis function would probably yield better results.

1.2.4

Introduce two Gaussian basis functions as features ϕ_1, ϕ_2, similar to Bishop, fig.4.12. Use identical isotropic covariance matrices $\Sigma = \sigma^2 I$ with $\sigma^2 = 0.2$, and center the basis functions around $\mu_1 = (0, 0)$ and $\mu_2 = (1, 1)$. Make a scatter plot of the data in the feature domain. Do you think logistic regression can be a good approach to classification with these features? Explain why.

Answer:

The following Python code generates figure 5:

```
# Exercise 1.2.4

# Use identical, isotropic covariance matrices Sigma = sigma^2 * I with sigma^2 = 0.2
# and center the basis functions around mu_1 = (0,0) and mu_2 = (1,1)

def gaussian_basis_function(data, origin, variance):
    sigma = variance * np.identity(np.array(data).shape[0])
    a = (data-origin).T
    normalizer = 1.0 / (2*np.pi)**(a.shape[0]/2) * 1.0 / np.linalg.det(sigma)**(1/2)
    return normalizer * np.exp(-(1.0/2)*np.dot(a.T, np.dot(inv(sigma), a)))

phi = np.array([[1, gaussian_basis_function(xx, (0, 0), 0.2), gaussian_basis_function(xx,
    (1, 1), 0.2)] for xx in x])

plt.scatter(phi[:, 1], phi[:, 2], c = t.squeeze())
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('figure_1_2_4.png')
plt.show()
```

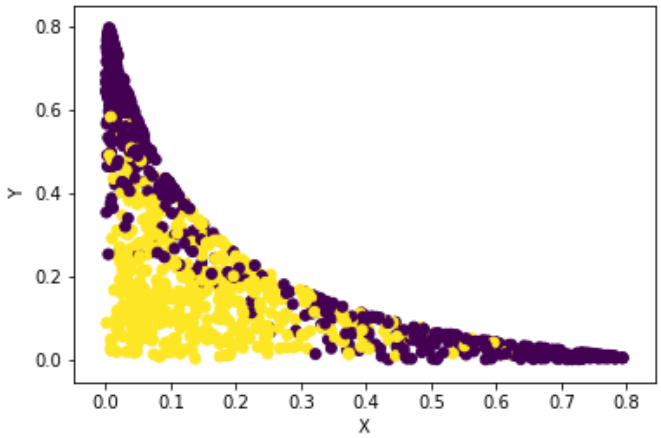


Figure 5: Scatter plot of two class data in new feature space using gaussian basis functions

The new feature space is a lot more suitable to a linear classifier because a linear decision boundary can be drawn which separates most of the two classes from each other, although not every point seems to be clearly separable. Nevertheless, this makes logistic regression an appropriate choice.

1.2.5

Modify the IRLS algorithm to use the features $\{\phi_1, \phi_2\}$ and the dummy basis function. Initialize with the weight vector $\mathbf{w}^T = [0, 0, 0]$.
Run the algorithm. Make a scatter plot of the data, similar to Figure 2, but now with colors that represent the data point probabilities $P(C = 1|X_n)$ according to this second model (after optimization). Compare the cross entropy error with the initial value. Did it improve? Much? Explain your findings.

Answer:

The following Python code generates figure 6:

```
# Exercise 1.2.5

y = sigmoid(np.dot(phi, w))
print("Cross entropy error before optimization: {}".format(cross_entropy_error(y, t)))

for i in range(0, 1000):
    y = sigmoid(np.dot(phi, w)) # class estimates
    current_gradient = gradient_of_error(phi, y, t)
    current_hessian = hessian_of_error(phi, y)

    w = w - np.dot(np.linalg.inv(current_hessian), current_gradient)
```

```
print("Cross entropy error after optimization: {}".format(cross_entropy_error(y, t)))
print("Weights: {}".format(w))

# Normalised [0,1]
y = (y - np.min(y))/np.ptp(y)

cmap = plt.get_cmap('Blues')
plt.scatter(x[:, 0], x[:, 1], c = np.squeeze(cmap(y)))
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('figure_1_2_5.png')
plt.show()

# Cross entropy error before optimization: [690.64901362]
# Cross entropy error after optimization: [346.50408046]
# Weights: [[  7.10834887]
# [-15.42143347]
# [-15.53830921]]
```

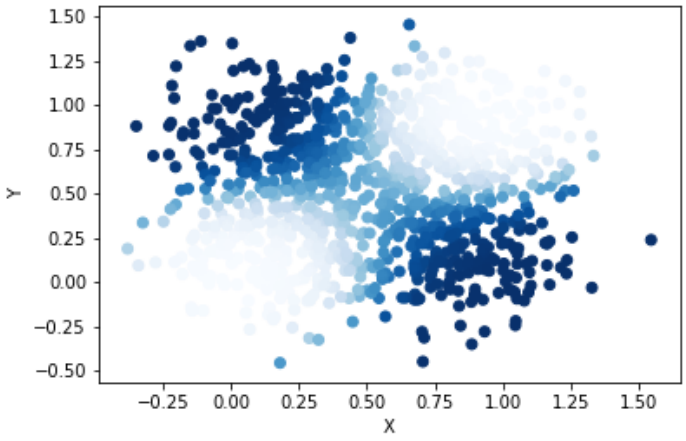


Figure 6: Scatter plot of two class data an their probabilities in new feature space using gaussian basis functions

The cross entropy error improved significantly to 346.50408046. Figure 6 shows that the gaussian basis functions are able to separate a big amount of the data at its origin of (0,0) and (1,1) and only data points in the middle have uncertain probabilities around 0.5. The cross entropy error would probably rise again if the origin of the gaussian basis functions shifted towards other origins.

Exercise 2 - Neural network regression (weight 5)

We train a neural network using backpropagation, to learn to mimic a 2D multimodal probability density. First, we implement the network and test its regression capabilities on a standard Gaussian; then we train it on the real data set. Visualization of the network output plays an important role in monitoring the progress.

2.1

Create a plot of an isotropic 2D Gaussian $y = 3 \cdot \mathcal{N}(\mathbf{0}|\frac{2}{5}\mathbf{I}_2)$ centered at the origin using the `meshgrid()`, `mvnpdf()` and `surf()` functions. Sample the density at 0.1 intervals over the range $[-2, 2] \times [-2, 2]$ and store the data in colum vector variables $\mathbf{X}$ (2D) and $\mathbf{Y}$ (1D).

Answer:

This can be done by using the following code:

```
def Gaus(X):
    FirstPart = 1/np.power(2*np.pi,len(X)/2)*1/(np.power(np.linalg.det(Sigma),0.5))
    Exp = -(1/2)*np.matmul(np.transpose(X-Mu),np.matmul(np.linalg.inv(Sigma),X-Mu))
    return FirstPart*np.exp(Exp)
```

```
5
6  def Plot(X,Y,Name):
7      x,y = np.hsplit(X,2)
8      x = np.reshape(x,(41,41))
9      y = np.reshape(y,(41,41))
10     z = np.reshape(Y,(41,41))
11     fig = plt.figure(Name)
12     ax = fig.gca(projection='3d')
13     surf = ax.plot_surface(x, y, z, cmap=cm.coolwarm,linewidth=1, antialiased=True)
14     fig.colorbar(surf, shrink=0.5, aspect=5,pad=-0.07)
15     ax.view_init(azim=-120,elev = 70)
16     ax.set_zlabel('\nY   ',fontsize=50)
17     ax.set_ylabel('\n$X_1$   ',fontsize=50)
18     ax.set_xlabel('\n\n$X_0$',fontsize=50)
19
20 Sigma = (2/5)*np.identity(2)
21 Mu = np.zeros(2)
22 x = y = np.arange(-2,2+0.1,0.1)
23 x,y = np.meshgrid(x,y)
24 X = []
25 for i in range(len(x)):
26     for j in range(len(x[0])):
27         X.append(np.array([x[i][j],y[i][j]]))
28 X = np.array(X)
29 Y = np.array([Gaus(x) for x in X])
```

The function "Gaus"(line 1) simply follows the definition of the multi-variable Gaussian as given at page 78 of Bishop:

$$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{(2\pi)^{\frac{D}{2}}}\frac{1}{|\Sigma|^{\frac{1}{2}}}e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)} \tag{5}$$

For plotting, we need to reshape it because this is the way that matplotlib plots, it takes 2 2D arrays for X and Y and 1 2D array for z. The other lines create the colorbar and axis labels. The construction with the for-loop ensures that X contains all points required uniquely. The plot of the Gaussian can be seen at figure 7:

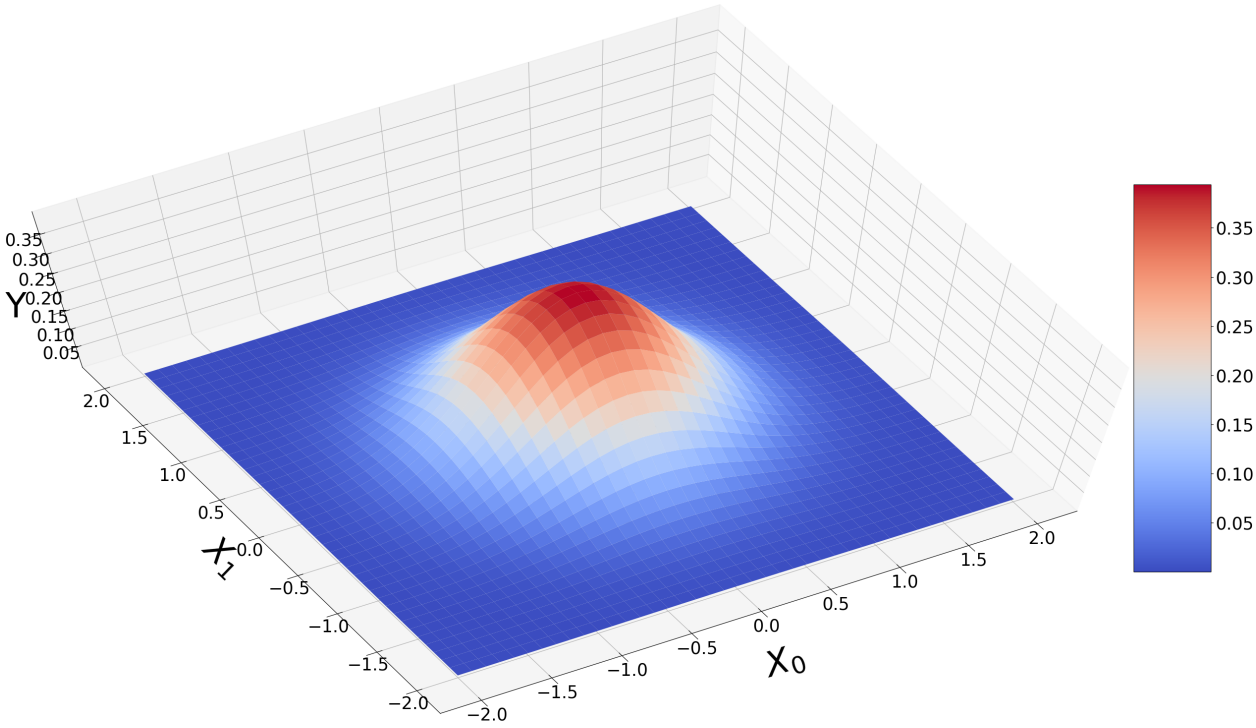


Figure 7: Gaussian function $Y = 3 \cdot \mathcal{N}(\mathbf{0}|\frac{2}{5}\mathbf{I}_2)$ graphed.

2.2

Implement a 2-layer neural network with $D = 2$ input nodes, $K = 1$ output nodes and M hidden nodes in the intermediate layer that can be trained using a sequential error backpropagation procedure, as described

in Bishop 5.3. Use $tanh(\cdot)$ activation functions for the hidden nodes and a linear activation function (regression) for the output node. Introduce appropriate weights and biases, and set the learning rate parameter $\eta = 0.1$. Initialize the weights to random values in the interval $[-0.5, 0.5]$. Plot a 2D graph of the initial output of the network over the same $[-2, 2] \times [-2, 2]$ grid as the Gaussian (again using `surf()`).

Answer:

For the network I created a class in Python:

```
class NeuralNW():
    def __init__(self,X,Y,eta):
        self.w1 = np.random.random_sample((D+1,M))-0.5
        self.w2 = np.random.random_sample(M+1)-0.5
        self.X = X
        self.Y = Y
        self.TrainX = []
        for x in self.X:
            self.TrainX.append(np.array([1,x[0],x[1]]))
        self.TrainX = np.array(self.TrainX)

    def setNetworkOnce(self,Num):
        self.HiddenLayer = np.tanh(np.dot(self.w1[0], self.TrainX[Num][0]) + np.dot(self.w1
    [1] , self.TrainX[Num][1]) + np.dot(self.w1[2],self.TrainX[Num][2]))
        self.HiddenLayer = np.insert(self.HiddenLayer,0,1)
        self.Output = np.sum(self.HiddenLayer*self.w2)

    def TrainNetwork(self,Num):
        self.setNetworkOnce(Num)
        Delta2 = self.Output - Y[Num]
        Delta1 = (1-np.square(self.HiddenLayer))*self.w2*Delta2
        Der2 = Delta2*self.HiddenLayer
        Delta1 = Delta1[1:]
        Der1 = np.array([Delta1*self.TrainX[Num][0],Delta1*self.TrainX[Num][1] , Delta1*self
    .TrainX[Num][2]])
        self.w2 -= eta*Der2
        self.w1 -= eta*Der1

    def TrainNetworkAll(self):
        for i in range(len(self.X)):
            self.TrainNetwork(i)

    def PlotOutput(self,Name):
        Out = []
        for i in range(len(X)):
            self.setNetworkOnce(i)
            Out.append(self.Output)
        Out = np.array(Out)
        Plot(self.X,Out,Name)
```

Initialization

When it is initialized it creates two sets of weights. The first (w1) are the weights form the inputs to the hidden layer. There are D inputs + 1 bias, so there are (D+1) sets of weights leading to the M hidden layers. This I represent in a two dimensional (D+1,M) matrix, where w1[0][5] stands for the weight between x_0 (bias) and z_5, of the hidden layer. The hidden layer is connected to the one output by M+1 weights, since we have another bias z_0. The fandom_sample function gives a random number between [0,1] so I lower it by 0.5 to make it between [-0.5,0.5]. Then it stores the X and Y values and for convenience creates an array "self.X" which already has x_0 as bias, which is always 1.

Set network

To run something through the model you can set all the layers of the network, including the output, by calling the "setNetworkOnce" function. This runs the nth (given by "Num") entry of the X array. The bias is set at 1 using "np.insert(self.HiddenLayer,0,1)".

TrainNetwork & TrainNetworkAll

This will run the nth point (given by Num) of the vector X and set the weights according to the 'simple example' given at page 245 of Bishop. TrainNetworkAll will call the function TrainNetwork for all points in X.

PlotOutput

This will put every point in X through the network and plot it's output.
The initial output is for M = 8 can be seen in the following figure 8:

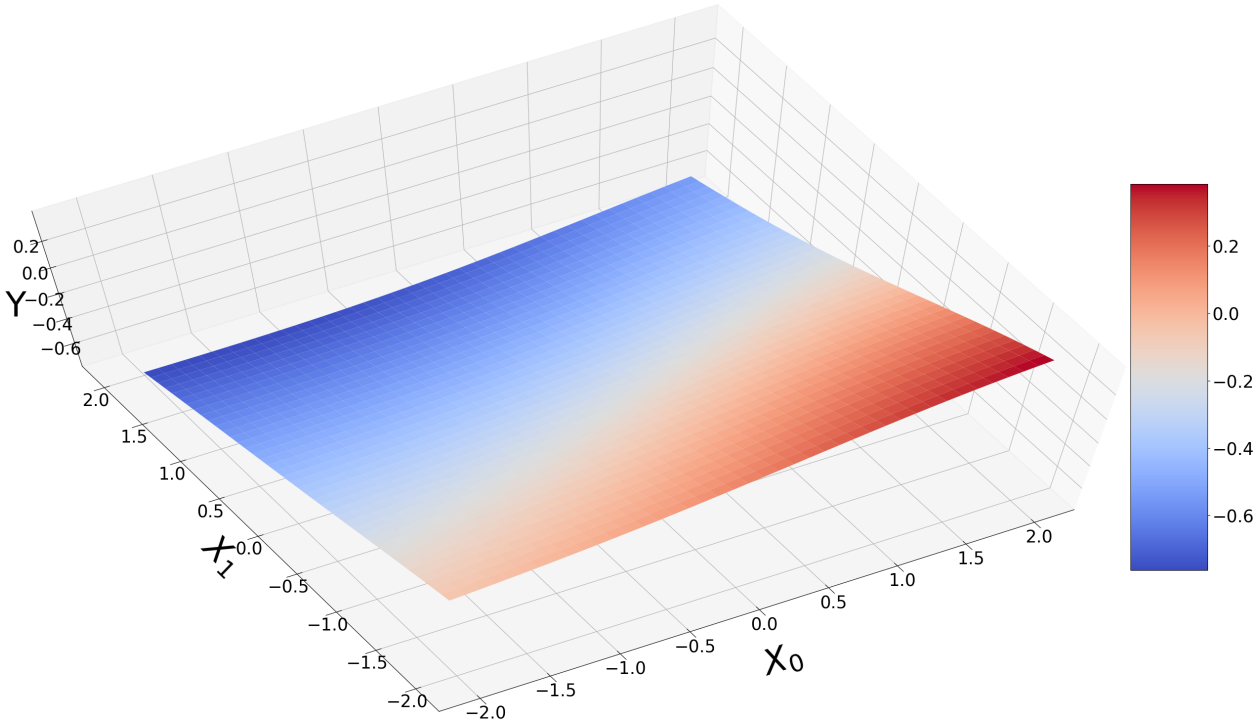


Figure 8: Inital output of the network for m = 8.

2.3

Train the network for $M = 8$ hidden nodes on the previously stored $\mathbf{X}$ and $\mathbf{Y}$ values (the $\{x_1, x_2\}$ input coordinates and corresponding output probability density y), by repeatedly looping over all datapoints and updating the weights in the network after each point. Repeat for at least 500 complete training cycles and monitor the progress of the training by plotting the output of the network over the $\mathbf{X}$ grid after each full cycle. Verify the output starts to resemble the Gaussian density after some 200 cycles (all be it with lots of 'wobbles').

Answer:

The code I used for this is:

```
eta = 0.1
D = 2
M = 8
Network = NeuralNW(X,Y,eta)
Now = time()

Network.PlotOutput("First")

for i in range(2000):
    if (i == 199):
        Network.PlotOutput("Second")
        print(time()-Now)
    Network.TrainNetworkAll()

Network.PlotOutput("Third")
print(time()-Now)
```

First we create an instance of the Network class. Then we plot the initial output and run it for some iterations, this case, 200. Then we plot the final output and the time it took to run the model.
200 iterations took me around 19.7 seconds and created the follow figure 9:

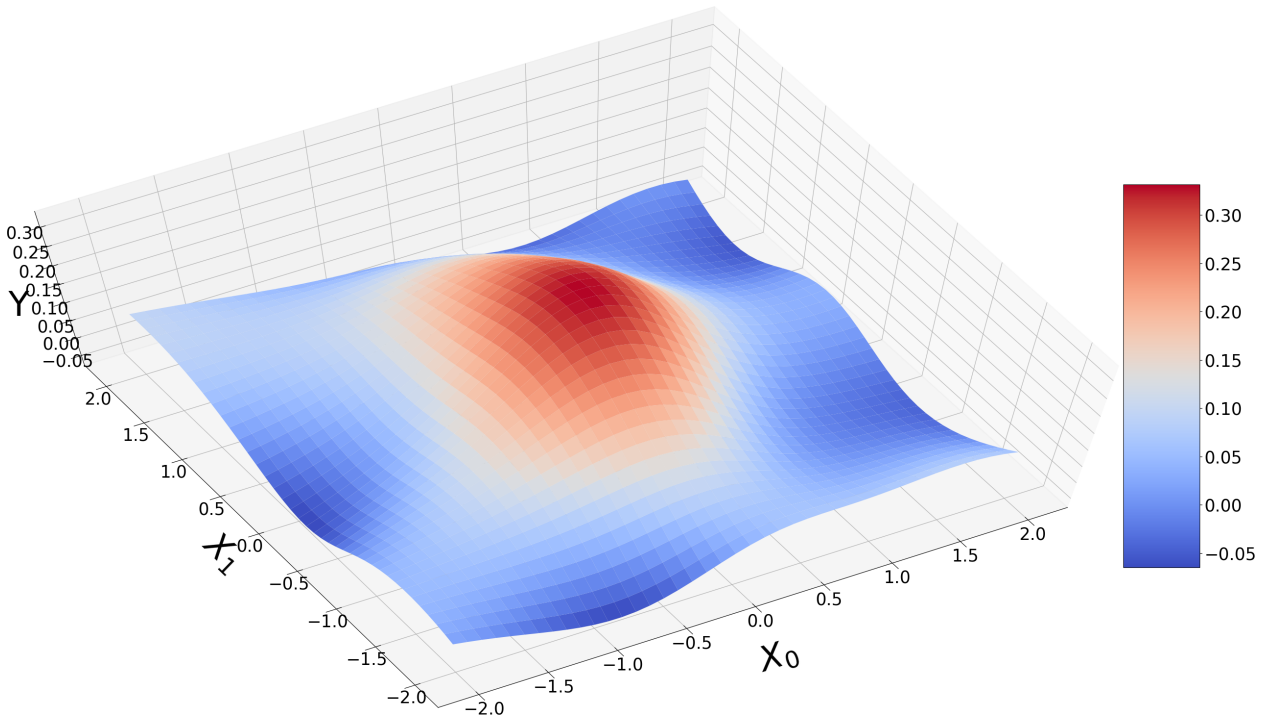


Figure 9: Output after 200 iterations of training.

Then we ran it for 2000 iterations and the results you can see at figure 10

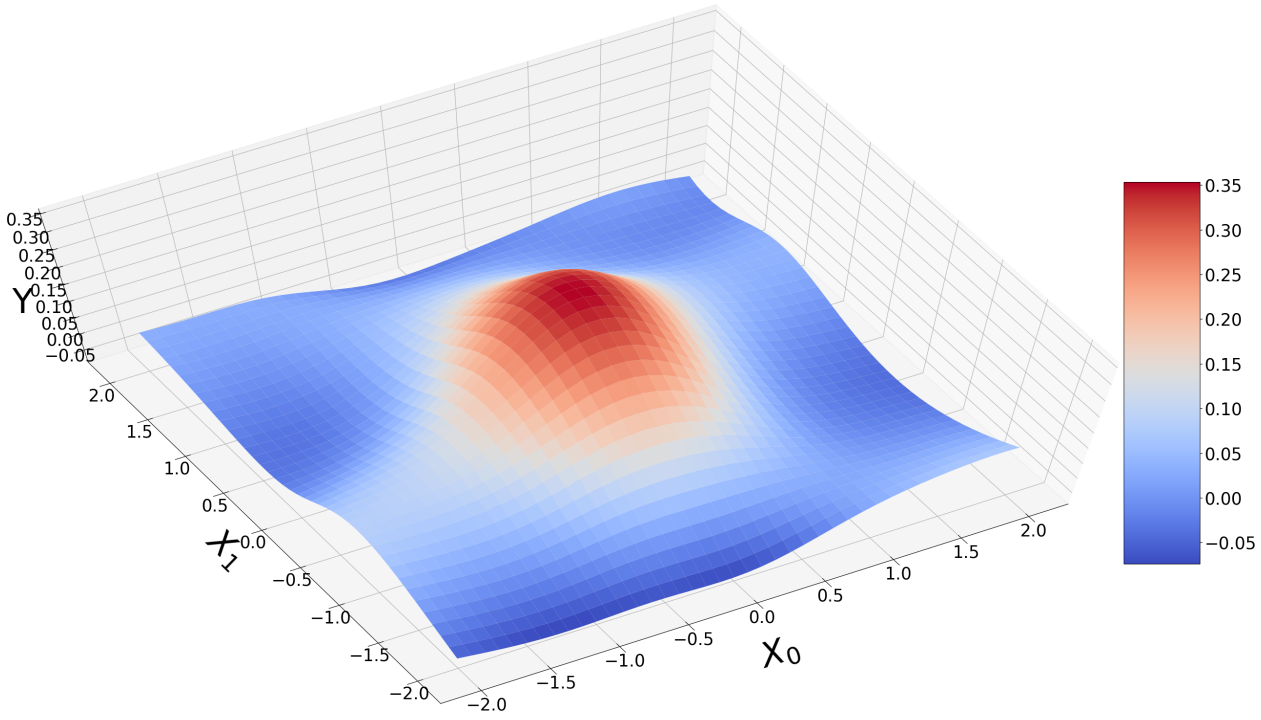


Figure 10: Output after 2000 iterations of training.

2.4

Permute the **X** and **Y** arrays to a random order using the `randperm()` function, keeping corresponding x and y together. Repeat the network training session using this randomized data set. Verify that convergence is now much quicker. Can you understand why? Try out the effect of different numbers of hidden nodes, different initial weights and different learning rates on speed and quality of the network training. Explain your results. **Answer:**

We permute by doing the following:

```
    def TrainNetworkRand(self):
        for i in np.random.permutation(len(self.X)):
            self.TrainNetwork(i)
```

This is a function of the NeuralNetwork class. Instead of training every data point from start to finish this randomizes the order, which is equal to randomizing X and Y. (It's a little bit better since it randomizes every time it's training)
We get an indication of the convergence by using the following error function:

```
    def Distance(self):
        Out = []
        for i in range(len(X)):
            self.setNetworkOnce(i)
            Out.append(self.Output)
        Out = np.array(Out)
        return np.sum(np.abs(self.Y - Out))
```

This takes the output for every point and calculates how far it is from the target, and sums it. The following code we used to show the convergence for a random permuation is indeed faster:

```
NetworkRand = NeuralNW(X,Y,eta)
for i in range(20):
    NetworkRand.TrainNetworkRand()
print(NetworkRand.Distance())

NetworkNorm = NeuralNW(X,Y,eta)
for i in range(20):
    NetworkNorm.TrainNetworkAll()
print(NetworkNorm.Distance())
```

A typical result would be something like 50 for the randomized training and 100-250 for the sequential training. By choosing random points the difference between the output and the desired output is higher, therefore more training can be done. With the desired output being close to the target not much training will be done since the gradient is not as high.

Different parameters

We've tested the convergence by choosing different parameters and looking at the Error after 100 iterations of random training. First we've tested for the amount of nodes M using the following code:

```
ErrorsM = []
Mlist = np.linspace(1,50,50,dtype=np.int)

for M in Mlist:
    print(M)
    NetworkTestM = NeuralNW(X,Y,eta)
    for i in range(100):
        NetworkTestM.TrainNetworkRand()
    ErrorsM.append(NetworkTestM.Distance())

plt.figure("Erors M")
plt.plot(Mlist,ErrorsM)
plt.xlabel('M')
plt.ylabel('Error')

plt.figure("Time M")
plt.plot(Mlist,TimeList)
plt.xlabel('M')
plt.ylabel('t(s)')
```

This creates an list of 1,2,...,50, to test Neural networks with these amount of hidden layers. Then it trains the network and append the error. It also keeps track of how long the training took. Finally it plots the error and the execution times. The result can be viewed in figure 11.

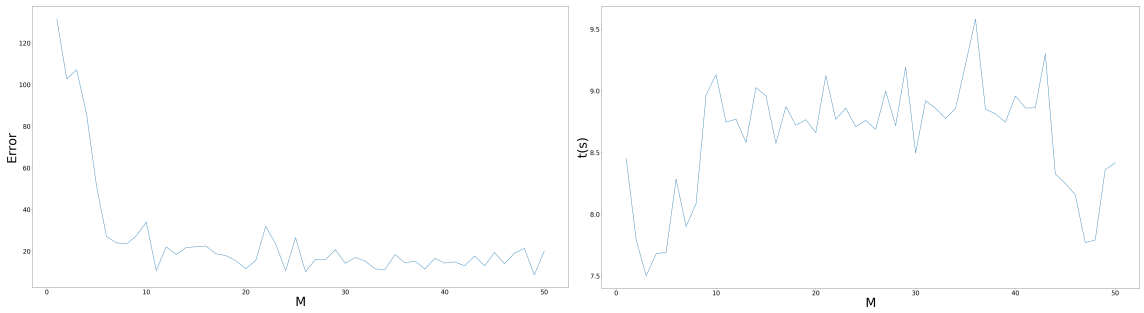


Figure 11: The error and time for various amount of hidden layers M. The time taken is fairly constant and the error seems to be constant after 10-20 hidden layers. So it's best to take around 20 hidden layers.

We also tested the effect of different η, for which we took a logarithmically, meaning that the log of the different $\eta's$ reveals a linear scale.
We used the following code:

```
Errorseta = []
M = 8
EtaLog = np.linspace(-4,2,50)

Etas = np.power(10,EtaLog)

for eta in Etas:
    print(eta)
    NetworkTesteta = NeuralNW(X,Y,eta)
    for i in range(100):
        NetworkTesteta.TrainNetworkRand()
    Errorseta.append(NetworkTesteta.Distance())

plt.figure("Errors eta")
plt.plot(Etas,Errorseta)
plt.xlabel('$\eta$')
plt.ylabel('Error')
```

So we took 50 different values for η between 10^-4 and 10^2. The plot we got from this calculation is figure 12:

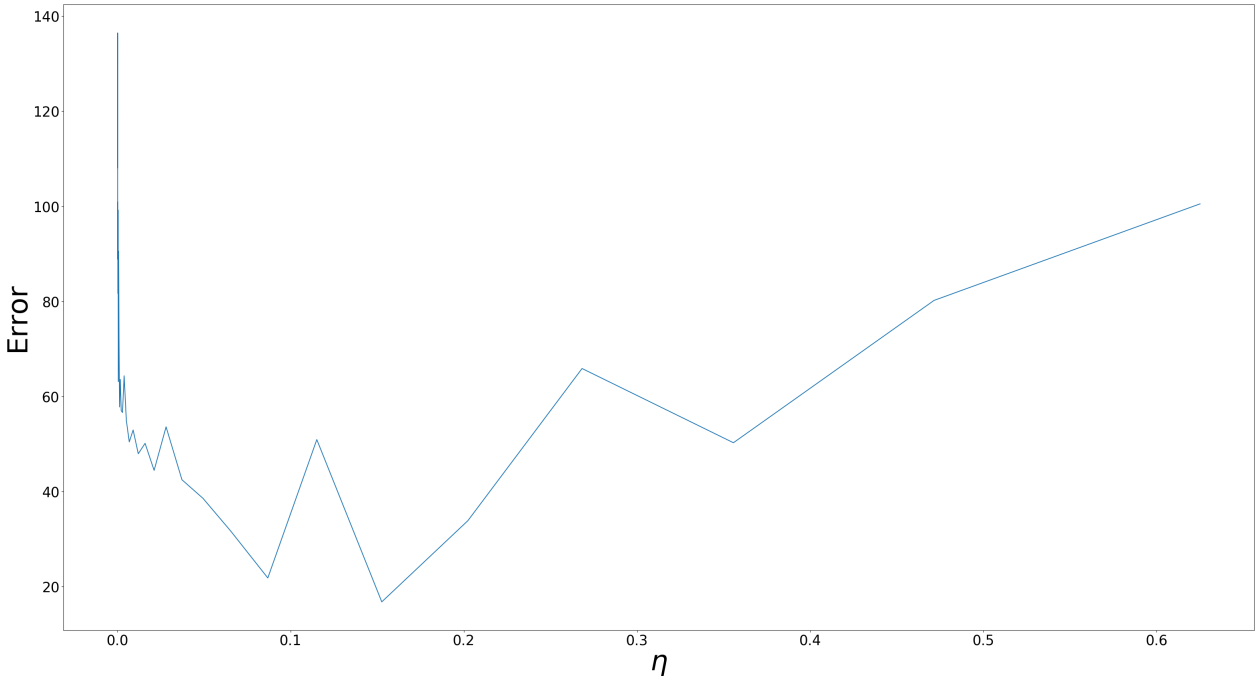


Figure 12: Errors for different values of η. A value around 0.1-1.5 seems to be the best.

Finally to test the influence of the starting values of the weights we we took a random value between $[-0.5+i, 0.5+i]$ with i varying from -5 to 5. So we can test if a lower random weight or a higher random weight would be beneficial. We did it using the following code:

```
ErrorW = []
Weights = []
eta = 0.1
for i in range(-5,5):
    print(i)
    Weights.append([])
    for j in range(10):
        w1 = np.random.random_sample((D+1,M))-0.5+i
        w2 = np.random.random_sample(M+1)-0.5+i
        Weights[-1].append((w1,w2))


for partWeight in Weights:
    ErrorW.append(0)
    for w1,w2 in partWeight:
        NetworkTestW = NeuralNW(X,Y,eta)
        NetworkTestW.setWeights(w1,w2)
        for i in range(100):
            NetworkTestW.TrainNetworkRand()
        ErrorW[-1] += NetworkTestW.Distance()

ErrorW = np.array(ErrorW)/10

WPlot = range(-5,5)
plt.figure("Errors in W")
plt.plot(WPlot,ErrorW)
plt.xlabel('Inital weights bias')
plt.ylabel('Error')

    def setWeights(self,w1,w2):
        self.w1 = w1
        self.w2 = w2
```

We tested very different w 10 times to reduce the bias for taking random values for w. The number i we called the 'bias'. The code from 30-32 is part of the Neuralnetwork class and is used to set the weights. The plot we got from this can be seen at figure 13.

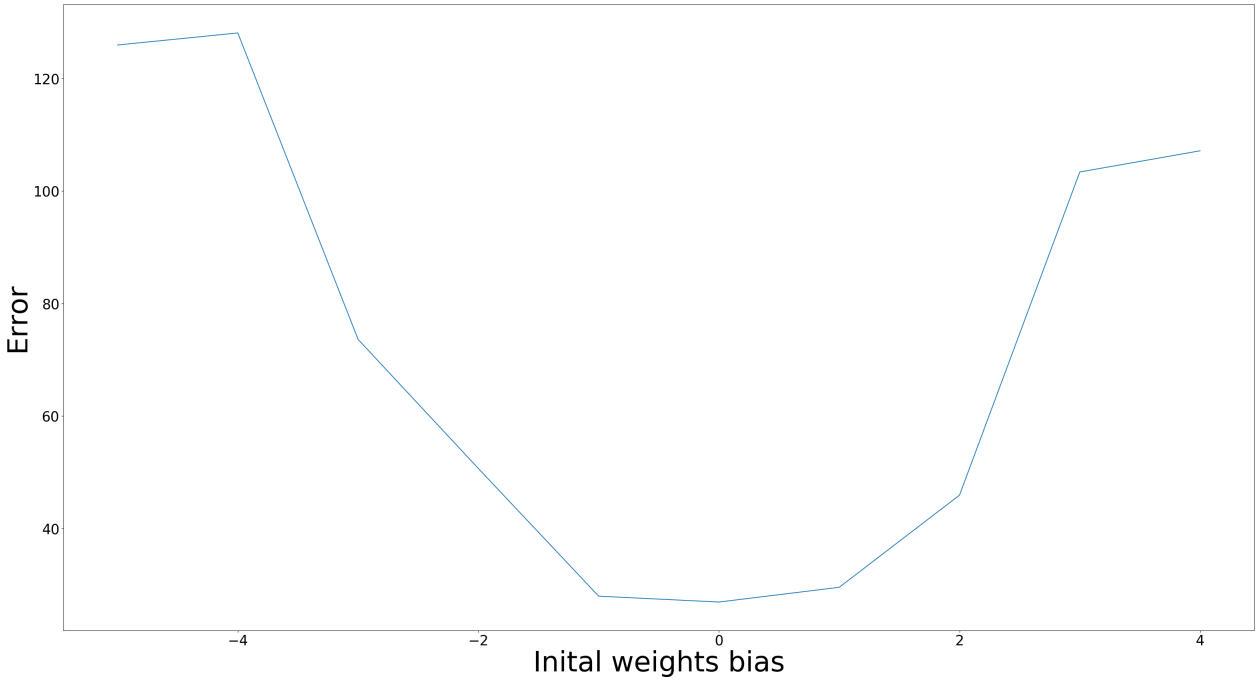


Figure 13: The bias on the random value of the inital weights. A bias of 0 seems to be the best.

To summarize: Only the amount of hidden networks could be a little bit higher in order to get a better convergence, but the values for w and η given in the assignment seemed to be quite close to the optimal, as we can conclude from our tests.

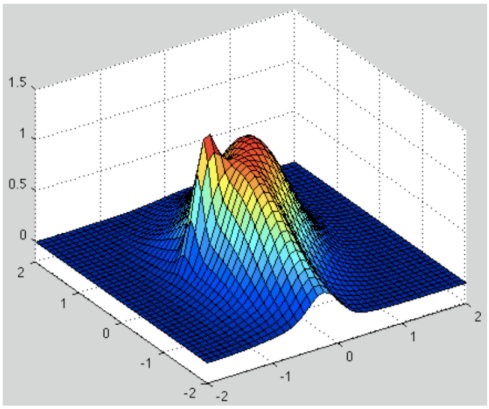


Figure 14: Multi-modal probability density

After these preliminaries we are now going to train the network on the real data set.
Load the data using

```
data = load('a017_NNpdfGaussMix.txt', '-ASCII');
X = data(:,1:2); Y = data(:,3);
```

2.5

Create a $2D$-plot of the target probability density function. Notice that the data is in the correct sequence to use in `surf()`.

Answer:

This has been done with the following code:

```
Data = np.genfromtxt('data/a017_NNpdfGaussMix.txt')

X = Data[:,0:2]
Y = Data[:,2]
Plot(X,Y,"2.5")
```

Figure 15 shows the plot:

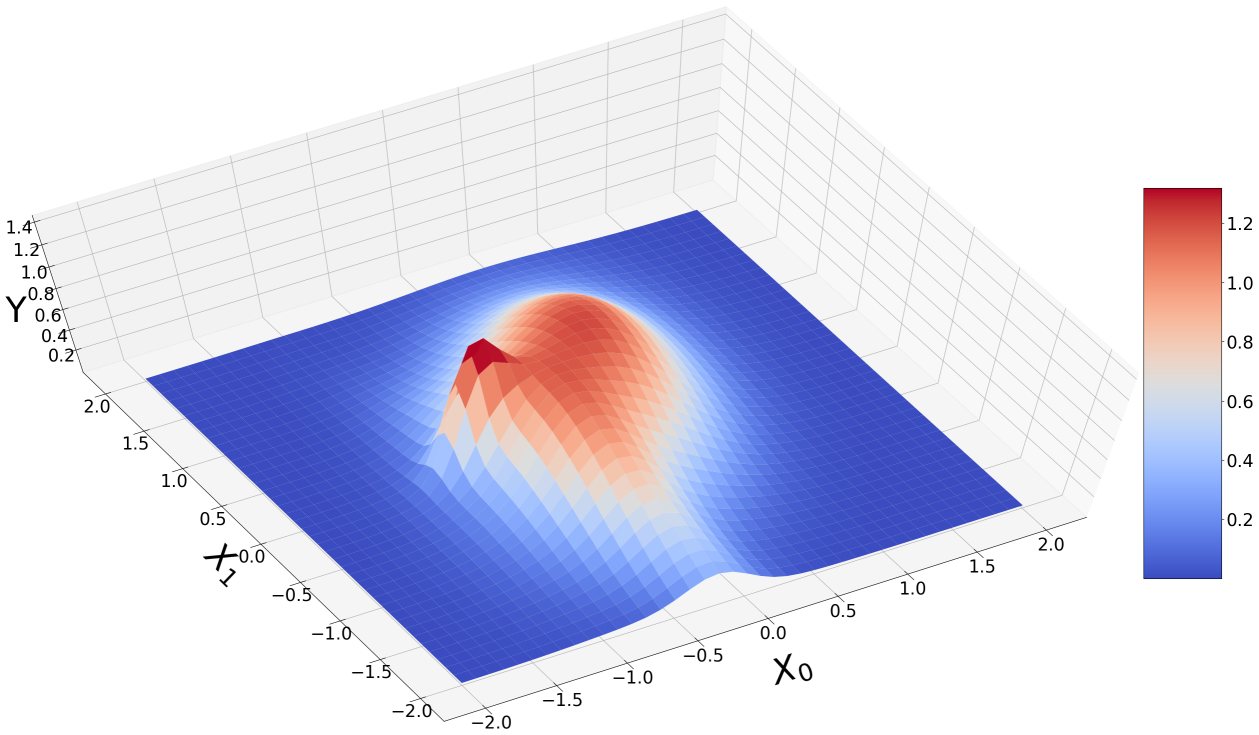


Figure 15: Plot of the data.

2.6

Train the network on this data set. Use at least 40 hidden nodes and a learning rate parameter no higher than $\eta = 0.01$. Make sure the input data is properly randomized. Run the training phase for at least 2000 complete cycles and follow the progress by plotting the updated network output after every 20 full cycles. How does the final output of the network compare to the target distribution in the data? Explain. How could you improve the neural network in terms of speed of convergence and/or quality of the approximation?

Exercise 3 - Gaussian processes (weight 5)

Part 1 - Sampling from Gaussian stochastic processes

One widely used kernel function for Gaussian process regression is given by the exponential of quadratic form, with the addition of constant and linear terms (eq. 6.63 Bishop):

$$k(\mathbf{x}, \mathbf{x}') = \theta_0 \exp\left(-\frac{\theta_1}{2}||\mathbf{x} - \mathbf{x}'||^2\right) + \theta_2 + \theta_3 \mathbf{x}^T \mathbf{x}' \tag{6}$$

We denote by $\boldsymbol{\theta} = (\theta_0, \theta_1, \theta_2, \theta_3)$ the hyperparameter vector governing the kernel function k.

3.1.1

Implement the kernel given by Equation (6) in Matlab as a function of $\mathbf{x}, \mathbf{x}'$ and $\boldsymbol{\theta}$. Note that $\mathbf{x}$ can have any dimension.

Answer:

¡¡¡¡¡¡¡ HEAD

```
def k(x,x2,T):
    return T[0]*np.exp(-(T[1]/2)*np.power(np.abs(x-x2),2))+T[2]+T[3]*x*x2
```

======= ¿¿¿¿¿¿¿ 271b8d91ec376e1687eef9c0bc3919adb2e6d095

3.1.2

We first consider the univariate case. For the parameter values $\boldsymbol{\theta} = (1, 1, 1, 1)$ and $N = 101$ equally spaced points $\mathbf{X}$ in the interval $[-1, 1]$, compute the Gram matrix $\mathbf{K(X,X)}$ (eq. 6.54 Bishop). What is the dimension of $\mathbf{K}$? How can we show that $\mathbf{K}$ is positive semidefinite?

Note: Even when $\mathbf{K}$ is positive definitive, some of its eigenvalues may be too small to accurately compute (same for the determinant). This may pose a problem when generating a multivariate Gaussian distribution using $\mathbf{K}$ as its covariance matrix. You can alleviate this issue by adding a small diagonal term to $\mathbf{K}$.

Answer:

A Gram matrix is a NxN symmetric matrix (Bischop page 293, just before (6.6)). So in this case, the matrix is $101x101$.
A Gram matrix can be computed usinng the following code:

```
def GramMatrix(X,T):
    N = len(X)
    Gram = np.zeros((N,N))
    for i in range(N):
        for j in range(N):
            Gram[i][j] = k(X[i],X[j],T)
    return Gram
```

It first creates an empty NxN matrix, then fills it using the kernel function as required by the definition of the matrix. To show that K is positive semidefinite we could calculate the eigenvalues of the matrix and see if they are all ≥ 0. This can be done with the following code:

```
def SemiDef(Matrix):
    FixedMatrix = Matrix + 1e-10*np.identity(len(Matrix))
    EigVals = np.linalg.eigvals(FixedMatrix)
    Lowest = np.min(EigVals)
    IsSemi = np.real(Lowest)>=0
    return IsSemi
```

This first adds a small value of 10^-10 to the diagonal, as suggested by the note. Then it calculates it's eigenvalues, takes the minimum and looks if the real part is greater then 0. (by default the eigenvalue is complex since any matrix has eigenvalues as a complex number, since some polynomials of the characteristic polynomial of a matrix (like $x^2 + 1$) cannot be factorized using only real numbers. Often in practise the imaginary part is due to inaccuracies and it needs to be neglected in order to see if the lowest value is greater or equal then 0.

3.1.3

We will now use the previously computed matrix $\mathbf{K(X,X)}$ to produce samples from the Gaussian process prior $\mathbf{y(X)}$ $\mathcal{N}(\mathbf{0, K(X,X)})$, with $\mathbf{X}$ being the previously determined N equally spaced points. Generate five functions $\mathbf{y(X)}$ with Matlab and plot them against the N input values $\mathbf{X}$. Repeat the process (remember to compute a new $\mathbf{K}$ each time) for the hyperparameter configurations from Bishop, Figure 6.5:

$$\boldsymbol{\theta} \in \{(1, 4, 0, 0), (9, 4, 0, 0), (1, 64, 0, 0), (1, 0.25, 0, 0), (1, 4, 10, 0), (1, 4, 0, 5)\}.$$

Describe the differences between the plots. Explain in which way each of the kernel parameters affects the generated samples.

Answer:

We've done this using the following code:

```
Theta = np.array([[1, 4, 0, 0], [9, 4, 0, 0], [1, 64, 0, 0], [1, 0.25, 0, 0],
                  [1, 4, 10, 0], [1, 4, 0, 5]])
for j in range(len(Theta)):
    T = Theta[j]
    K = GramMatrix(X,T)
    for i in range(5):
        TestGaus = np.random.multivariate_normal([0]*101,K)
        Ax = plt.subplot(2, 3, j + 1)
        Ax.set_xticks(np.round(np.linspace(-1, 1, 5), 2))
        for tick in Ax.xaxis.get_major_ticks():
```

```
            tick.label.set_fontsize(30)
        for tick in Ax.yaxis.get_major_ticks():
            tick.label.set_fontsize(30)
        Ax.set_xlim([-1,1])
        Ax.plot(X, TestGaus)
        Ax.set_title(str(T),size=30)
```

First we save the different Theta's. Then we use the multivariate.normal function to randomly take a value from the multivariate Gaussian $p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|0, \mathbf{K})$. We plot these in a subplot and then make sure the plot looks better by adjusting the size of the labels and set the xlimit.
What we got can be seen in figure 16:

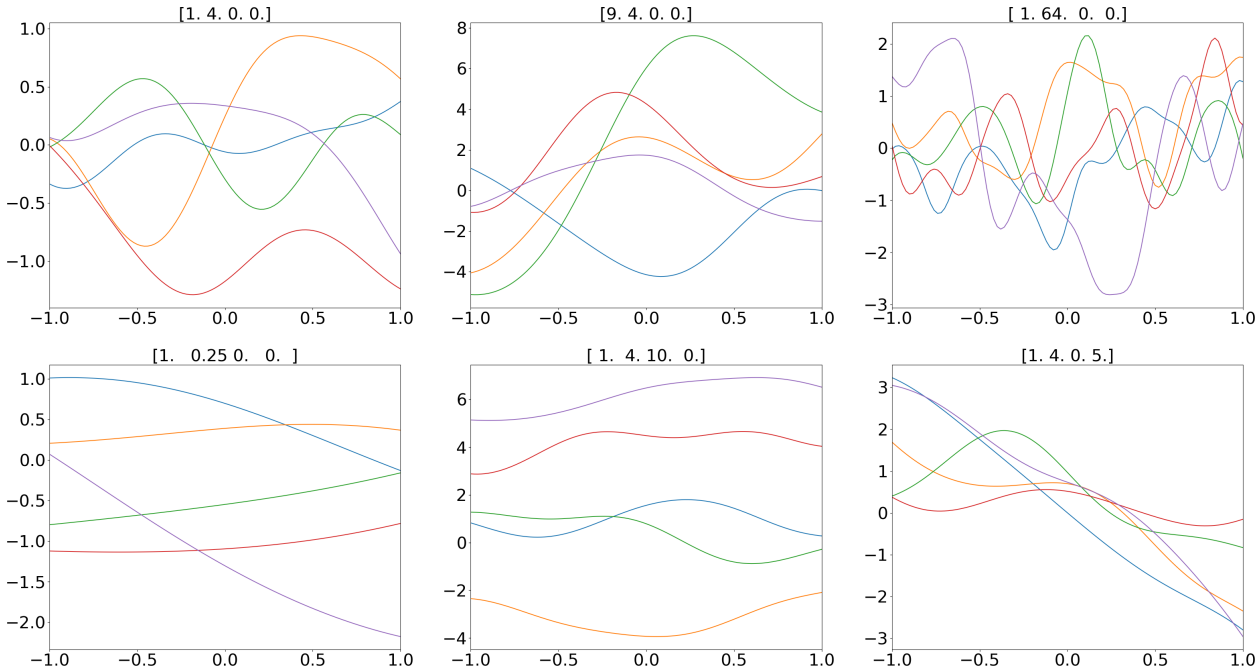


Figure 16: Samples from Gaussian process prior using the Gram matrix with the $\theta's$ given.

3.1.4

We now move to the bivariate case. Instead of an interval, we now consider a 2-D grid of equally spaced points of size $N = 21 \times 21$ in $[-1, 1] \times [-1, 1]$. We collect all these grid points in a data matrix $\mathbf{X}$, where each one of the 441 observations has two dimensions. What is the dimension of $\mathbf{K}$ now? What does this tell you about the scalability of sampling multivariate functions from Gaussian processes in higher dimensions?

Answer:

The Gram matrix K is now a 441 x 441 matrix, with 194,481 entries. This tells that this doesn't scale so well in higher dimensions, for the size of the matrix grows very fast. (The amount of points grows to the power of the amount of dimensions, or: $\#points = n^{2D}$, with n the amount of points in one dimension and D the amount of dimensions. The factor two appears for the Gram Matrix is a square matrix.

3.1.5

Using the same kernel from (6), compute the Gram matrix $\mathbf{K(X,X)}$ on the grid for each hyperparameter configuration $\boldsymbol{\theta} \in \{(1, 1, 1, 1), (1, 10, 1, 1), (1, 1, 1, 10)\}$. For each $\mathbf{K}$, generate and plot four random surfaces from the Gaussian process prior $\mathbf{y(X)}$ $\mathcal{N}\mathbf{(0,\ K(X,X))}$. Compare the observed differences to the univariate case.

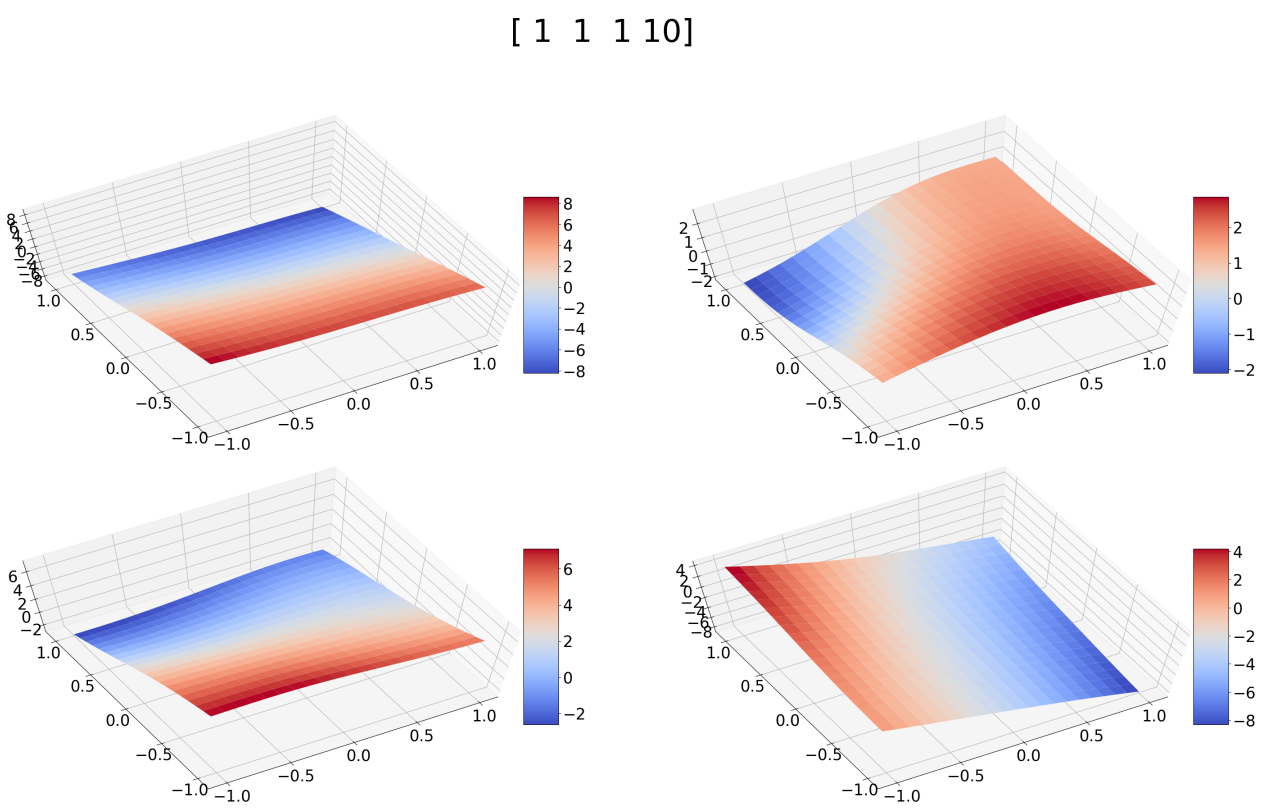
Answer:

We used the following code:

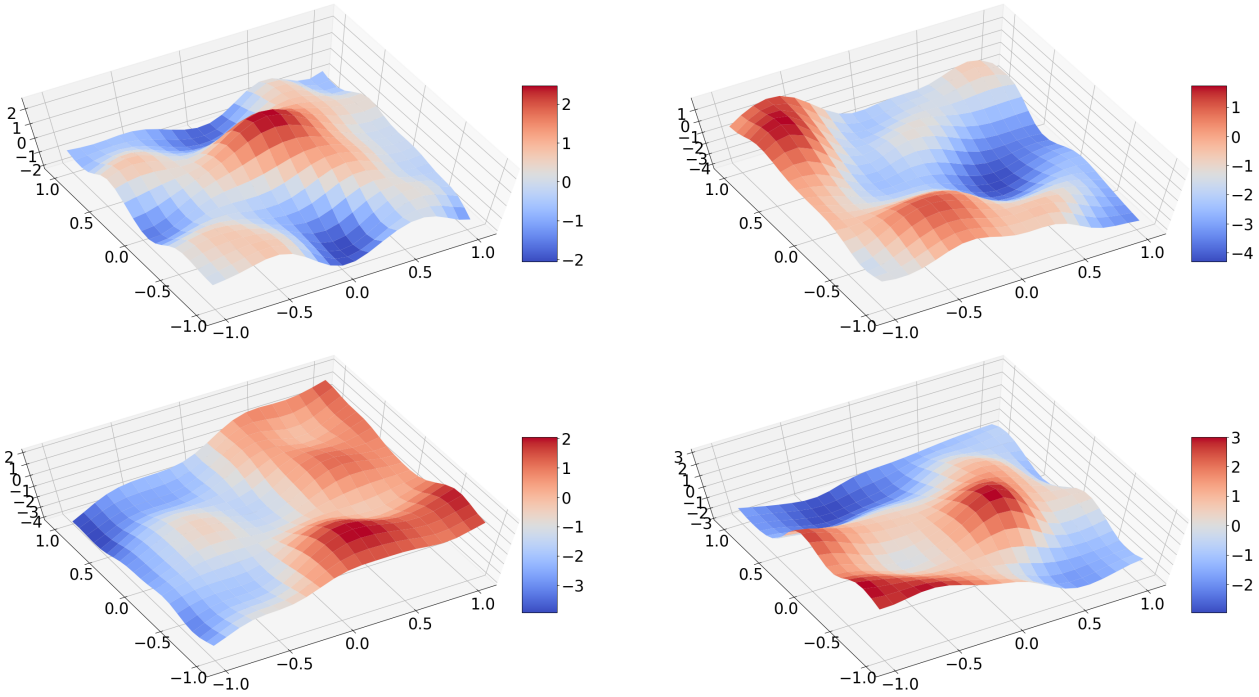
```
Theta = np.array([[1, 1, 1, 1], [1, 10, 1, 1], [1, 1, 1, 10]])
for T in Theta:
    fig = plt.figure(str(T))
```

```
    fig.suptitle(str(T),fontsize=50)
    Gram2 = GramMatrix(X,T)
    for i in range(4):
        Z = np.random.multivariate_normal([0]*441,Gram2)
        Z = np.reshape(Z,(21,21))
        ax = fig.add_subplot(221+i,projection='3d')
        ax.set_xticks(np.round(np.linspace(-1, 1, 5), 2))
        ax.set_yticks(np.round(np.linspace(-1, 1, 5), 2))
        surf = ax.plot_surface(x, y, Z, cmap=cm.coolwarm,linewidth=1, antialiased=True)
        fig.colorbar(surf, shrink=0.5, aspect=5,pad=-0.07)
        ax.view_init(azim=-120,elev = 70)
    pylab.get_current_fig_manager().window.showMaximized()
    plt.subplots_adjust(wspace=0, hspace=0)
```

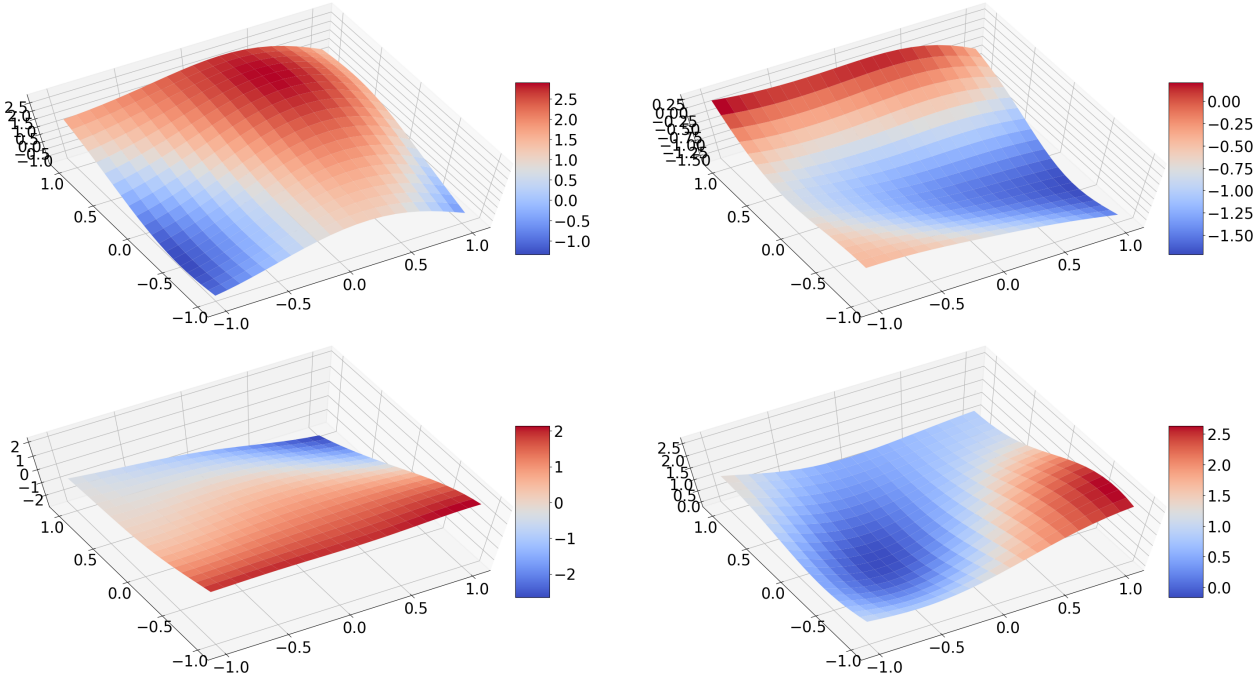
This resulted in the following images:



[1 10 1 1]



[1 1 1 1]



Part 2 - Gaussian processes for regression

We would like to apply Gaussian process models to the problem of regression (Bishop 6.4.2). We consider a noisy model of the form

$$t_n = y_n + \epsilon_n,$$

where $y_n = y(x_n)$ and ϵ_n are i.i.d. samples from a random noise variable on the observed target values. Furthermore, we assume that the noise process has a Gaussian distribution given by:

$$p(t_n|y_n) = \mathcal{N}(t_n|y_n, \beta^{-1}) \tag{7}$$

Going back to a one-dimensional input space, we consider the following training data consisting of four data points

$$\mathcal{D}\{(x_1 = -0.5, t_1 = 0.5), (x_2 = 0.2, t_2 = -1), (x_3 = 0.3, t_3 = 3), (x_4 = -0.1, t_4 = -2.5)\}$$

3.2.1

Just as before, compute the Gram matrix of the training data for $\boldsymbol{\theta} = (1, 1, 1, 1)$. Then, taking $\beta = 1$ in Equation 7, compute the covariance matrix $\mathbf{C}$ corresponding to the marginal distribution of the training target values: $p(\mathbf{t}) = \mathcal{N}(\mathbf{t}|\mathbf{0},\ \mathbf{C})$.

Answer:

Given equation (6.61) and (6.62) from Bishop:

$$p(t) = \int p(\mathbf{t}|\mathbf{y})p(\mathbf{y})d\mathbf{y} = \mathcal{N}(\mathbf{t}|\mathbf{0}, \mathbf{C}) \tag{6.61}$$

$$C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{nm} \tag{6.62}$$

Therefore we simply get:

$$C = K + \beta^{-1}I_N \tag{8}$$

3.2.2

Using the previous results, compute the mean and the covariance of the conditional distribution $p(t|\mathbf{t})$ of a new target value t corresponding to the input $x = 0$. Which equations from Bishop do you need?
Answer:

3.2.3

Does the mean of the conditional distribution $p(t|\mathbf{t})$ go to zero in the limit $x \to \pm\infty$? If so, explain why this happens. If not, how would you set the parameters $\boldsymbol{\theta}$ of the kernel function to make it happen?

Answer:

Exercise 4 - EM and doping (weight 5)

In a certain hypothetical sport, banned substance '**X**' has become popular as a performance enhancing drug, as its presence is hard to establish in blood samples directly. Recently, it has been discovered that users of the drug tend to show a strong positive correlation between concentrations of two other quantities, x_1 and x_2, present in the blood. In contrast, 'clean' athletes tend to fall in one of two or three groups, that either show no or a negative correlation between x_1 and x_2. Unfortunately, as each sample contains only a single, instantaneous, measurement for each variable, it is not possible to establish this correlation from the sample. However, in many cases it is possible to distinguish to which *class* a certain sample belongs by also looking at the values of two other measured variables, x_3 and x_4: certain combinations of measured values are often typical for one class but highly unusual for others.
After a high profile event, a large scale test has resulted in 2000 samples. Rumours suggest the number of positives could be as high as 20%. However, the exact relationship between different classes and typical x values is still not clear. This is where the EM-algorithm comes in...

The blood sample measurements are modelled as a mixture of K Gaussians, one for each class

$$p(x|\mu, \Sigma, \pi) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \sigma_k) \tag{9}$$

where $\mathbf{x} = [x_1, x_2, x_3, x_4]$ represents the values for the measured quantities in the blood sample, $\mu = \{\mu_1, ..., \mu_K\}$ and $\Sigma = \{\Sigma_1, ..., \Sigma_K\}$ are the means and covariance matrices of the Gaussians for each class, and $\pi = \{\pi_1, ..., \pi_K\}$ are the mixing coefficients in the overall data set.

Load the data using

```
data = load('a011_mixdata.txt', '-ASCII');
```

and set N to the number of datapoints and D to the number of variables in the dataset X.

4.1

Try to give an estimate of the number, size and shape of the classes in the data, by plotting the distribution of the variables, e.g. using `hist()`, `scatter()` or `scatter3()`.

Answer:

4.2

Implement an EM-algorithm using the description and formulas given in Bishop, 9.2.2. Use variable K for the number of classes and choose a priori equal mixing coefficients π_k. Initialize the means $\boldsymbol{mu}_k$, to random values around the sample mean of each variable, e.g. set $\mu_{k,1}$ to $\bar{x}_1 + [-1 \leq \epsilon \leq +1]$. Initialize the Σ_k to diagonal matrices with reasonably high variances, e.g. $4 * \texttt{rand()} + 2$, to avoid very small responsibilities in the first step. Make sure the EM-loop runs over at least 100 iterations. Display relevant quantities, at least the log likelihood (9.28), after each step so you can monitor progress and convergence. Write a plot routine that plots the x_1, x_2 coordinates of the data points, and color each data point according to the most probable component according to the mixture model.

Answer:

4.3

Set $K = 2$, initialize your random generator and run the EM-algorithm on the data. Describe what happens. Try different random initializations and compare results.
(Should converge within 50 steps to two clusters, accounting for $\pm 1/3$ *resp.* $2/3$ *of the data).* Plot the x_1, x_2 coordinates colored according to the most probable component. Compute the correlation coefficients

$$p_{12} = \frac{cov[x_1, x_2]}{\sqrt{var[x_1] var[x_2]}} \tag{10}$$

of each of the components (i.e., use their covariance matrices to compute variances and covariances in (9), see also (Bishop, eq. (2.93))). Does either class show the characteristic strong[1] positive correlation for $\{x_1, x_2\}$?

Answer:

4.4

Increase the number of classes to $K = 3$ and rerun your algorithm on the data, again trying different random initializations. Plot the x_1, x_2 coordinates colored according to the most probable component and compute the correlation coefficients of each of the components. Check both your plot and your coefficients if one of

[1] According to Wikipedia, the correlation is none if $|p| < 0.1$, small if $0.1 < |p| < 0.3$, medium if $0.3 < |p| < 0.5$ and strong if $|p| > 0.5$

the clusters now displays the strong positive $\{x_1, x_2\}$ correlation we are looking for.
Increase to $K = 4$, do the same, and see if this improves your result (in terms of detection of the doping-clusters). Based on your findings, is the rumoured 1-in-5 estimate for users of X credible?

Answer:

4.5

Having found the offending cluster in the data using the EM-algorithm, we are now presented with four samples $\{A, B, C, D\}$, with values for $[x_1, x_2, x_3, x_4]$ given as

$$A = [11.85, 2.2, 0.5, 4.0]$$
$$B = [11.95, 3.1, 0.0, 1.0]$$
$$C = [12.00, 2.5, 0.0, 2.0]$$
$$D = [12.00, 3.0, 1.0, 6.3]$$

One of these is from a subject who took drug X, and one is from a subject who tried to tamper with the test by artificially altering one or more of the x_i levels in his/her blood sample.

Identify which sample belongs to the suspected user and which one belongs to the 'fraud'.

Answer: