COMS30007 - Machine Learning

Carl Henrik Ek

November 30, 2019

1 Introduction to Python

Abstract

In this worksheet we will have a look at some operations essential to machine learning and at the same time become comfortable with `Python` as a programming language and how to use the packages `numpy`, `scipy` and `matplotlib`. There is a lot of tutorials on how to use `Python` out there[1] and this is by no means a complete one. We have tried to focus on the elements that you will need to work on the labs in the unit. Importantly, many of you are very likely much better programmers than me, in this case trust your own judgement and don't list too much to what I have to say about programming.

1.1 Environment

The first thing we are going to do is to install an environment and all the necessary packages. To not make a mess of your previous installs we are going to create a new environment. Follow the

1. Download 64-bit Linux Miniconda installation script for Python 3.X from here.
2. From a terminal execute

   Code
   ```
   sh <path-to-folder>/Miniconda3-latest-Linux-x86_64.sh
   ```

3. Scroll through the license (press the space bar to move through quickly), type 'yes' to approve the terms, and then accept all the installation defaults.
4. Close the Terminal.
5. Open the Terminal again and execute the following commands,

   Code
   ```
   conda update conda
   conda create -n myenv python
   conda install -n myenv jupyter scipy numpy matplotlib
   ```

6. Now we have installed the necessary packages and created a miniconda environment called `myenv` which can be started by,

   Code
   ```
   source activate myenv
   ```

[1] an excellent *cheat-sheet* can be found here.

1. Development Now there are many ways that you can write the code and perform the experiments. If you have a sadistic nature you could even run them `Matlab` but that would be wrong on so many levels.

   Option 1 stand-alone python, pick your editor of choice[2] simply execute the code in python. Say that you write some code called `coms30007.py` you simply open a terminal and run,

   Code
   ```
   python coms30007.py
   ```

   Option 2 Using the `iPython` interactive shell. This approach allows you to open a shell where the variables that is generated by your code are resident. This means that you can interactively play around with your workspace. This is my preferred method when I work as the machine learning process is often very interactive. First open the `iPython` shell from a terminal,

   Code
   ```
   ipython
   ```

   You can now run code and interact with the results. Lets say that we have some code called `coms30007.py` that creates a variable `x`.

   Code
   ```
   run coms30007
   print(x)
   ```

   Option 3 Using `Jupyter Notebook`. This is a browser based way to interact with the `iPython` shell. Start the `Jupyter` server by executing the following in a terminal and follow the instructions. Personally I find it dreadful to write code in a browser but for `emacs` users there is obviously a better alternative.

   Code
   ```
   jupyter notebook
   ```

   Do play around with the different methods and find the one that is most suitable for you. The labs are here not for you to simply finish a bit of code and be done but rather to play around with code to get a better understanding of the underlying theory.

1.2 Python

Programming languages comes and goes but currently Python is the king of the hill when it comes to machine learning. The main benefit of Python is that it is very easy to interact with other languages, parsing data is super simple and there now exists good nummerical libraries and tools for plotting. If you are using a interactive Python shell you can write `foo.bar?` to print out the help of `foo.bar`, if you write `foo.bar??` you instead print the actual implementation.

1.2.1 Numpy

The main library that we use to build machine learning models and methods will be `Numpy`. Numpy is very well implemented and contains a lot of useful functions to work on vectors and matrices. The normal convention is to import numpy as `np`. To create vectors and matrices you use `np.array`.

[2] emacs obviously

Code

```
import numpy as np
x = np.array([1, 2, 3]) # create 3 vector
y = np.array([[2], [3], [4]]) # create a 3x1 vector
A = np.array([[1, 2, 4],[2, 6, 8],[3, 3, 3]]) # create a 3x3 matrix

# print dimensionality
print('x:', x.shape, 'y:', y.shape, 'A:', A.shape)
```

There are several prebuilt constructions that returns commonly used matrices,

Code

```
np.zeros((2,2))       # create a 2x2 zero matrix
np.ones((2,3))        # create a 2x3 matrix of ones
np.eye(5)             # create a 5x5 identity matrix
np.empty((3,4))       # create a 3x4 placeholder matrix
np.arange(1,9,2)      # create a vector with values from 1 to 9 with increment of 2
np.linspace(0,1,100) # create a vector of 100 linearly spaced values between 0 and 1
```

Often we want to inspect a matrix/vector and get its dimensions/shape this can be done as follows,

Code

```
x = np.eye(5)
x.shape
```

1. Arithmetic The normal operations for addition and subtraction have been overloaded to `numpy` objects, so we can simply use `+,-,*,/` as we would normally do. Importantly, the operations are element-wise so the do not implement matrix arithmetics. Lets first generate some data,

   Code

   ```
   import numpy as np

   x = np.arange(3).reshape(3,1) # 3x1 vector
   y = np.arange(3).reshape(1,3) # 3x1 vector
   A = np.arange(9).reshape(3,3)  # 3x3 matrix
   ```

   As long as we are working with objects with the same dimensionality everything makes but you can also do a few more involved operations where the results are a bit more surprising. Try the following operations out and make sure you understand what they do.

   Code

   ```
   print(x-y)
   print(y-x)
   print(A-x)
   print(A-y)
   print(x*y)
   print(A*x)
   print(x*A)
   print(A*y)
   ```

To work with matrix operations we use the following notation,

Code

```
A.dot(x)        # scalar product
A.dot(y.T)      # scalar product with transpose of y
np.outer(x,y.T) # outer product
```

Now its time to introduce one of the annoying things with `numpy` its very relaxed Broadcasting rules. Test the following,

Code

```
A = np.arange(9).reshape(3,3)
x = np.arange(3)
y = np.arange(3).reshape(3,1)
A.dot(x)
A.dot(x.T)
A.dot(y)
A.dot(y.T)
```

The last dot-product above should fail, which makes sense as you are trying take the scalar product between a `3x3` matrix and a `1x3` vector. However both the first and the second dot-product computes, isn't this weird? Yes it is, but what is happening is that in numpy you can create arrays which have 1 dimension. If you check `x.shape` you will se that it is `(3,)`. This is very annoying as effectively it picks the shape of the vector to be whatever is needed to get the operation to work. One way to avoid this is to always reshape vectors so that they have a two dimensional shape. You can easily do this by,

Code

```
x = x.reshape(-1,1)
```

where I have used the `-1` notation which means if you take the index of a vector which is `-1` you get the last index. So for the vector above replacing `-1` with `3` would have the same effect. However many functions that you will call in numpy assumes that a vector have the shape `(dim,)` so when calling them you have to revert back to the one-dimensional numpy array, see below for an example that you will probably use a lot,

Code

```
mu = np.ones(2).reshape(-1,1)
x = np.random.multivariate_normal(mu.flatten(), np.eye(2))
```

Here is another strange thing,

Code

```
x = np.ones((32,1))
y = np.ones(32)
z = x+y
print(z.sum())
```

at least I was a bit surprised by the result of this computation. So my advice, make sure that each vector is of shape `(dim, 1)` and use `.flatten()` when required.

To keep things consistent I try to always follow this convention for the shape of my data, vectors are column vectors and when I concatenate `N` `d` dimensional vectors into a matrix I do it so that each row of the matrix corresponds to a data-point thereby generating an `N` by `d` matrix.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{Nd} \end{bmatrix} \tag{1}$$

This notation might feel rather unatural, and in many ways it really is but its a legacy from computer science. In statistics one would concatenate $\mathbf{X}$ such that each column is a vector as this makes more mathematical sense. As an example, say that I have a linear mapping from a d_2 dimensional to a d_1 dimensional space parametrised as a matrix $\mathbf{A}$. Now I can evaluate the mapping of a point $\mathbf{x}_i$ as,

$$\underbrace{\mathbf{y}_i}_{d_1 \times 1} = \underbrace{\mathbf{A}}_{d_1 \times d_2} \cdot \underbrace{\mathbf{x}_i}_{d_2 \times 1},$$

just as we are used to. The difference comes if we want to evaluate `N` data-points at the same time. By concatenating the data as a statistician its follows naturally,

$$\underbrace{\mathbf{Y}}_{d_1 \times N} = \underbrace{\mathbf{A}}_{d_1 \times d2} \cdot \underbrace{\mathbf{X}}_{d_2 \times N}$$

while the concatenation used in machine learning will be the same in the individual point case it will be rather different in the matrix case as,

$$\underbrace{\mathbf{Y}}_{N \times d_1} = \underbrace{\left(\underbrace{\underbrace{\mathbf{A}}_{d_1 \times d_2} \cdot \underbrace{\mathbf{X}^{\mathrm{T}}}_{d_2 \times N}}_{d_1 \times N} \right)^{\mathrm{T}}}_{N \times d_1} = \left(\mathbf{X}^{\mathrm{T}}\right)^{\mathrm{T}} \mathbf{A}^{\mathrm{T}} = \mathbf{X}\mathbf{A}^{\mathrm{T}}.$$

So in the statistics case everything works out naturally and the order of the operation stays the same while in the computer science notation it doesn, why would anyone want to use the latter. Personally I believe this is a legacy from thinking of implementation first. If I have a matrix in memory I most often want to access each dimension in turn, this means that if I have a pointer at the start of the matrix I can do this by incrementing the pointer one step at the time. In the statistics notation I will have to increment the pointer with the number of columns in the matrix. I don't know if this is true but its the best explanation I can come up with. Sadly this notation have gotten stuck in machine learning so in the literature you will see the row-vector concatenation, to that end we will use this notation throughout the unit.

2. Linear Algebra To compute the inverse of a matrix we want to solve the following system of linear equations for an unknown $\mathbf{A}$ and a known $\mathbf{M}$,

$$\mathbf{A}\mathbf{M} = \mathbf{I}.$$

The most straightforward way to do this is to call the matrix inverse function in `np.linalg`,

Code

```
# generate a random matrix
M = np.random.randn(100).reshape(10,10)

# compute the inverse of M
A = np.linalg.inv(M)

# print the diagonal elements
print(np.diag(A.dot(M)))

# pring the sum of the absolute values of the off-diagonal elements
print(np.abs(A.dot(M)).sum() - np.abs(A.dot(M))[np.diag_indices(M.shape[0])].sum())
```

In the code above we print first the values of the diagonal elements using the function `np.diag()` which extracts them from the matrix. Then we print the sum of the off-diagonal elements using another useful function `np.diag_indices()` this function returns the indices of the diagonal elements. As you can see the diagonal elements are all `1.0` while the off-diagonal elements are all very close to `0`.

(a) **EXTRA** Cholesky Decomposition Now computing the inverse of a matrix is something that we will do a lot and its a very expensive operation with cubic complexity, not something that you want to have in your inner-loop. Furthermore `np.inv` is a completely general function and often we have matrices with specific structures or types. This is something that we should take into account in our computation[3]. The Cholesky decomposition is valid for any positive definite matrix as,

$$\mathbf{M} = \mathbf{L}\mathbf{L}^{\mathrm{T}},$$

where $\mathbf{L}$ is an lower-triangular matrix. As the decomposition is unique to the matrix it is first much cheaper to store as it has fewer values. So lets that we want to compute the inverse in order to solve the following system of equations,

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

where $\mathbf{A}$ and $\mathbf{x}$ is known and $\mathbf{b}$ is unknown. Furthermore we know that the matrix $\mathbf{A}$ is positive-definite[4]. We know that we can compute the inverse of $\mathbf{A}$ and get the solution by,

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b},$$

but this is not exploiting the fact that we know that $\mathbf{A}$ is positive definite. In this case we can exploit the Cholesky decomposition of a matrix to good effect making the computation both faster and more stable.

[3] Good blog post on this
[4] You will see this a lot later in the unit

Code

```
# generate a "hopefully" PSD matrix
A = np.random.randn(100).reshape(10,10)
A = A.dot(A.T)+np.eye(A.shape[0])

# Ax=b
b = np.ones(A.shape[0]).reshape(-1,1)

# compute cholesky factor
L = scipy.linalg.cholesky(A, lower=True)
# solve system of equations using cholesky
x = scipy.linalg.cho_solve((L, True), b)

# print the "difference" between the two
print(np.abs(b-A.dot(x)).sum())
```

Most of the problems that we do in this unit you will involve small and reasonably well behaved matrices so directly computing the inverse will be fine but when you get to large problems understanding what is stable and efficient is key to writing well behaved and efficient code.

(b) Eigenvalues We often want to do eigen-decompositions of matrices as this representation is much more interpretable. The eigen-decomposition takes a square matrix `M` and decomposes it into,

$$\mathbf{M} = \mathbf{V}\mathbf{U}\mathbf{V}^{\mathrm{T}},$$

where $\mathbf{V}$ are the eigenvectors which forms an orthonormal basis and the diagonal matrix $\mathbf{U}$ which have the eigenvalues on the diagonal. This can easily be done using,

Code

```
# create a hopefully PSD matrix
M = np.random.randn(100).reshape(10,10)
M = M.dot(M.T)+np.eye(M.shape[0])

# compute eigen decomposition
U, V = np.linalg.eig(M)

# print element-wise error
print(np.abs(V.dot(np.diag(U).dot(V.T))-M).sum())
```

as you can see above when calling `np.diag` with a vector as an argument it creates a matrix with the vector on the diagonal.

1.2.2 Indexing

One of the real benefits of `numpy` and `Python` in general is how easy it is to *slice'n'dice* objects and apply operations only on a subset of the matrix. A matrix is indexed as `[row, col]` with the first index being `0`. There are several different indexing modes. For `numpy` arrays its important to know that there is no copying of data involved, you are simply providing a different view of an array by indexing. A good guide to the indexing modes in `numpy` is available here

1. Direct Matrices in `Python` are indexed by `[row, col]`, in order to access all elements you can use a `:` as in the examples below,

Code

```
# create a 10x10 matrix
A = np.arange(100).reshape(10,10)

# differnet direct index modes
print(A[0,0]) # first row and column
print(A[:,0]) # all rows of first column
print(A[4,:]) # all columns of fourth row
print(A[:])   # all elements
```

You can also use the `:` notation to access parts of the matrix. The notation `[i:j,3]` implies column `3` and row `i` up till, but not including `j`. If you do not explicitly write identify the `start` it is assumed to be `0`

Code

```
print(A[1:3, 2])  # access elements at row 1,2 and column 2
print(A[:3, 2])   # access element at row 0,1,2 and column 2
print(A[:2, 4:6]) # access element at row 0,1 and column 4,5
```

As we have seen before index `-1` implies the last index,

Code

```
print(A[-1,-1])  # access the element at the last row and column
print(A[:-1,0])  # access all the rows except for the last of the first column
print(A[0:-1,0]) # same as the above
print(A[:10, 0]) # access all the row of the first column
```

2. Operations along axes Many operations have an `axis` argument, what this argument indicates is the `axis` of the matrix where the operation is applied to. So as an example

Code

```
print(A.mean())        # compute the mean value of the matrix
print(A.mean(axis=0))  # compute the mean value of each column of the matrix
print(A.mean(axis=1))  # compute the mean value of each row of the matrix
```

Many commands take `axis` arguments so its worth looking at the documentation by `?` as it can often lead to cleaner code. A very useful example is `np.sort` where you can decided to sort column or row-wise.

3. Boolean Many operations that are boolean in `Python` are vectorised. For example, if you write,

Code

```
A = np.random.randn(10,10)
B = A>0
```

`B` will be a boolean matrix with `True` at locations where the element is larger than 0 and `False` elsewhere. The nice thing about this is that we can use the matrix `B` as a means of indexing a matrix. So say that we would want to get all the values of the matrix `A` which are positive we could simply do,

Code

```
A = np.random.randn(10,10)
A[A>0]
```

This can be used to write very compact and interpretable code and I find it especially useful for drawing conditional plots when I am just interested in plotting points that fore fill a certain criteria.

1.2.3 Matplotlib

A friend of mine explained machine learning not as programming but as debugging[5]. His analogy of the machine learning task is that you walk down the street and you find a hard-drive lying on the street, you know that there is something important on this but that's all you know. You have no idea what file-system it is, what the important information is etc. So what do you do, you start poking it with all the tools that you have to figure out what the important information is. A very important tool for this process is to visualise your data, plot different things to give you a visual understanding of the data. However horrible `Python` is as a language the biggest benefit it has is the amazing visualisation packages that exists. We will use `matplotlib` a lot but there are other more specific packages for plotting things like distributions[6]. We will follow the convention of importing `matplotlib.pyplot` as `plt`.

Disclaimer

In the following examples there will be a set of commands towards the end of the code that saves the image and a return statement, please ignore these its just so that the image gets saved and included in the LaTeX document. You can replace this by `plt.show()` as this will display the figure you have generated.

There are mainly two objects that you need to know about, `figures` and `axes`. The former corresponds to a "physical" figure while the latter corresponds to sections of the latter. You first create a `figure` and then you add `axes` to this figure. In the example below we create a single `figure` called `fig` and then we add `2*3=6` subplots to this figure and call the call the first one of these `ax`.

Code

```
import matplotlib.pyplot as plt #import packages

fig = plt.figure(figsize=(10,5)) #create figure handle
ax = fig.add_subplot(231)        #create axis handle
```

Once we have create a figure we can do some simple plotting, in the example below we create a figure with one row and two columns and create a simple line plot and a scatter plot. These are really

[5] `http://inverseprobability.com/2017/03/14/data-science-as-debugging`
[6] A very useful package you might want to look at is seaborn

Code

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

# create data
x = np.linspace(-np.pi, np.pi, 100)
y = np.sin(x)

# line plot
ax1.plot(x, y, linewidth=2.0, color='b')

# add noise
y += 0.3*np.random.randn(y.shape[0])

# scatter plot
ax2.scatter(x, y, s=20, color='r')

# IGNORE THESE
plt.tight_layout()
plt.savefig(path, transparent=True)
return path
```

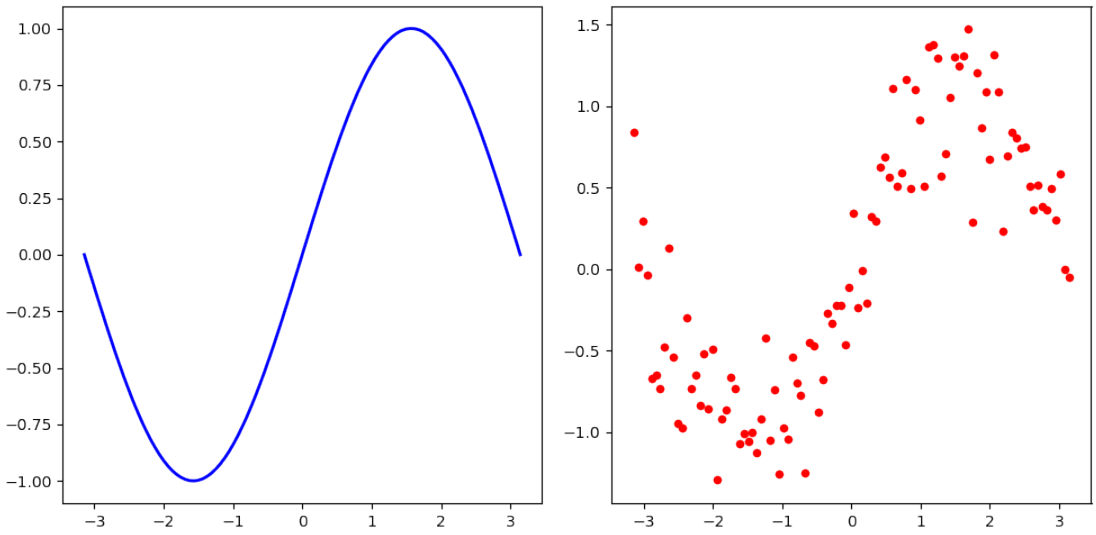

Figure 1: The left most pane is associated with `ax1` and the rightmost with `ax2`

Code

```
ax.set_xlim()
ax.set_xticks()
```

We will often work with rather large matrices in machine learning and getting an idea of how they are structured is often very telling. If the matrix consists of real values we can simply consider it to be an image and visualise it in this way.

Code

```
import numpy as np
import matplotlib.pyplot as plt

M = np.sort(np.random.randn(256,256))
fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
im1 = ax1.imshow(M)
ax2 = fig.add_subplot(122)
im2 = ax2.imshow(20*M)
ax2.set_yticks([])
plt.colorbar(im1, ax=(ax1))
plt.colorbar(im2, ax=(ax2))

plt.savefig(path, transparent=True)
return path
```

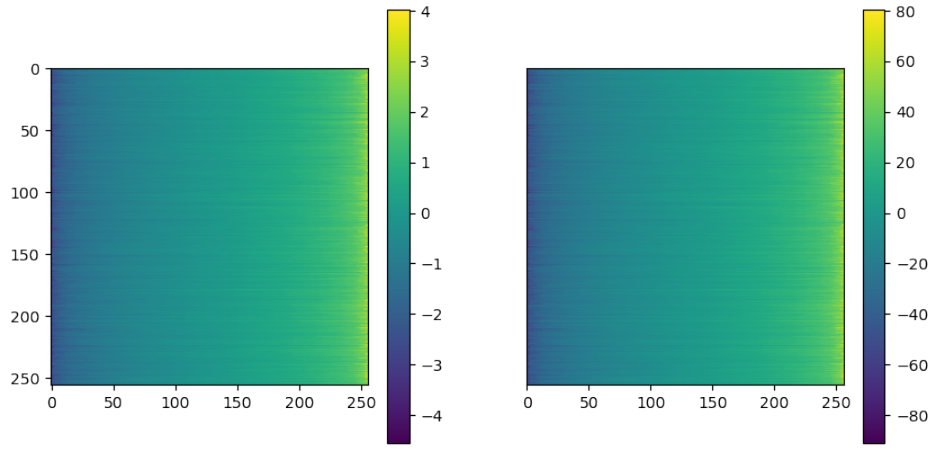

Figure 2: Figure showing how we can visualise matrices as images. Note that `imshow()` rescales the colours so to the range of the values in the matrix. The colourbars to the left shows the different maps used.

There is lots more to `matplotlib` than this and its really an excellent package, if you are interested in learning more look at the documentation here.

1.3 Distributions

`Numpy` has a class called `random` which contains a lot of different distributions. The distributions in `np.random` are used for generating samples from distributions. To see all the functions/members of the object `object` you can execute.

Code

```
{ k:v for k,v in vars(object).items() if not k.startswith('_') }
```

Hopefully you will recognise quite a few of the names of distributions under here. Most of the distributions have the same structure. The distribution that we will use the most in the unit is the `Gaussian` or `Normal` distribution so let us have a

Code

```
loc = np.array([0,0])
scale = np.array([[1, 0],[0, 1]])
f = np.random.normal(loc, scale)
```

In many cases we are instead interested in the actual probability functions rather than samples from it. We can access them through the package `scipy.stats` and directly create random variables. In next weeks lab we will use the `beta` distribution so lets start with this. We first import the distribution from `scipy.stats` and then we create an instance of this which we now will refer to as a *random variable*. This is a concept that will be central to anything that we do in machine learning. In most cases we are used to *deterministic variables* these are variables that takes a specific value, for example in C syntax `int a = 3` is a variable `a` that takes the value `3`. In contrast a random variable is a variable that takes values following a specific distribution as an example,

$$\epsilon \sim \mathcal{N}(0, 1),$$

is a variable ϵ that takes values that follows a normal distribution with mean `0` and variance `1`. Don't worry if this is not clear right now, you will see a lot of this over the next few weeks. When we have created the random variable `rv` in the code below we have access to lots of different functions that we can interact with. Below we plot the *density* the *cummulative density* and also sample from the variable.

Code

```
from scipy.stats import beta
import numpy as np
import matplotlib.pyplot as plt

# create random variable
rv = beta(10.0, 10.0)

# create an index set for the distribution
index = np.linspace(0+1.0e-8, 1, 100)

# create plot
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)

# plot density function
ax.plot(index, rv.pdf(index), color='g')
ax.fill_between(index, rv.pdf(index), color='g', alpha=0.3)

# plot cumulative density function
ax.plot(index, rv.cdf(index), color='r')
ax.fill_between(index, rv.cdf(index), color='r', alpha=0.3)

# sample from random variable
Y = rv.rvs(1000)

# plot histogram
hist, bins = np.histogram(Y)
ax.bar(bins[:-1], hist/hist.sum(), alpha=0.4, color='b',width=0.05)

plt.tight_layout()
plt.savefig(path, transparent=True)
return path
```

When you have gotten this code to work now re-write this to work with the normal distribution instead. First try to repeat the same thing in one dimension and then change to do the same with a two dimensional normal distribution instead. The two distributions that you want to use are,

Code

```
from scipy.stats import norm
from scipy.stats import multivariate_normal
```

make sure that you check the documentation by writing `multivariate_normal?` to see how it needs to be initialised. For a two dimensional distribution the index-set will be different a simple way to create a two dimensional grid is this,

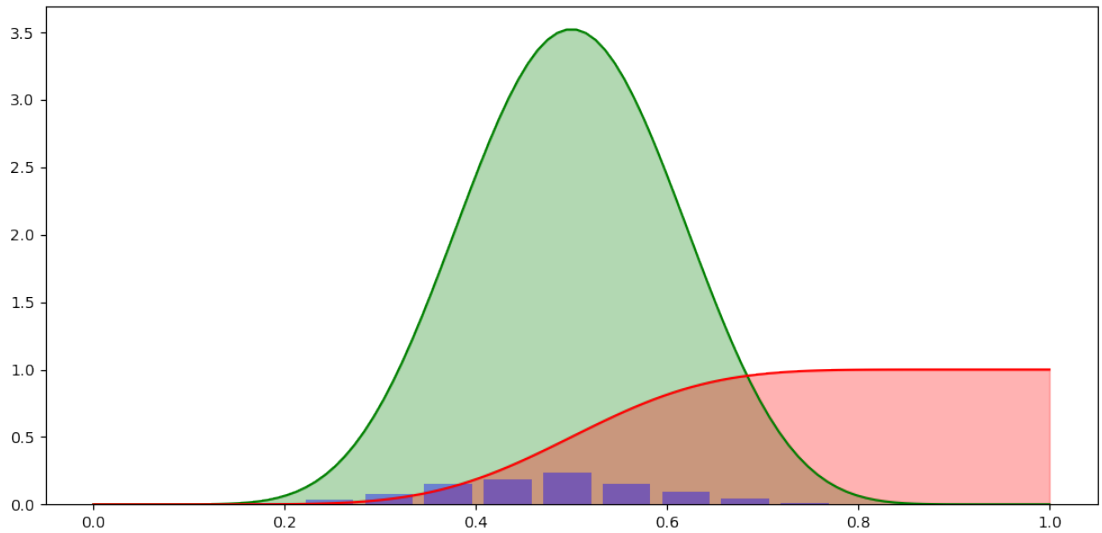

Figure 3: The green curve shows the density function, the red the cummulative density function and the blue a histogram of samples from the random variable.

Code

```
# create a 1d index
x = np.linspace(-1.5,1.5,100)
# create all combinations of locations from the 1d index
x1p, x2p = np.meshgrid(x,x)
# stack these positions into a single vector
pos = np.vstack((x1p.flatten(), x2p.flatten())).T
```

1.4 Summary

Hope that you managed to get through this worksheet and found a sensible way to working with Python. As said in the beginning you are most likely much better programmers than me and I'm sure I've said more than one wrong thing in this document. The important thing is that this unit is **not** a programming unit, we will only write code in oder to experiment to give ourselves an intuition of the underlying math. Most exercises will not contain much more code than what we did today. Next week will look at distributions and do a first easy example of how we can learn from data. Hope that you are looking forward to this.

2 Distributions

Abstract

In this part we will look at how we can learn from data. The aim with this lab is to understand the concepts of *conjugacy* and the interplay between *likelihood*, *prior* and *posterior*. In this task the data is binary and next week we will work on continuous data. Importantly the ideas are the same, the computations are the same, your challenge is therefore to abstract this document beyond the details and understand the underlying concepts.

We have been given the task to observe a system which has a binary outcome, you can think of the example of modelling a coin toss. We will parametrise the system using a single parameter that describes how often we get each outcome, with the coin analogy how biased the coin is. We will refer to a single outcome of the system as x and if we run the system for N iterations we will refer to all the data $\mathbf{x}$. We will first create a function to generate the data by sampling from a distribution with known parameters, the machine learning task is then to *recover* the parameters of the distribution from only the data-points.

In order to phrase this as a learning problem we need to formulate the probabilistic objects that constitutes the model of how the data have been generated. We will formulate the *likelihood* function that will express our belief in specific types of data if we know the parametrisation of the system. Say that if I think that I have an unbiased coin then seeing 100 heads in a row would be a very unlikely outcome while seeing 50 heads and 50 tails would be highly likely. We will then parametrise the *prior* which will encode our belief in the parameter value of the system, i.e. how likely do we believe the coin to be biased before we see data. Having these two objects we have completely specified our model and can generate data. Now the *inference* procedure starts where we try to reach the *posterior* distribution. The posterior distribution is the distribution that encapsulate both our prior belief about the system with the evidence in the data.

2.1 Likelihood

The first thing we need to think of is what is the likelihood function. For a binary system it makes sense to use a `Bernoulli Distribution` as likelihood function,

$$p(x|\mu)\text{Bern}(x|\mu) = \mu^x(1-\mu)^{1-x}.$$

This distribution takes a single parameter μ which completely parametrises our likelihood. This means, we have a conditional distribution and the system is completely specified by μ. If we know μ we can generate output that is similar to what the actual system does, this means we can predict how the system behaves. Now we want to use examples of the systems predictions `training data` find μ. So far we have only set the likelihood for one data-point, what about when we see lots of them? Now we will make our first assumption, we will assume that each output of the system is independent. This means that we can factorise the distribution over several trials in a simple manner,

$$p(\mathbf{x}|\mu) = \prod_{i=1}^{N} \text{Bern}(x|\mu) = \prod_{i=1}^{N} \mu^x(1-\mu)^{1-x}.$$

Now we have the likelihood for the whole data-set $\mathbf{x}$. Think a bit about what this assumption implies. It means that,

1. each data-point is independent as our likelihood function is invariant to any permutation of the data.
2. it assumes that each of the data-points are generated by the same distribution. For the coin example this means that by tossing the coin you do not change the actual coin by the act of tossing it[7].

[7] this is probably an OK assumption for metal coins while for chocolate coins this assumption is too simplistic

2.2 Prior

In order to say something about the system we need to have a prior belief about what we think the parameter μ is. Now our prior knowledge comes into play, what do I know about the system? If our system is the outcome of a coin toss then we have a lot of prior knowledge, most coins that I toss are unbiased so I have quite a good idea of what I think it should be. Another way of seeing this is that I would need to see a lot of coin tosses saying something else for me to believe that a coin is not biased. If it is not a coin toss but something that I have no experience in my prior might be different. Once we have specified the prior we can just do Baye's rule and get the posterior,

$$p(\mu|\mathbf{x}) = \frac{p(\mathbf{x}|\mu)p(\mu)}{p(\mathbf{x})}.$$

Now comes the first tricky part, what should the distribution be for the prior? If you choose your prior or likelihood wrong[8] then this computation might not even be analytically possible to perform. So this is when we will use *conjugacy*. First lets note that the posterior distribution is proportional to the likelihood times the prior (or the joint distribution),

$$p(\mu|\mathbf{x}) \propto p(\mathbf{x}|\mu)p(\mu).$$

The second thing we will think of is that it does make sense that if I have a prior of a specific form (say a Gaussian) then I would like the posterior to be of the same form (i.e. Gaussian). So this means that we should *choose* a prior such that when multiplied with the likelihood it leads to a distribution that is of the same form of the prior. So why is this useful? This is very useful because it means we do **not** have to compute the denominator in Baye's rule, the only thing we need to do is to multiply the prior with the likelihood and then **identify** the parameters that of the distribution, we can do this as we know its form.

So how do we know which distributions are conjugate to what? Well this is something that we leave to the mathematicians most of the time, we simply exploit their results. There is a list on Wikipedia of conjugate priors here URL. Its like a match making list for distributions. For the Bernoulli distribution the conjugate prior to its only parameter μ is the `Beta` distribution,

$$\text{Beta}(\mu|a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1},$$

where $\Gamma(\cdot)$ is the gamma function. The role of the Gamma function is to normalise this to make sure it becomes a distribution not just any function. Now we have choosen our prior we are ready to derive the posterior.

[8] its a bit more complicated than this but you have to believe me on this right now but we will see more of this later in the unit

2.3 Posterior

To get to the posterior we are going to multiply the likelihood and the prior together,

$$
\begin{aligned}
p(\mu|\mathbf{x}) &\propto p(\mathbf{x}|\mu)p(\mu) && (2)\\
&= \prod_{i=1}^{N} \text{Bern}(x|\mu)\text{Beta}(\mu|a,b) && (3)\\
&= \prod_{i=1}^{N} \mu_i^x(1-\mu)^{1-x_i}\frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1} && (4)\\
&= \mu^{\sum_i x_i}(1-\mu)^{\sum_i(1-x_i)}\frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1} && (5)\\
&= \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{\sum_i x_i}(1-\mu)^{\sum_i(1-x_i)}\mu^{a-1}(1-\mu)^{b-1} && (6)\\
&= \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{\sum_i x_i+a-1}(1-\mu)^{\sum_i(1-x_i)+b-1}. && (7)
\end{aligned}
$$

Now comes the trick with conjugacy, *we know the form of the posterior*. This means we can just identify the parameters of the posterior and in this case it is trivial,

$$
p(\mu|\mathbf{x}) \propto \mu^{\underbrace{\sum_i x_i + a - 1}_{a_n - 1}}(1-\mu)^{\underbrace{\sum_i (1-x_i) + b - 1}_{b_n - 1}}.
$$

Now what is left is to make sure that the expression is actually a probability distribution such that it integrates to one. This means that we need to solve the following,

$$
1 = Z\int p(\mu|\mathbf{x})\mathrm{d}\mu = Z\int \mu^{a_n-1}(1-\mu)^{b_n-1}.
$$

In this case this is trivial as we know the normaliser of the beta distribution which means that,

$$
Z = \frac{\Gamma(a_n+b_n)}{\Gamma(a_n)\Gamma(b_n)}
$$

This mean that my posterior is,

$$
p(\mu|\mathbf{x}) = \text{Beta}(\mu|a_n,b_n) = \frac{\Gamma\left(\sum_i x_i + a + \sum_i(1-x_i) + b\right)}{\Gamma\left(\sum_i x_i + a\right)\Gamma\left(\sum_i(1-x_i)+b\right)}\mu^{\sum_i x_i + a - 1}(1-\mu)^{\sum_i(1-x_i)+b-1}
$$

So thats it, we have the posterior and now we can fix our parameters for the prior a and b and then compute the posterior and get a_n and b_n after seeing n data-points. So lets write code that simulates one of these experiments.

2.4 Implementation

Common practice if you want to test something is to generate data from your model with known parameters, throw away the parameters and then see if you can recover the parameter. What we first then want to do is to sample a large set of binary outcomes. We can do this by using the `Binomial`[9] in `numpy`. So we start of with setting μ to 0.2 and then generate 200 values from this distribution, i.e. running the system 200 iterations or tossing a coin 200 times. Then we define our prior by setting the parameters a and b. Now we can compute our posterior, we know its form, we both derived it above, but in most cases you just write it

[9] `https://en.wikipedia.org/wiki/Binomial_distribution`

down, that is what you will do for linear regression. Now we can plot the posterior when we see more and more examples and see what will happen. Below is the image that it generated for me,

Code

```
import numpy as np
from scipy.stats import beta
import matplotlib.pyplot as plt

def posterior(a,b,X):
    a_n = a + X.sum()
    b_n = b + (X.shape[0]-X.sum())

    return beta.pdf(mu_test,a_n,b_n)

# parameters to generate data
mu = 0.2
N = 100

# generate some data
X = np.random.binomial(1,mu,N)
mu_test = np.linspace(0,1,100)

# now lets define our prior
a = 10
b = 10

# p(mu) = Beta(alpha,beta)
prior_mu = beta.pdf(mu_test,a,b)

# create figure
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)

# plot prior
ax.plot(mu_test,prior_mu,'g')
ax.fill_between(mu_test,prior_mu,color='green',alpha=0.3)

ax.set_xlabel('$\mu$')
ax.set_ylabel('$p(\mu|\mathbf{x})$')

# lets pick a random (uniform) point from the data
# and update our assumption with this
index = np.random.permutation(X.shape[0])
for i in range(0,X.shape[0]):
    y = posterior(a,b,X[:index[i]])
    plt.plot(mu_test,y,'r',alpha=0.3)

y = posterior(a,b,X)
plt.plot(mu_test,y,'b',linewidth=4.0)

# ignore this
plt.tight_layout()
plt.savefig(path, transparent=True)
return path
```

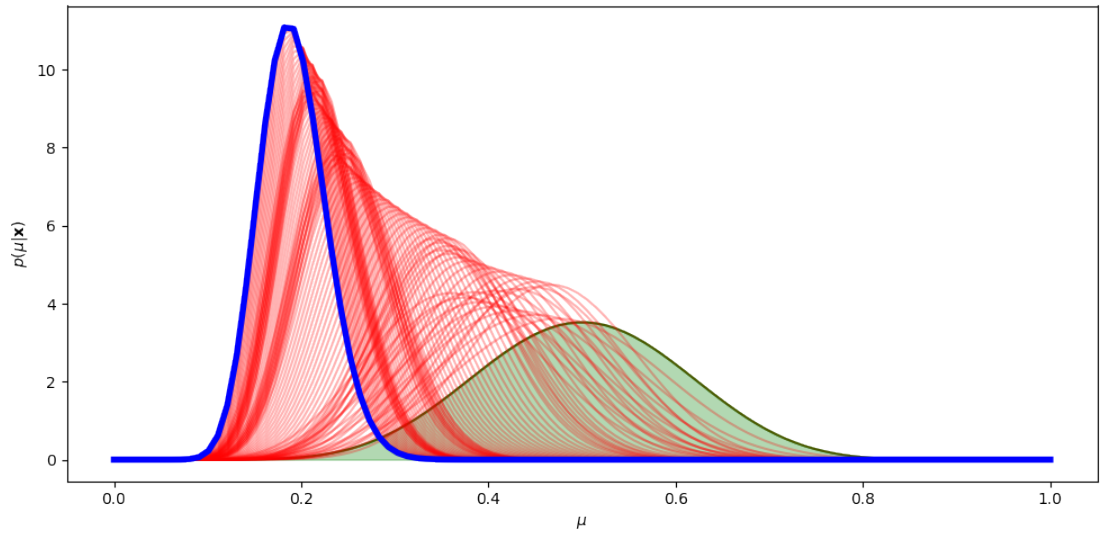

Figure 4: *The green distribution is the prior distribution over μ and the red distributions are the updated belief, our posterior when I see more and more data-points and the blue is the final posterior when we have seen all the points.*

As you can see from the plot above the posterior distribution places significant mass over the value of μ that was used to generate the data. Furthermore, when there is very little evidence i.e. we have seen very little data, the posterior is very close to the prior.

Now when we have built our model we can try out lots of different things, implement the tasks below and evaluate what they mean

1. what happens if you choose a prior that is very confident at a place far from the true value?
2. what happens if you choose a prior that is very confident at the true value?
3. in the plot above we cannot see the order of the red lines, create a plot where the `x-axis` is the number of data-points that you have used to compute the posterior and the `y-axis` is the distance of the posterior mean to the prior mean.
4. How much does the order of the data-points matter? Redo the plot above but with many different random permutations of the data. Now plot the `mean` and the `variance` of each iteration. What can you see?
5. Even disregarding computational benefits why does it make sense too choose the conjugate prior?

2.5 Summary

The intention of this lab was to show the mechanism that allows us to learn from data, how we can take our prior beliefs and update them by learning from data. Even though a coin-toss might be a silly example[10] it importantly contains all the elements of what we will do over the next few weeks. So your task is to really try and abstract this simple example to what it really "means" and then you will find the more mathematically complicated tasks later a lot simpler.

[10] not if you are playing cricket where if you win the toss you **always** choose to bat

3 Linear Regression

Abstract

In this lab we will extend the example from last week to more complicated data. In this case we will look at a model that relates two continuous variables to each other also known as a *regression task*. To simplify things we will limit the hypothesis space and only look at relationships that are linear. Importantly the concepts of the lab is *exactly* the same as in the binary case and its important that you realise this. However differently from last weeks lab I will be less verbose when deriving the model and inference scheme so some parts are left for you to fill in.

We are interested in the task of observing a data-set $\mathcal{D} = \{\tilde{x}_i, y_i\}_{i=1}^N$ where we assume the following relationship between the variates,

$$y_i = f(\tilde{x}_i). \tag{8}$$

Our task is to infer the function $f(\cdot)$ from $\mathcal{D}$. To simplify things we are going to limit the hypothesis space to be only of linear functions. This means that we can write Eq 30 as,

$$y_i = w_1\tilde{x}_i + w_0 = \mathbf{w}^{\mathrm{T}}\mathbf{x}_i = \begin{bmatrix} w_1 \\ w_0 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \tilde{x}_i \\ 1 \end{bmatrix} \tag{9}$$

where we have rewritten the input variate by appending a one so that we can write everything on matrix form. The task that we will perform in this lab is to infer the function parametrised by $\mathbf{w}$ by observing $\mathcal{D}$.

3.1 Model

Now we will make our first assumption, we will assume that the data we observed is not instantiations of the "true" underlying function but rather have been corrupted by *addative* noise. This means that we get the following model,

$$y_i = \mathbf{w}^{\mathrm{T}}\mathbf{x}_i + \epsilon. \tag{10}$$

Now can we make an argument what form this noise will take? One assumption would be to say that the noise is independent of where in the input space we evaluate the function, this is called *homogenous* noise. Furthermore we could argue that we know the form of the noise, one idea would be to say that the noise is Gaussian,

$$\epsilon \sim \mathcal{N}(0, \beta^{-1}),$$

with precision β. Now what this would mean is that if we could directly observe the noise we could formulate a likelihood as,

$$p(\epsilon) = \mathcal{N}(\epsilon|0, \beta^{-1}). \tag{11}$$

Now we can use Eq.10 and rewrite,

$$y_i = \mathbf{w}^{\mathrm{T}}x_i + \epsilon \tag{12}$$

$$y_i - \mathbf{w}^{\mathrm{T}}x_i = \epsilon \tag{13}$$

If we now combine this new expression of the noise with the assumption of the stochastic form in Eq.11 we get,

$$p(\epsilon) = \mathcal{N}(\epsilon|0, \beta^{-1}) \tag{14}$$

$$= \mathcal{N}(y_i - \mathbf{w}^{\mathrm{T}}x_i|0, \beta^{-1}) \tag{15}$$

$$= \mathcal{N}(y_i|\mathbf{w}^{\mathrm{T}}x_i, \beta^{-1}). \tag{16}$$

The last step can be done because we can "translate" a Gaussian distribution.

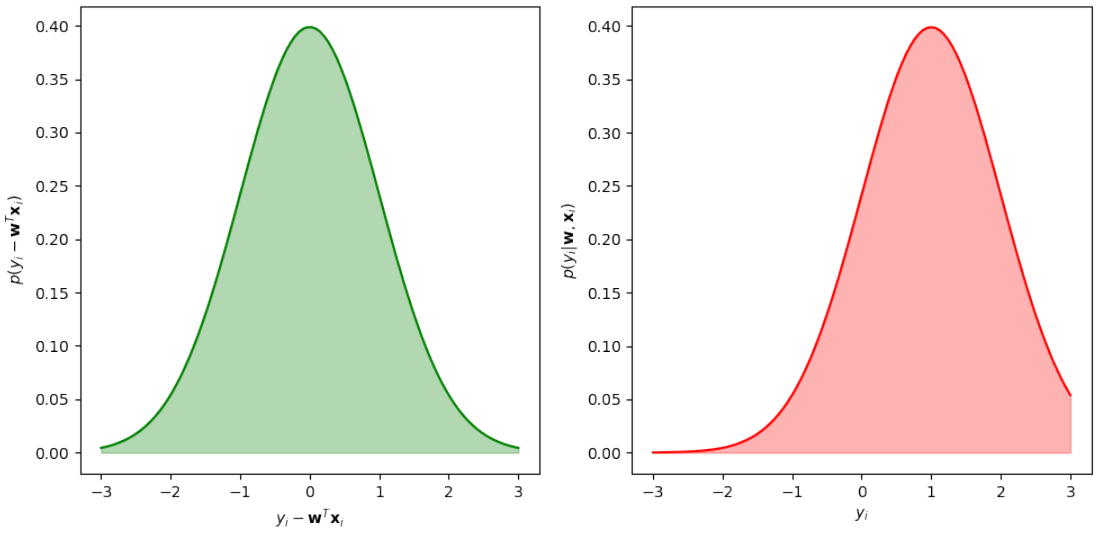

Figure 5: Figure showing how we can re-parametrise a Gaussian by translation. In the first case we have a Gaussian with mean zero over the difference between the observed output y_i and the output of the function $\mathbf{w}^{\mathrm{T}}\mathbf{x}_i$ which is the same as a Gaussian over the output y_i with mean $\mathbf{w}^{\mathrm{T}}\mathbf{x}_i$ where in this case the latter equates to 1.

What we have just done is to formulate a likelihood function, how likely is an observed output location given that we know the parametrisation of the function. Now the formulation is for a single data-point but also assuming that the noise is independent we can easily formulate the likelihood for a sequence of data,

$$p(\mathbf{y}|\mathbf{w}, \mathbf{X}) = \prod_{i=1}^{N} p(y_i|\mathbf{w}, \mathbf{x}_i), \tag{17}$$

where $\mathbf{X} = [x_1, \ldots, x_N]^{\mathrm{T}}$ and $\mathbf{y} = [y_1, \ldots, y_N]^{\mathrm{T}}$. Think back to the Bernoulli trial in last weeks lab, its exactly the same thing. One important difference is how we motivated the construction of the likelihood, it came through making an assumption of the structure of the noise. Concretely we made three assumptions,

1. the noise is addative
2. the noise is Gaussian
3. the noise is independent of the input location.

These assumptions needs to be justified on a problem to problem basis. Say for example that we are measuring the vibrations in a car as a function of speed. It might be the case that the noise in the sensor measuring the vibrations also depends on the speed of the car then we cannot use a homoscedastic noise model. In this case we use these assumptions because we want to show how to build models rather than looking at specific data.

So now we have our likelihood function and if we knew the parameters of the function $\mathbf{w}$ we would be able to generate data. Now our task is to recover the function by observing data. In order to do so we first need to specify our prior belief in the parameters.

3.2 Prior

In order to specify our prior distribution we will again use the concept of conjugacy. If we look at the likelihood in Eq.17 it is itself a Gaussian distribution[11]. We will also assume that the parameters of the

[11] check this for yourself by writing up the product of the individual terms

noise β is known. Now we can again look-up what the conjugate prior is for a Gaussian with known variance[12]] as it turns out this in itself is another Gaussian distribution. So in order to exploit conjugacy we will use a Gaussian prior for the parameters of the function such that,

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}_0, \mathbf{S}_0). \tag{18}$$

The structure of the prior covariance tells us that we assume that the two parameters w_1 and w_0 are independent with equal variance. Think about this assumption, does this make sense for a line? Well the grate thing about distributions is that you can sample from it and generate the results.

Code

```
import numpy as np
import matplotlib.pyplot as plt

def plot_line(ax, w):
    # input data
    X = np.zeros((2,2))
    X[0,0] = -5.0
    X[1,0] = 5.0
    X[:,1] = 1.0

    # because of the concatenation we have to flip the transpose
    y = w.dot(X.T)
    ax.plot(X[:,0], y)

# create prior distribution
tau = 1.0*np.eye(2)
w_0 = np.zeros((2,1))

# sample from prior
n_samples = 100

w_samp = np.random.multivariate_normal(w_0.flatten(), tau, size=n_samples)

# create plot
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)

for i in range(0, w_samp.shape[0]):
    plot_line(ax, w_samp[i,:])

# save fig
plt.tight_layout()
plt.savefig(path, transparent=True)
return path
```

1. what is the most likely line according to your prior belief?
2. what is the least likely line according to your prior belief?
3. is there any lines that have zero probability in this belief?

[12] https://en.wikipedia.org/wiki/Conjugate_prior

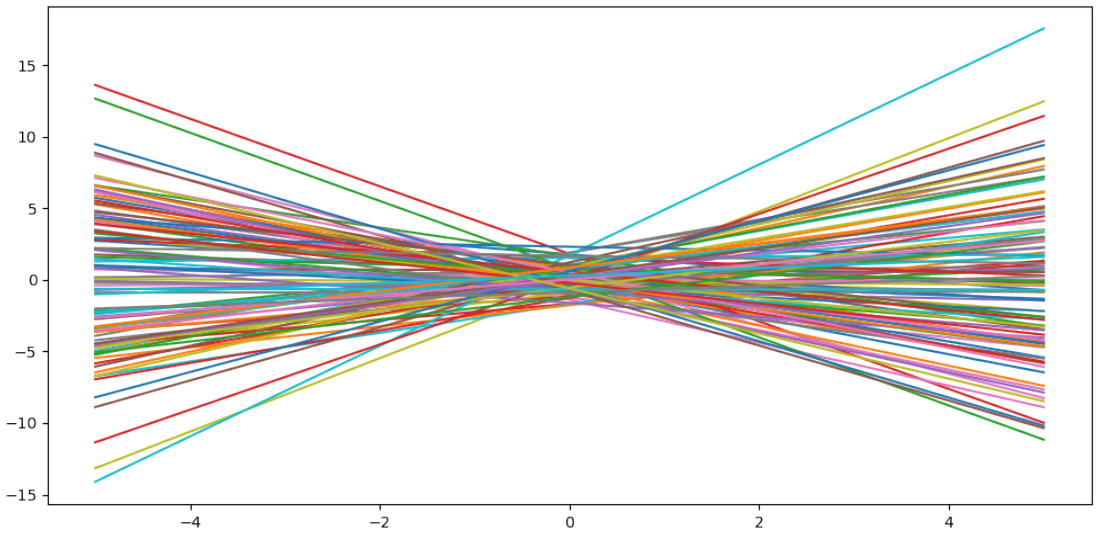

Figure 6: Samples from the prior with the prior covariance set to identitiy and prior mean of 0.

3.3 Posterior

Now we have encoded our prior belief and we have formulated our likelihood function and its time to formulate the posterior distribution. We will do exactly the same thing as we did in the Bernoulli trial but the math will be a bit more complicated. Go back and look at lecture 4 and make sure that you understand the derivation that we did as we will now use these results.

To derive the posterior distribution we will use two concepts,

1. that the posterior is proportional to the likelihood times the prior
2. due to conjugacy we know that the posterior is a Gaussian

Proportionality means that,

$$p(\mathbf{w}|\mathbf{y},\mathbf{X}) \propto p(\mathbf{y}|\mathbf{X},\mathbf{w})p(\mathbf{w}). \tag{19}$$

Conjugacy means that we know the form of the right-hand side,

$$p(\mathbf{w}|\mathbf{y},\mathbf{X}) = \mathcal{N}(\mathbf{w}|\mu(\mathbf{y},\mathbf{X}),\sigma(\mathbf{y},\mathbf{X})). \tag{20}$$

Now what remains to be done is to first multiply the left-hand side of Eq.19 and identify the unknown terms $\mu(\cdot)$ and $\sigma(\cdot)$ in Eq.20. [13]. Doing so will lead to the posterior distrubution,

$$p(\mathbf{w}|\mathbf{y},\mathbf{X}) = \mathcal{N}\left(\mathbf{w}|\left(\mathbf{S}_0^{-1}+\beta\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\left(\mathbf{S}_0^{-1}\mathbf{w}_0+\beta\mathbf{X}^{\mathrm{T}}\mathbf{y}\right),\left(\mathbf{S}_0^{-1}+\beta\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\right). \tag{21}$$

The expression above looks rather daunting at first but actually does make a lot of sense when you start looking at it. One way of making sense of the posterior is to look at some extreme scenarios, think about the following

1. what would happen if you assume a noise-free situation i.e. $\beta \to \infty$
2. what would happen if we assume a zero mean prior?
3. what happens if we do not observe any data?

[13] look through the lecture hand-out for the full derivation

4. when you observe more and more data which terms are going to dominate posterior?

Make sure that the expression makes sense and that you build an intuition. We will now proceed to experiment with the expression

3.4 Implementation

Once you have done the mathematical interogation of the posterior it is time to evaluate the model by generating some data and then aim to recover the parameters that generated this specific data. Again, just the same proceedure as we did in the Bernoulli trial. The first thing we need to do is to decide on some paramters, let us assume that the data have been generated as,

$$y_i = \mathbf{w}^{\mathrm{T}}\mathbf{x}_i + \epsilon \tag{22}$$

$$\epsilon \sim \mathcal{N}(0, 0.3) \tag{23}$$

$$\mathbf{X} = \begin{bmatrix} -1 & 1 \\ -0.99 & 1 \\ \vdots & \vdots \\ 1 & 1 \end{bmatrix} \tag{24}$$

$$\mathbf{w} = \begin{bmatrix} -1.3 \\ 0.5 \end{bmatrix} \tag{25}$$

First visualise the prior distribution over $\mathbf{w}$. As this is a two dimensional distribution we have to show this as an image. One simple way is to create a function that samples evaluates the distribution on a grid and then creates a contour plot,

Code

```
"""
Create a contour plot of a two-dimensional normal distribution

Parameters
----------
ax : axis handle to plot
mu : mean vector 2x1
Sigma : covariance matrix 2x2

"""

def plotdistribution(ax,mu,Sigma):
    x = np.linspace(-1.5,1.5,100)
    x1p, x2p = np.meshgrid(x,x)
    pos = np.vstack((x1p.flatten(), x2p.flatten())).T

    pdf = multivariate_normal(mu.flatten(), Sigma)
    Z = pdf.pdf(pos)
    Z = Z.reshape(100,100)

    ax.contour(x1p,x2p,Z, 5, colors='r', lw=5, alpha=0.7)
    ax.set_xlabel('w_0')
    ax.set_ylabel('w_1')

    return
```

Now we will do an iterative proceedure where we pick a random point from the data-set, compute and visualise the posterior and visualise the sample functions from the same distribution. So first combine the code to plot sample with the visualisation of the distribution and then generate a loop similar to this,

Code

```
index = np.random.permutation(X.shape[0])
for i in range(0, index.shape[0]):
    X_i = X[index,:]
    y_i = y[index]

    # compute posterior
    # visualise posterior
    # visualise samples from posterior with the data
    # print out the mean of the posterior
```

You can iterate through this and add a pause statement in your loop or you can skip the loop completely and just run the code above by setting `i` as a variable and testing for interesting values. If this works as it should you should be able to regenerate the plots from the lecture. Observe what the mean of the posterior is, in an ideal setting we should eventually recover the parameters that generated the data. Play with the parameters of the model, the noise variance, the prior and get an intuitive feeling for how everything fits together.

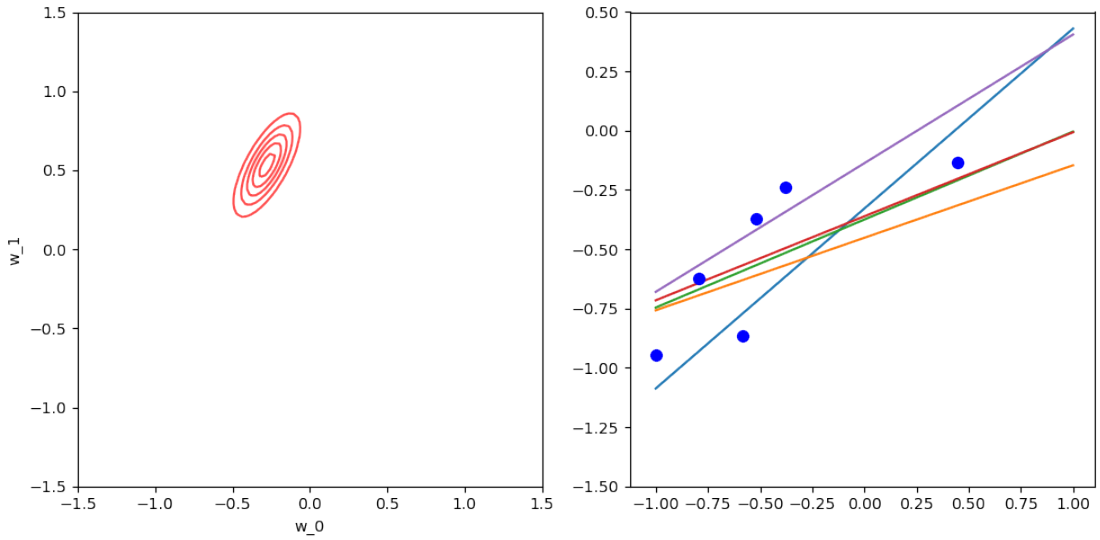

Figure 7: The right pane shows the posterior distribution after seeing the 6 data-points on the left. The left pane shows the samples from this posterior.

Now think about the following questions,

1. our prior is spherical, what assumption does this encode? Does this make sense for a line?
2. with a few data-points the posterior starts quickly to look non-spherical, what does this mean? Does this make sense?
3. with many data-points the posterior becomes spherical again? Why is this? Look at Eq.21 can you see why this is the case for this data?

3.5 Predictive Posterior

So far we have a way of learning the parameters of the function, but the parameters is just a means to an end, what we really want is to perform predictions. This means that given a new input location $\mathbf{x}_*$ we want to have a distribution over what we believe the output location to be. Of course this distribution should take into account the training data we have used to learn the weights of the function. The way to get to this point is to *marginalise out* the parameters of the function $\mathbf{w}$. In other words generate all possible lines and weigh them with how much we believe in each of them based on what we have learned. This can be done as follows,

$$p(y_*|\mathbf{x}_*, \mathbf{X}, \mathbf{y}) = \int p(y_*|\mathbf{x}_*, \mathbf{w}) p(\mathbf{w}|\mathbf{X}, \mathbf{y}) \mathrm{d}\mathbf{w}, \tag{26}$$

where we integrate the likelihood for a new point with the posterior distribution over the parameters. Being that both these are Gaussian we can compute this integral again in closed form which leads to the following distribution,

$$p(y_*|\mathbf{x}_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}\left(\mathbf{y}|\mathbf{m}_N^{\mathrm{T}}\mathbf{X}, \beta^{-1} + \mathbf{X}^{\mathrm{T}}\mathbf{S}_N\mathbf{X}\right), \tag{27}$$

where $\mathbf{m}_N$ and $\mathbf{S}_N$ is the posterior mean and variance for $\mathbf{w}$ having seen N points from the training data. As they have this nice dependency just write a new function that wraps the posterior over the weights something like,

Code

```
def predictiveposterior(m0, S0, beta, x_star, X, y):
    mN, SN = posterior(m0, S0, beta, X, y)

    m_star = mN.T.dot(x_star)
    S_star = 1.0/beta + x_star.T.dot(SN).dot(x_star)

    return m_star, S_star
```

3.6 EXTRA Non-linear basis functions

If you are interested you can extend the linear case and work with a non-linear basis function instead. The code should be exactly the same you just first need to map the input data to a different space Z and then perform the linear regression there,

$$\Phi : \tilde{\mathcal{X}} \to \mathcal{Z} \tag{28}$$

$$y_i = \mathbf{w}^{\mathrm{T}}\Phi(\mathbf{x}_i) + \epsilon \tag{29}$$

A simple example of a mapping is to use an exponential function as,

$$\phi_d(\tilde{x}_i) = e^{-(\tilde{x}_i - b_d)^{\mathrm{T}}(\tilde{x}_i - b_d)},$$

where b_d is the "center" of the basis function. You can distribute these linearly along the input space, say that if you pick 10 basis functions you will do linear regression in a 10 dimensional space but map the result back to the original problem space where the 10 dimensional line will look like a non-linear function. It is a bit mind boggling to get your head around this one so experiment till you understand the concept. We will look at this more in future labs and already next lab we will take this to the extreme and let the number of basis-functions go towards infinity and then some really cool stuff starts to happen. But don't worry to much about the interpretation for now just justify for yourself that you do get non-linear functions when doing this approach.

3.7 Summary

Now you have reached the end of this lab and hopefully you manged to generate the plots and have understood the connection between the assumptions, the mathematical inference proceedure and the results. Even though it might feel like a very simple example it really is not, you have taken prior knowledge, combined this with data and generated new knowledge. Now we will move onto more complicated models but it is important that you see that it is all the same proceedure.

4 Gaussian Processes

Abstract

In this lab we will look at Gaussian processes or GPs. These are non-parametric models that allows us to reason not about parameters of the function that we want to infer but in terms of more interpretable components as the characteristics of the function. GPs are used exentsively for many different applications, for reinforcement learning [1] for modelling complex dynamical systems [2] and for black-box optimisation [3]. As an example of the latter Gaussian processes played a central role in how AlphaGo managed to beat the world GO champion [4]. So these are really cool and useful things but as always we will try to keep the application as simple as possible so that we actually understand what goes on under the hood of these models.

Last week we looked at regression where we aimed to fit a function to a set of data. However, our hypothesis space was rather limited and we could only work with linear functions. This week we will look at the same model again, but now use a much richer set of hypothesis, Gaussian processes. The most natural way to think of Gaussian processes are as distributions over the space of functions. Importantly they have this wonderful characteristic that they put non-zero probability over every continous function. This seems like a strange concept at first as we naturally then think about *parametrisations* of functions. What would be a sufficiently rich parametrisation that includes all functions? This is where the concept of non-parametrics comes in, we will not think about functions as parametric objects but as non-parametric objects where we specifically describe the characteristic of the *output* of the function. This is the weird and beautiful world of non-parametrics and getting some clarity about this is the aim of this lab.

4.1 Model

We are interested in the task of observing a data-set $\mathcal{D} = \{x_i, y_i\}_{i=1}^N$ where we assume the following relationship between the variates,

$$y_i = f(x_i). \tag{30}$$

Our task is to infer the function $f(\cdot)$ from $\mathcal{D}$. We will make the same assumptions as in the linear regression case which leads us to the following likelihood function,

$$p(\mathbf{y}|f(\mathbf{x})) = \prod_{i=1}^{N} p(y_i|f(x_i)), \tag{31}$$

where $f(x_i)$ is the output of the function at location x_i. As you can see the generative model here is slightly different as we have not made any assumptions on the structure of the function. But lets for a minute assume that we had access to a distribution that encoded our belief in a general function $p(f)$ we could then formulate the joint distribution such as,

$$p(\mathbf{y}, f|\mathbf{x}) = p(\mathbf{y}|f, \mathbf{x})p(f). \tag{32}$$

We will now proceed to define $p(f)$.

4.2 Prior

Our task here is to define a prior distribution over the space of functions, this means a distribution $p(f)$ such that you can evaluate the probability of any function. This is a very strange concept so we are going to re-write this in a different and non-parametric way. Lets think of what a function actually is, or rather, how we can observe a function. What we can see is the *values* of a function, so how about we actually parametrise the function directly by its output values? To do so let us alter our representation slightly, lets write down the output of the function as a variable,

$$f_i = f(x_i). \tag{33}$$

Now because we do not know what f_i is we are going to treat this as a random variable. In specific we are going to make the assumption that this variable is Gaussian such as,

$$f_i \sim \mathcal{N}(\mu(x_i), k(x_i)), \tag{34}$$

where $\mu(x_i)$ and $k(x_i)$ are functions that evaluates the mean and the variance of the random variable as a function of the input value. Now a function can be evaluate in more than one place, so what we will say is that not only is each of the function values normally distributed, *they are jointly normally distributed*. What this means is that if we now have a set of input locations $\mathbf{x} = [x_1, \ldots, x_N]$ they induce the following set of random variable,

$$\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu(x_1) \\ \mu(x_2) \\ \vdots \\ \mu(x_N) \end{bmatrix}, \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_N) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_N) \\ \vdots & \vdots & \cdots & \vdots \\ k(x_N,x_1) & k(x_N,x_2) & \cdots & k(x_N,x_N) \end{bmatrix}\right), \tag{35}$$

where $k(\cdot,\cdot)$ is a function that computes the covariance between of the function values at two locations[14] As it is possible to evaluate the function at infinitely many locations the cardinality of the set is actually the *uncountable infinity*. Now we will use one very important result from the Gaussian identities namely the *marginal property* of the Gaussian distribution. If we want the marginal distribution of a sub-set of variables where the whole set is jointly Gaussian we only need to "pick" the corresondping parts of the mean and the co-variance matrix. What this means is that somewhere there exists an infinite Gaussian distribution, where each dimension of the distribution corresponds to a random varible but we can project this to *any* finite subspace by just evaluating or picking the relevant part of the mean and co-variance. This infinite dimensional object is called a *Gaussian process*. This marginal property is incredibly powerful as it implies that whatever we do in the finite case can be considered a projection from the infinite, therefore we can think of Eq. 35 as a distribution over the function evaluated at a finite set of locations $\mathbf{x}$ which are consistent with the whole infinite input space.

Using a Gaussian process as a prior over the function implies that we need to specify a mean function $\mu(\cdot)$ and a co-variance function $k(\cdot,\cdot)$. For now let us use the constant zero function as mean and focus on the co-variance function. The co-variance function describes how the function outputs varies together as a function of their inputs. Its not free to be any function as the matrix evaluate on any subset of the input space still has to be a valid covariance matrix. The class of functions is the same as the class of valid kernel functions, i.e. it need to define an inner-product space if you are interested in generalisations of Euclidean spaces you can read more about this here. Without going into too much depth on what consistutes a valid covariance function the main things that you need to think about is that the matrix needs to be positive-definite, i.e. all its eigenvalues needs to be positive[15]. One simple function that is guaranteed to do this is what is called the exponentiated quadratic, the squared exponential or the radial-basis kernel,

$$k(x, x') = \sigma^2 e^{\frac{-(x-x')^{\mathrm{T}}(x-x')}{\ell^2}} \tag{36}$$

where σ^2 is referred to as the variance and ℓ as the length-scale of the function.

4.2.1 Implementation

In order to work with a GP it makes sense to write a bit more sensible structured code and break up a few things into functions. The first thing we want to implement is a function that allows us to compute the covariance matrix. It could be nice to try and modularise this so that you can easily use your code with several different covariance functions.

[14] through the definition of the variance it therefore needs to compute the variance of the random varialbe if both arguments are the same.

[15] its a fun to show why this has to be the case

Code

```
from scipy.spatial.distance import cdist

def rbf_kernel(x1, x2, varSigma, lengthscale):
    if x2 is None:
        d = cdist(x1, x1)
    else:
        d = cdist(x1, x2)

    K = varSigma*np.exp(-np.power(d, 2)/lengthscale)

    return K
```

Now when we have written the our covariance function we are ready to sample from our prior. Now let us try and sample from a GP defined with a zero mean and the exponentiated quadratic defined above. We will first look at functions with one-dimensional inputs so the infinite process will be indexed by $\mathbb{R}$. The first thing we will do is to decide which marginal of the real-line that we will look at. We will refer to this as the index set. Lets now compute this marginal of the process, which being finite is just a Gaussian distribution, and plot a couple of samples from this.

Code

```
# choose index set for the marginal
x = np.linspace(-6, 6, 200).reshape(-1, 1)
# compute covariance matrix
K = rbf_kernel(x, None, 1.0, 2.0)
# create mean vector
mu = np.zeros(x.shape)

# draw samples 20 from Gaussian distribution
f = np.random.multivariate_normal(mu, K, 20)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, f.T)
```

Run the code above and test a few different parameters for the covariance function, how does the behaviour of the samples change? The important thing to understand is that even if the process is infinite, as are the number of values the function can be evaluated across, due to the consistency of a GP we can decide to only look at a finite subset of the process. This is what we define using `x`. Test to increase and decrease the cardinality of the index set.

Sampling from the prior is very important as it allows us to see what assumptions we can encode with the parameters. Importantly, the GP places non-zero probability on every function so if you sample for long enough everything will appear, with a few samples you are only seeing the most likely things.

Now lets try to change the covariance function and see how the samples change, lets implement three more covariance functions, a white-noise, a linear and a periodic covariance.

Code

```
def lin_kernel(x1, x2, varSigma):
    if x2 is None:
        return varSigma*x1.dot(x1.T)
    else:
        return varSigma*x1.dot(x2.T)

def white_kernel(x1, x2, varSigma):
    if x2 is None:
        return varSigma*np.eye(x1.shape[0])
    else:
        return np.zeros(x1.shape[0], x2.shape[0])

def periodic_kernel(x1, x2, varSigma, period, lenthScale):
    if x2 is None:
        d = cdist(x1, x1)
    else:
        d = cdist(x1, x2)

    return varSigma*np.exp(-(2*np.sin((np.pi/period)*np.sqrt(d))**2)/lengthScale**2)
```

Generate samples from GPs defined by the different kernels and see what type of behaviour the prior now encodes. There are several operations you are allowed to do to a kernel while in [5] there is a list of operations. What you can try is to add different kernels together, you can also multiply them together. Try this out and look at the samples. The key thing that I want you to see is how much larger the class of beliefs that you can specify with GPs compared to the lines that we did in the previous lab[16].

4.3 Posterior

Now when we are happy about our prior it is time to move to the posterior distribution. As the GP is a non-parametric model where we dirctly model the output rather than a specific set of parameters the posterior takes a form much more similar to the predictive posterior in the linear regression case. Deriving the posterior is a very simple proceedure making use of the definition of the GP. In specific a GP is defined as a infinite collection of random variables which are all *jointly* Gaussian distributed. So lets make use of this. Lets assume that we have observed data $\mathcal{D} = \{y_i, x_i\}_{i=1}^N$ and now we want to predict what the output of the function is at locations $\mathbf{x}_* = \{x_{*1}, \ldots, x_{*M}\}$. Now the union of the input locations specifies the index set of the infinite process meaning that we can write up the joint distribution as,

$$
\begin{aligned}
&p(f_1, \ldots, f_N, f_{*1} \ldots, f_{*M} | \mathbf{x}, \mathbf{x}_*, \boldsymbol{\theta}) \\
&= \mathcal{N}\left(\begin{bmatrix} \mu(x_1) \\ \vdots \\ \mu(x_N) \\ \mu(x_{*1}) \\ \vdots \\ \mu(x_{*M}) \end{bmatrix}, \begin{bmatrix} k(x_1,x_1) & \cdots & k(x_1,x_N) & k(x_1,x_{*1}) & \cdots & k(x_1,x_{*M}) \\ \vdots & \ddots & \vdots & \vdots & \cdots & \vdots \\ k(x_N,x_1) & \cdots & k(x_N,x_N) & k(x_N,x_{*1}) & \cdots & k(x_N,x_{*M}) \\ k(x_{*1},x_1) & \cdots & k(x_{*1},x_N) & k(x_{*1},x_{*1}) & \cdots & k(x_{*1},x_{*M}) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ k(x_{*M},x_1) & \cdots & k(x_{*M},x_N) & k(x_{*M},x_{*1}) & \cdots & k(x_{*M},x_{*M}) \end{bmatrix} \right)
\end{aligned} \tag{37}
$$

where $\mu(\cdot)$ and $k(\cdot,\cdot)$ are the mean and covariance function and θ are the parameters of the latter. Now this is just the prior distribution over indexed at the subset of the data and where we are interested in evaluating the function. We are now interested in reaching the posterior distribution over the index set $\mathbf{x}_*$. Maybe this

[16] the plots that you have above should be compared with the plots that you did by sampling from the prior in the previous lab

feels a bit daunting at first but lets think of what we actually have and what we want to reach. We have a joint distribution and we want to get the conditional distribution over the index set $\mathbf{x}_*$ given the index set $\mathbf{x}$. If we apply the product rule and factorise the joint distribution we get,

$$p(\mathbf{f}, \mathbf{f}_*|\mathbf{x}, \mathbf{x}_*, \boldsymbol{\theta}) = p(\mathbf{f}_*|\mathbf{f}, \mathbf{x}, \mathbf{x}_*, \theta)p(\mathbf{f}|\mathbf{x}, \theta). \tag{38}$$

Now this is just the factorisation of a Gaussian distribution and we actually proved this in Lecture 3. So we can just use those results and write down what the conditional distribution that we are interested in is.

$$p(\mathbf{f}_*|\mathbf{f}, \mathbf{x}, \mathbf{x}_*, \theta) = \mathcal{N}(\mu_{\mathbf{x}_*|\mathbf{x}}, K_{\mathbf{x}_*|\mathbf{x}}) \tag{39}$$

$$\mu_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x})k(\mathbf{x}, \mathbf{x})^{-1}\mathbf{f} \tag{40}$$

$$K_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{x})k(\mathbf{x}, \mathbf{x})^{-1}k(\mathbf{x}, \mathbf{x}_*), \tag{41}$$

where we have assumed that the mean function $\mu(\cdot)$ is the constant zero function. Compare this with Eq. 64 in the summary document can you see the similarity? Now let us generate some data and sample from this distribution. Lets generate some data from a noisy sine wave.

Code

```
N = 5
x = np.linspace(-3.1,3,N)
y = np.sin(2*np.pi/x) + x*0.1 + 0.3*np.random.randn(x.shape[0])
x = np.reshape(x,(-1,1))
y = np.reshape(y,(-1,1))
x_star = np.linspace(-6, 6, 500)
```

Now we are likely to use compute the posterior quite a few times for different index-sets so its probably worthwhile to dedicate a function for this. Something like the one below would probably be suitable.

Code

```
def gp_prediction(x1, y1, xstar, lengthScale, varSigma, noise):

    k_starX = rbf_kernel(xstar,x1,lengthScale,varSigma,noise)
    k_xx = rbf_kernel(x1, None, lengthScale, varSigma, noise)
    k_starstar = rbf_kernel(xstar,None,lengthScale,varSigma,noise)

    mu = k_starX.dot(np.linalg.inv(k_xx)).dot(y1)
    var = k_starstar - (k_starX).dot(np.linalg.inv(k_xx)).dot(k_starX.T)

    return mu, var, xstar
```

Once we have the the functionality to compute predictive posterior we can now sample from this.

Code

```
Nsamp = 100
mu_star, var_star, x_star = gp_prediction(x1, y1, x, lengthScale, varSigma, noise)
fstar = np.random.multivariate_normal(mu_star, var_star, Nsamp)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x_star, f_star.T)
ax.scatter(x1, y1, 200, 'k', '*', zorder=2)
```

If things works out as it should you should achieve a plot looking something similar to the one in Figure 8, As you can all the samples from the function goes through the data.

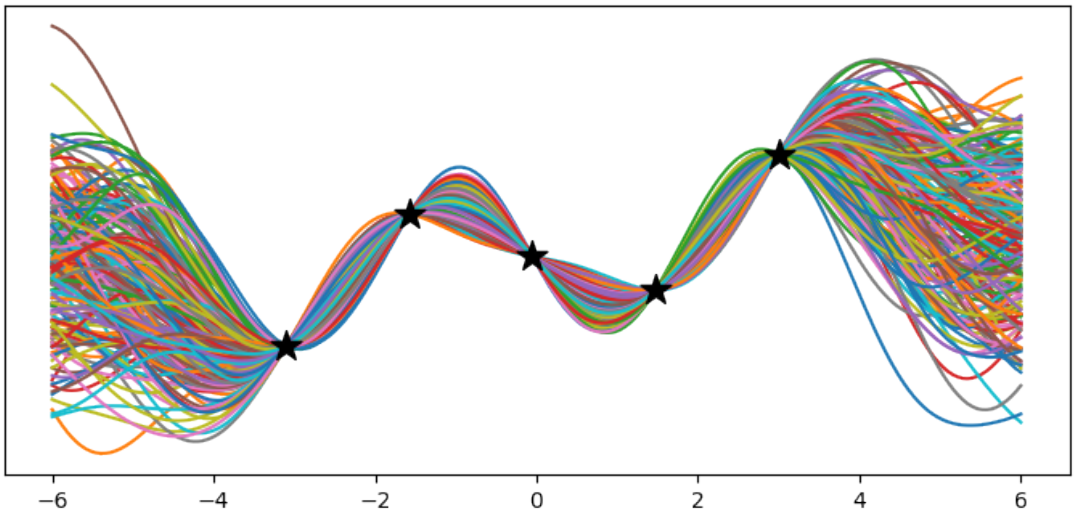

Figure 8: The above image shows samples from the predictive posterior of a Gaussian process prior specified by a exponentiated quadratic kernel. As you can see all the samples passes exactly through the data.

Rather than plotting the samples we can also show the mean and the variance around each point in the index set. This should generate a plot looking something similar to the one shown in Figure 9. Now we are pretty much there, we have shown how we can formulate our prior and then reach our posterior. But there is one little thing remaining, we haven't talked about the likelihood at all Eq. 31. Now it turns out that this is actually trivial to include, what we have observed is y but what we have shown is the plots over f. The relationship between these two variables was,

$$y_i = f(x_i) + \epsilon,$$

where ϵ followed a Gaussian distribution. So importantly this is simply some added randomness between the two variables. Importantly the randomness is Gaussian, and we already have a formulation of ever function instantiation that is jointly Gaussian now we have some additional independent stochasticity to include. So what we need to do is to also account for this. This can easily be done by altering the joint distribution in Eq. 37 to be,

$$\begin{aligned} &p(y_1, \dots, y_N, f_{*1} \dots, f_{*M} | \mathbf{x}, \mathbf{x}_*, \boldsymbol{\theta}) \\ &= \mathcal{N}\left(\begin{bmatrix} \mu(x_1) \\ \vdots \\ \mu(x_N) \\ \mu(x_{*1}) \\ \vdots \\ \mu(x_{*M}) \end{bmatrix}, \begin{bmatrix} k(x_1, x_1) + \frac{1}{\beta} & \cdots & k(x_1, x_N) & k(x_1, x_{*1}) & \cdots & k(x_1, x_{*M}) \\ \vdots & \ddots & \vdots & \vdots & \cdots & \vdots \\ k(x_N, x_1) & \cdots & k(x_N, x_N) + \frac{1}{\beta} & k(x_N, x_{*1}) & \cdots & k(x_N, x_{*M}) \\ k(x_{*1}, x_1) & \cdots & k(x_{*1}, x_N) & k(x_{*1}, x_{*1}) & \cdots & k(x_{*1}, x_{*M}) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ k(x_{*M}, x_1) & \cdots & k(x_{*M}, x_N) & k(x_{*M}, x_{*1}) & \cdots & k(x_{*M}, x_{*M}) \end{bmatrix}\right) \end{aligned} \tag{42}$$

This leads to a tiny change to the predictive posterior in Eq. 41 as we now only need to include the

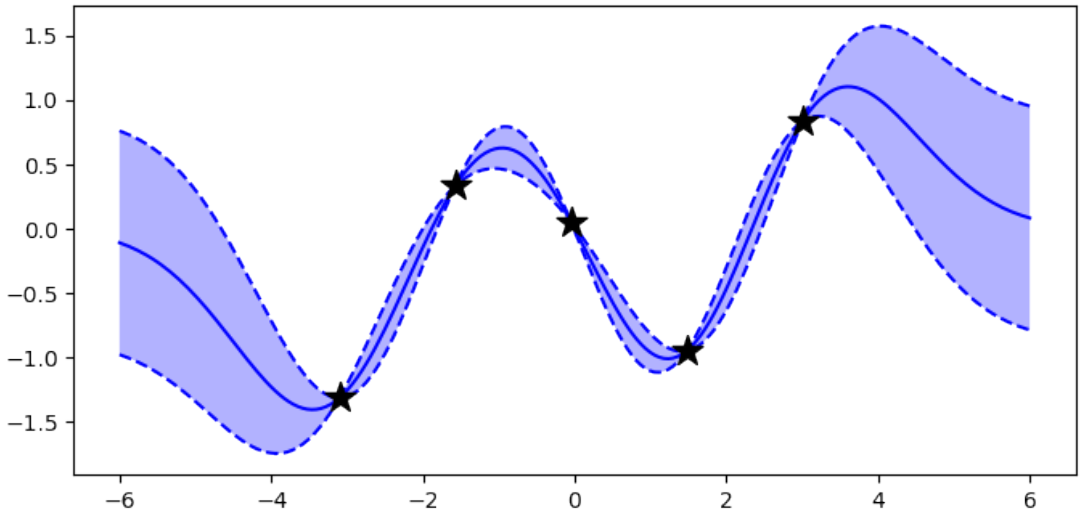

Figure 9: The above image shows the mean prediction and one standard deviation around the mean. You can generate the plot by using the nice command `np.fill_between()`.

independent noise term as,

$$p(\mathbf{f}_*|\mathbf{y}, \mathbf{x}, \mathbf{x}_*, \theta) = \mathcal{N}(\mu_{\mathbf{x}_*|\mathbf{x}}, K_{\mathbf{x}_*|\mathbf{x}}) \tag{43}$$

$$\mu_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x}) \left(k(\mathbf{x}, \mathbf{x} + \frac{1}{\beta}\mathbf{I} \right)^{-1} \mathbf{y} \tag{44}$$

$$K_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{x}) \left(k(\mathbf{x}, \mathbf{x} + \frac{1}{\beta}\mathbf{I} \right)^{-1} k(\mathbf{x}, \mathbf{x}_*). \tag{45}$$

Think about this and justify to yourself that this makes sense. If you now generate samples from the above distribution you will see that the samples no longer necessarily goes through the data. We are now allowing some of the variations in the data to be explained by noise instead of by signal. What you can do now is to play around with different covariance functions and different parameters etc. and get an intuitive feeling for what we have done.

4.4 Summary

Hopefully you have gotten to the end of this worksheet and realised how simple Gaussian processes actually are. They do really cool stuff and they are called all-sorts of fancy names but really at the core the only thing it really is is one big Gaussian where the index set have been associated with and interesting interpretation. If this hasn't become clear just yet, take a step back and play around with them a bit more, alter the covariance function, draw some samples write up the derivation. Hopefully after some time you will feel that this all makes sense. So even though the result looked quite different, what new stuff have we actually done? Very little. Think about it, we have walked through exactly the same procedure for a third week in a row, it was just a different prior. The process and the language is all the same, and that is the real benefit of thinking about machine learning in a Bayesian perspective, its a consistent story. This story will repeat itself once more, but after that I hope that this idea should have stuck.

References

[1] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian Processes for Data-Efficient Learning in Robotics and Control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, stat.ML(2):408–423, February 2015.

[2] M. Kaiser, C. Otte, T. Runkler, and C. H. Ek. Bayesian alignments of warped multi-output gaussian processes. In *Advances in Neural Information Processing Systems 32, [NIPS Conference, Montreal, Quebec, Canada, December 3 - December 8, 2018]*, 2018.

[3] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[4] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. Bayesian optimization in alphago. *CoRR*, 2018.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] Neil D Lawrence. Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *Journal of Machine Learning Research*, 6:1783–1816, 2005.

[7] Keith Grochow, Steven L Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *SIGGRAPH '04: SIGGRAPH 2004 Papers*, August 2004.

[8] Raquel Urtasun, David J Fleet, and P. Fua. 3D people tracking with Gaussian process dynamical models. *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, 1:238–245, 2006.

[9] Steindór Sæmundsson, Katja Hofmann, and Marc Peter Deisenroth. Meta reinforcement learning with latent variable gaussian processes. *CoRR*, 2018.

[10] K. B. Petersen and M. S. Pedersen. The Matrix Cookbook, November 2012. Version 20121115.

[11] Iain Murray and Zoubin Ghahramani. A note on the evidence and Bayesian Occam's razor. Technical Report GCNU-TR 2005-003, August 2005.

[12] J. Močkus. On bayesian methods for seeking the extremum. In G. I. Marchuk, editor, *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*, pages 400–404, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

5 Unsupervised Learning

Abstract

In this lab we will again look at the modelling proceedure that we have gone through over the last couple of weeks. However we will do this in a slightly different setting doing something often referred to as *unsupervised learning*. Personally I think this is a bit of a misnomer as it is easy to think that unsupervised means that we do not place any demands on the learning proceedure. This is not the case at all, if it was it wouldn't be possible to learn. Let us take the example of training a dog, supervised learning would be when you throw a ball and you want the dog to figure out that it should fetch it, while unsupervised learning is when you just let the dog out of the house and it can do a little bit as it wants. However, importantly, you still have demands on the behaviour of the dog, just because you let it out of the house and you haven't given it any explicit instructions it doesn't mean it is now OK for the dog to eat your cat. So even though the instructions and demands are more subtle they are still there, otherwise the exercise is pointless.

The aim of unsupervised learning is to observe a set of data $\mathbf{Y}$ and make an assumption that this data have been generated from a latent representation $\mathbf{X}$ through some mapping $f(\cdot)$ such that,

$$\mathbf{y}_i = f(\mathbf{x}_i). \tag{46}$$

Now our task is to recover both the latent representation and the mapping. This seems like a really odd problem at first. A very simple solution to it is to say that the mapping is the identity meaning that the latent and the observed data is the same. So we have to add some form of supervision, some form of direction that tells us what it is that we are looking for. One very general scenario is that we might look for a lower dimensional representation of the data. This is often beneficial as many algorithms scale very badly with dimension so if we can reduce the dimensionality of $\mathbf{Y}$ from say $\mathbb{R}^D$ to $\mathbb{R}^Q$ if $Q < D$ then we can apply our expensive algorithm on the latent representation instead. But there might be lots of other demands as well where for one reason or the other we wish to reorganise our data in a different manner than it was presented.

The important thing is that the problem that we are trying to solve is badly constrained, there are clearly so many possible solutions so that we need to somehow encode our preference towards certain solutions. Now here comes a slightly different interpretations of beliefs namely preference. They actually play a very similar role as what happened during learning in the previous scenarios we worked on was that given two equal solutions under the data we prefered the one we believed in more, now that is just the same as a preference. So another way to think of priors is as preferences over the different hypothesis.

We are going to use the same regression model in this task as the previous two labs, we will have the same likelihood function as we have used for both previous regression tasks. We will then use two different types of priors for the mapping, both the linear and the Gaussian process prior to recover different representations. As we are now starting to build a bit more complicated models several of the computations can no longer be done in closed form. Later in the unit we will see how we can adress these computations in a principled manner. For now you will just have to trust me on some of these. We will use the same Gaussian likelihood as we have used in the last two labs,

$$p(\mathbf{Y}|\mathbf{F}) = \mathcal{N}(\mathbf{F}, \beta^{-1}\mathbf{I}). \tag{47}$$

5.1 Principle Component Analysis

We will start with a model where we use a linear model for our mapping from the latent to the observed space. We will use the same model as we did in the linear regression case with the only difference being that as we do not know the input we are going to specify a preference/belief over this. Note that throughout this derivation I have switched back to Carl notation rather than the Bishop one that was used during the lecture. First we have a likelihood function which describes how likely observations is under our model,

$$p(\mathbf{Y}|\mathbf{W}, \mathbf{X}) = \mathcal{N}(\mathbf{X}\mathbf{W} + \mu, \beta^{-1}\mathbf{I}), \tag{48}$$

where μ is a constant offset. We will then use a Gaussian prior over both the weights $\mathbf{W}$ and the latent coordinates $\mathbf{X}$. This leads to the following joint distribution,

$$p(\mathbf{Y}\mathbf{W}, \mathbf{X}) = p(\mathbf{Y}|\mathbf{W}, \mathbf{X})p(\mathbf{X})p(\mathbf{W}). \tag{49}$$

Our aim is now to derive the posterior distribution over the unknown parameters $\mathbf{W}$ and $\mathbf{X}$. To do so we need to marginalise out these parameters from the joint defined above. This means we need to compute,

$$p(\mathbf{Y}) = \int p(\mathbf{Y}|\mathbf{W}, \mathbf{X})p(\mathbf{X})p(\mathbf{W}). \tag{50}$$

Sadly the following integral is not tractable, we cannot solve it for both $\mathbf{W}$ and $\mathbf{X}$. In order to proceed we are going to integrate out one of the variables and take a point estimate for the other. Now we could either choose $\mathbf{W}$ or $\mathbf{X}$ to optimise for so which one should we choose? Well lets say that we have N data points and we are looking for a Q dimensional latent representation. This means that $\mathbf{W} \in \mathbb{R}^{D\times Q}$ and $\mathbf{X}^{N\times Q}$ assuming that $N > D$ it makes sense to treat as many of the random variables with principle as we can and integrate out $\mathbf{X}$ while we optimise $\mathbf{W}$. Another motivation is that if we are given new data and we want to infer its latent location then we want to have the posterior distribution $p(\mathbf{x}|\mathbf{y})$ and we can only reach this distribution if we marginalise out the latent space.

We are going to do the computations in a bit of a different way compared to before. First we are going to make use of the fact that the product of two Gaussians is also a Gaussian. So if we multiply our likelihood and our prior we should now again have a Gaussian,

$$p(\mathbf{X}, \mathbf{Y}|\mathbf{W}) = p(\mathbf{Y}|\mathbf{W}, \mathbf{X})p(\mathbf{X}). \tag{51}$$

Now we know both of the terms on the left hand-side of the expression,

$$p(\mathbf{Y}|\mathbf{W}, \mathbf{X}) = \mathcal{N}(\mathbf{X}\mathbf{W} + \mu, \beta^{-1}\mathbf{I}) \tag{52}$$
$$p(\mathbf{X}) = \mathcal{N}(\mathbf{0}, \mathbf{I}). \tag{53}$$

We will now derive the joint distribution of a pair of points, $\mathbf{z}$ and $\mathbf{x}$ rather than the full data matrix. As we know that the product is a Gaussian we can write up the definition of the terms in this joint distribution,

$$p(\mathbf{y}, \mathbf{x}|\mathbf{W}) = \mathcal{N}\left(\begin{bmatrix} \mathbb{E}[\mathbf{y}] \\ \mathbb{E}[\mathbf{x}] \end{bmatrix}, \begin{bmatrix} \mathbb{E}[(\mathbf{y} - \mathbb{E}[\mathbf{y}])(\mathbf{y} - \mathbb{E}[\mathbf{y}])^{\mathrm{T}}] & \mathbb{E}[(\mathbf{y} - \mathbb{E}[\mathbf{y}])(\mathbf{x} - \mathbb{E}[\mathbf{x}])^{\mathrm{T}}] \\ \mathbb{E}[(\mathbf{x} - \mathbb{E}[\mathbf{x}])(\mathbf{y} - \mathbb{E}[\mathbf{y}])^{\mathrm{T}}] & \mathbb{E}[(\mathbf{x} - \mathbb{E}[\mathbf{x}])(\mathbf{x} - \mathbb{E}[\mathbf{x}])^{\mathrm{T}}] \end{bmatrix}\right). \tag{54}$$

We will now walk through each of the expectations in turn to specify the joint distribution above. Once we have the joint we can use the by know well know identities to get the marginal and the posterior distribution that we are looking for. Let us start with the means,

$$\mathbb{E}[\mathbf{x}] = \mathbf{0} \tag{55}$$
$$\mathbb{E}[\mathbf{y}] = \mathbb{E}[\mathbf{W}\mathbf{x} + \mu + \epsilon] = \mathbb{E}[\mathbf{W}\mathbf{x}] + \mathbb{E}[\mu] + \mathbb{E}[\epsilon] \tag{56}$$
$$= \mathbf{W}\mathbb{E}[\mathbf{x}] + \mathbb{E}[\mu] + \mathbb{E}[\epsilon] = \mathbf{0}\mathbf{W} + \mu + \mathbf{0} \tag{57}$$
$$= \mu \tag{58}$$

The mean of the latent variable is easy as it come directly through its definition. The one for the observed data comes through the model of the observed data so we first need to rewrite it in terms of the model. Now when we have the means we can move on to deriving the elements of the covariance matrix,

$$\mathbb{E}[(\mathbf{x} - \mathbb{E}[\mathbf{x}])(\mathbf{y} - \mathbb{E}[\mathbf{y}])^{\mathrm{T}}] = \mathbb{E}[(\mathbf{x} - 0)(\mathbf{y} - \mu)^{\mathrm{T}}] \tag{59}$$
$$= \mathbb{E}[\mathbf{x}(\mathbf{W}\mathbf{x} + \mu + \epsilon - \mu)^{\mathrm{T}}] \tag{60}$$
$$= \mathbb{E}[\mathbf{x}(\mathbf{W}\mathbf{x} + \epsilon)^{\mathrm{T}}] = \mathbb{E}[\mathbf{x}\left(\mathbf{W}\mathbf{x}\right)^{\mathrm{T}} + \mathbf{x}\epsilon^{\mathrm{T}}] \tag{61}$$
$$= \mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}] + \mathbb{E}[\mathbf{x}]\mathbb{E}[\epsilon] = \mathbb{E}[(\mathbf{x} - \mathbf{0})(\mathbf{x} - \mathbf{0})^{\mathrm{T}}]\mathbf{W}^{\mathrm{T}} + 0 \cdot 0 \tag{62}$$
$$= \mathbf{I}\mathbf{W}^{\mathrm{T}} = \mathbf{W}^{\mathrm{T}}. \tag{63}$$

We can then derive the final element of the joint distribution the variance of the observed data,

$$\begin{aligned}
\mathbb{E}[(\mathbf{y}-\mathbb{E}[\mathbf{y}])(\mathbf{y}-\mathbb{E}[\mathbf{y}])^{\mathrm{T}}] &= \mathbb{E}[(\mathbf{W}\mathbf{x}+\mu+\epsilon-\mu)(\mathbf{W}\mathbf{x}+\mu+\epsilon-\mu)^{\mathrm{T}}] && (64)\\
&= \mathbb{E}[(\mathbf{W}\mathbf{x}+\epsilon)(\mathbf{W}\mathbf{x}+\epsilon)^{\mathrm{T}}] = \mathbb{E}[\mathbf{W}\mathbf{x}\,(\mathbf{W}\mathbf{x})^{\mathrm{T}} + \mathbf{W}\mathbf{x}\epsilon^{\mathrm{T}} + \epsilon\mathbf{W}\mathbf{x}^{\mathrm{T}} + \epsilon\epsilon^{\mathrm{T}}] && (65)\\
&= \mathbb{E}[\mathbf{W}\mathbf{x}\mathbf{x}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}] + \mathbb{E}[\mathbf{W}\mathbf{x}\epsilon^{\mathrm{T}}] + \mathbb{E}[\epsilon(\mathbf{W}\mathbf{x})^{\mathrm{T}}] + \mathbb{E}[\epsilon\epsilon^{\mathrm{T}}] && (66)\\
&= \mathbf{W}\mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}]\mathbf{W}^{\mathrm{T}} + \mathbf{W}\mathbb{E}[\mathbf{x}]\mathbb{E}[\epsilon] + \mathbb{E}[\epsilon]\mathbb{E}[\mathbf{x}^{\mathrm{T}}]\mathbf{W}^{\mathrm{T}} + \mathbb{E}[(\epsilon-0)(\epsilon-0)^{\mathrm{T}}] && (67)\\
&= \mathbf{W}\mathbf{I}\mathbf{W}^{\mathrm{T}} + \mathbf{W}0 + 0\mathbf{W}^{\mathrm{T}} + \sigma^2\mathbf{I} && (68)\\
&= \mathbf{W}\mathbf{W}^{\mathrm{T}} + \sigma^2\mathbf{I}. && (69)
\end{aligned}$$

Now we have all the elements and can write up the final joint distribution as,

$$p(\mathbf{y},\mathbf{x}|\mathbf{W}) = \mathcal{N}\left(\begin{bmatrix}\mu\\ \mathbf{0}\end{bmatrix}, \begin{bmatrix}\mathbf{W}\mathbf{W}^{\mathrm{T}}+\sigma^2\mathbf{I} & \mathbf{W}\\ \mathbf{W}^{\mathrm{T}} & \mathbf{I}\end{bmatrix}\right), \tag{70}$$

from which we can derive the marginal over the observed data and the conditional distribution over the latent space using the Gaussian identities as,

$$p(\mathbf{y}|\mathbf{W}) = \mathcal{N}(\mu, \mathbf{W}\mathbf{W}^{\mathrm{T}}+\sigma^2\mathbf{I}) \tag{71}$$

$$p(\mathbf{x}|\mathbf{y},\mathbf{W}) = \mathcal{N}(\mathbf{W}^{\mathrm{T}}\left(\mathbf{W}\mathbf{W}^{\mathrm{T}}+\sigma^2\mathbf{I}\right)^{-1}(\mathbf{y}-\mu), \mathbf{I}-\mathbf{W}^{\mathrm{T}}(\mathbf{W}\mathbf{W}^{\mathrm{T}}+\sigma^2\mathbf{I})^{-1}\mathbf{W}). \tag{72}$$

Now we have everything that we need in order to proceed. Our first goal is to take a point estimate of the weights of the mapping. Now there are several ways that we can do this, one option would be to take the derivatives of the marginal distribution and maximise this for the data. However, in this case it turns out that there is actually a closed form solution for the weights in the limit when $\beta \to \infty$. We will cheat a little bit and use this as a solution. We can get this by solving a simple eigenvalue problem using the code below.

Code

```
def MLW(Y,q):
    v,W = np.linalg.eig(np.cov(Y.T))
    idx = np.argsort(np.real(v))[::-1][:q]
    return np.real(W[:,idx])
```

Once we have the $\mathbf{W}$ that maximises the marginal we can use this in the posterior distribution and recover the latent space. Now we have everything that we need and it is time to generate some data and test the approach. We will generate a simple spiral in two dimensions and then we will draw a random mapping that embedds this in $\mathbb{R}^{1017}$.

[17] the code below is a bit messy as I had to save and load the data, so you have to do more than just copy paste the code

Code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from tempfile import TemporaryFile

# maximum likelihood solution to W
def MLW(x,q):
    v,w = np.linalg.eig(np.cov(x.T))
    idx = np.argsort(np.real(v))[::-1][:q]
    return np.real(w[:,idx])

# posterior distribution of latent variable
def posterior(w, x, mu_x, beta):
    A = np.linalg.inv(w.dot(w.T)+1/beta*np.eye(w.shape[0]))
    mu = w.T.dot(A.dot(x-mu_x))
    varSigma = np.eye(w.shape[1]) - w.T.dot(A.dot(w))

    return mu, varSigma

# Generate a spiral
t = np.linspace(0,3*np.pi,100)
x = np.zeros((t.shape[0],2))
x[:,0] = t*np.sin(t)
x[:,1] = t*np.cos(t)

# pick a random matrix that maps to Y
w = np.random.randn(10,2)
y = x.dot(w.T)
y += np.random.randn(y.shape[0],y.shape[1])
mu_y = np.mean(y,axis=0)

# get maximim likelihood solution of W
w = MLW(y,2)

# compute predictions for latent space
xpred = np.zeros(x.shape)
varSigma = []
for i in range(0, y.shape[0]):
    xpred[i,:], varSigma = posterior(w, y[i,:], mu_y, 1/2)

np.save('05tmp1.npy', xpred)
np.save('05tmp2.npy', varSigma)
```

When we have got the code working to describe the joint distribution we can now use the posterior and map back all the training data-points to the latent space and look at how their distribution looks like.

Code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

xpred = np.load('05tmp1.npy')
varSigma = np.load('05tmp2.npy')

# generate density
N = 300
x1 = np.linspace(np.min(xpred[:,0]), np.max(xpred[:,0]), N)
x2 = np.linspace(np.min(xpred[:,1]), np.max(xpred[:,1]), N)
x1p, x2p = np.meshgrid(x1, x2)
pos = np.vstack((x1p.flatten(), x2p.flatten())).T

# compute posterior
Z = np.zeros((N,N))
for i in range(0, xpred.shape[0]):
    pdf = multivariate_normal(xpred[i,:].flatten(), varSigma)
    Z += pdf.pdf(pos).reshape(N,N)

# plot figuresy
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(121)
ax.scatter(xpred[:,0], xpred[:,1])
ax.set_xticks([])
ax.set_yticks([])
ax = fig.add_subplot(122)
ax.imshow(Z,cmap='hot')
ax.set_ylim(ax.get_ylim()[::-1])
ax.set_xticks([])
ax.set_yticks([])

#IGNORE THIS
plt.tight_layout()
plt.savefig(path, transparent=True)
return path
```

Now as you can see the data looks similar to what we expected, we do recover the spiral as we expected. Now as it turns out we could have actually gotten any rotation of the spiral as the marginal distribution over $\mathbf{W}$ is actually invariant to a rotation. To see this, you can rotate the matrix,

$$\tilde{\mathbf{W}} = \mathbf{W}\mathbf{R} \tag{73}$$

$$\tilde{\mathbf{W}}\tilde{\mathbf{W}}^{\mathrm{T}} = \mathbf{W}\mathbf{R}(\mathbf{W}\mathbf{R})^{\mathrm{T}} \tag{74}$$

$$\mathbf{W}\underbrace{\mathbf{R}\mathbf{R}^{\mathrm{T}}}_{\mathbf{I}}\mathbf{W}^{\mathrm{T}} = \mathbf{W}\mathbf{W}^{\mathrm{T}}. \tag{75}$$

Because a rotation matrix is an orthonormal matrix $\mathbf{R}^{\mathrm{T}} = \mathbf{R}^{-1}$. Later in the unit when we have seen approximate inference we can go back to this model and solve it in a more principled manner. If you already now want to understand in more detail how you solve this in a more principled manner you can read [5] Chapter 12.2.

A different way to solve the maximisation is to take a gradient based approach. This means that we will

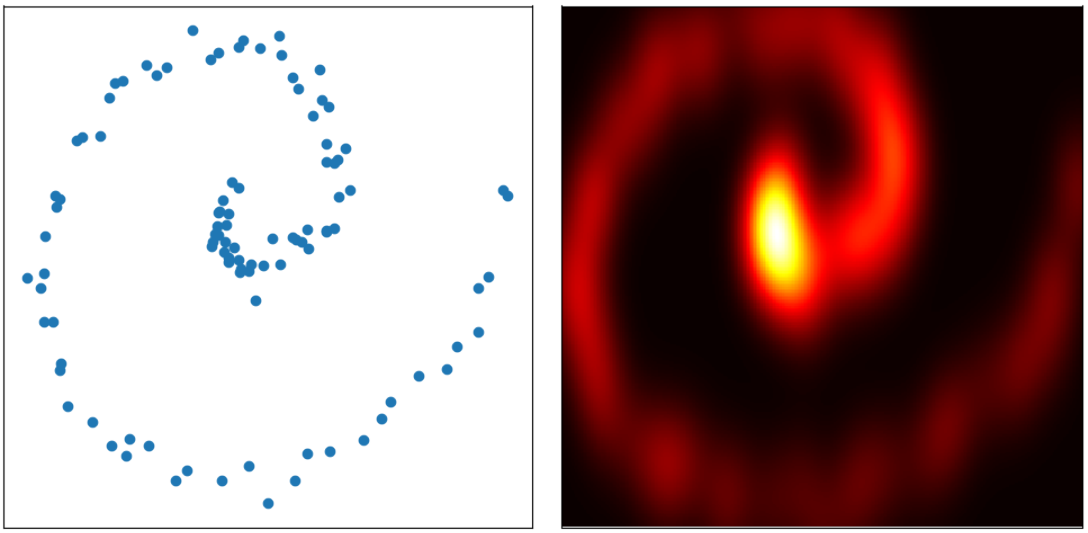

Figure 10: The figure above shows the PCA solution to a 2D sprial embedded in $\mathbb{R}^{10}$. The right plot shows the posterior distribution of each individual point in the training data-set super-imposed on each other.

maximise the marginal distribtuion $p(\mathbf{Y}|\mathbf{W})$ using an iterative gradient based method.

5.2 Gaussian Process Latent Variable Model

We can also use a GP prior over the mapping f. If we do this it lead to a model called the Gaussian Process Latent Variable Model [6]. This is a really powerful model that has been used in a large range of different settings such as animantion [7], pose estimation [8] and reinforcement learning [9]. Using a GP changes the model slightly as our prior now need to be indexed by the locations that we want to evaluate it at. This means that we have the following joint distribution,

$$p(\mathbf{Y}, \mathbf{X}, \mathbf{F}, \theta) = p(\mathbf{Y}|\mathbf{F})p(\mathbf{F}|\mathbf{X}, \theta)p(\mathbf{X})p(\theta). \tag{76}$$

Now we will do a couple of assumptions to make things a little bit easier. First we will say that given the input the output dimensions are independent. This might sounds strange at first, but think about it again, what it really says is that if I know the input location where I want to query the output of the function the input encodes all the information necessary to decide on the output. This means that we can factorise the prior as this,

$$p(\mathbf{F}|\mathbf{X}, \theta) = \prod_{i=d}^{D} p(\mathbf{f}_d|\mathbf{X}, \theta). \tag{77}$$

The noise assumption that we have done for the likelihood also implies that given the output of the function the data points and the dimensions are independent,

$$p(\mathbf{Y}|\mathbf{F}) = \prod_{i=1}^{N}\prod_{d=1}^{D} p(y_{id}|f_{id}) = \prod_{i=1}^{N}\prod_{d=1}^{D} \mathcal{N}(y_{id}|f_{id}, \beta^{-1}). \tag{78}$$

Now what we want to infer is the latent locations $\mathbf{X}$ and the parameters of the GP namely the parameters of the covariance function θ. This means that we need to solve the following integral,

$$p(\mathbf{Y}) = \int p(\mathbf{Y}, \mathbf{X}, \mathbf{F}, \theta)\mathrm{d}\mathbf{X}\mathrm{d}\mathbf{F}\mathrm{d}\theta = \int p(\mathbf{Y}|\mathbf{F})p(\mathbf{F}|\mathbf{X}, \theta)p(\mathbf{X})p(\theta)\mathrm{d}\mathbf{X}\mathrm{d}\mathbf{F}\mathrm{d}\theta. \tag{79}$$

Sadly it is intractable for us to compute this in closed form. What we can do is integrate out $\mathbf{F}$ but $\mathbf{X}$ and θ are both intractable. In order to proceed we are going to compute the integral over $\mathbf{F}$ first and then we are going to find a maximum-likelihood solution for $\mathbf{X}$. First we can intergrate out the actual function values leading to the marginal likelihood,

$$p(\mathbf{Y}|\mathbf{X},\theta) = \frac{1}{(2\pi)^{\frac{DN}{2}}|K|^{\frac{D}{2}}} e^{-\frac{1}{2}\text{tr}(\mathbf{Y}\mathbf{K}^{-1}\mathbf{Y}^{\text{T}})} \tag{80}$$

Now when we have this distribution we are going to find the latent locations $\mathbf{X}$ that maximises this. This means that we should solve the following optimisation problem,

$$\hat{\mathbf{X}} = \text{argmax}_{\mathbf{X},\theta} p(\mathbf{Y}|\mathbf{X},\theta) = \text{argmin}_{\mathbf{X},\theta} - \log p(\mathbf{Y}|\mathbf{X},\theta) = \mathcal{L}(\mathbf{X}). \tag{81}$$

As no closed form can be found of the following we have to solve this using gradient based methods. Doing so is quite a challenge while we leave this for extra work. The important thing is that you see how similar the two models are.

5.2.1 Implementation

Now sadly the implementation of the GP-LVM is a bit more tedious to do and we need to think quite a bit more about how to compute things both in more clever ways but also how to implement things in a stable way. Therefore the implementation goes beyond the scope of this lab but I've left the derivation here if you are interested in having a look at this.

Computing the gradients is rather tedious affair so we went ahead and did this for you. If you want to compute them on your own you can use the excellent matrix cookbook [10] or if you are really lazy you can use an automatic differentiation package such as JAX that is capable of computing gradients automatically from `numpy` code[18]. Computing gradients of expressions involving matrices are a little bit more involved compared to scalar operators. Effectively what we can think is that we can compute derivatives of a scalar with a matrix and a matrix with a scalar but a matrix with a matrix is not easily defined. You do **not** need to understand or be able to perform the commputations, they are simply here so that you see that it is not magic and so that you can do them if you want. Also, there might be errors in the derivation so if you spot something strange let me know. In order to work out the gradients I've made heavy use of the matrix cookbook [10]. You can see the gradient calculations in Section 5.4. Once we have the gradients we can now use a gradient based optimiser that is implemented in `scipy.optimise`. What we need to provide the optimiser with is a function that returns the objective value for a specific parameter setting and a function that returns the gradients. If we have a function $f(\cdot)$ that takes $\mathbf{x}$ as an input we need to implement a structure such as this,

Code

```
import numpy as np
import scipy as sp
import scipy.optimize as opt

def f(x, *args):
    # return the value of the objective at x
    return val

def dfx(x,*args):
    # return the gradient of the objective at x
    return val

x_star = opt.fmin_cg(f,x0,fprime=dfx, args=args)
```

[18] we will have a look at automatic differentiation in the last optional lab of the unit

The function `f(x, *args)` is the objective function $\mathcal{L}(\mathbf{X})$ and `dfx(x, *args)` is the gradients of the objective. We can use `*args` to pass arguments to the function that we do not optimise. By calling `opt.fmin_cg` the gradients and the objective will be called by the mechanism that is implemented in the optimiser, in order for this to work `x` always needs to be in the form of a vector. This means that we when `f` is called the input is a vector and we need to reshape it to the correct size in order to calculate the objective functions value, same thing is true for the gradient, the returned object needs to be a vector.

5.3 Summary

This lab ends the first part of the unit where we have been building models. Think about what you have done, you have created parametrised probabilities using hierarchical distributions. You have tried to describe the generative process of the data using the tools that you have. Once we have done this we have formulated our beliefs in prior distributions, the random variables on top of the chain. Then we used the tools of probability, the sum and the product rule to try and get the conditional distributions of interest so that we could extract the information that we wanted from data. Up till this lab the computations could be done in closed form but as you could see now the more complicated the model of the data becomes the more challenging the computations have become. In this lab we had to circumvent this problem by throwing principle out the window and perform maximum likelihood estimation. This is in general a dangerous path to walk as we no longer have uncertainty in the estimate, simply because we do not have any beliefs in them. Toward the end of the unit we will see how we can perform approximate computation in order to circumvent the intractability of these models. But till then we just have to feel a little bit bad about having done a couple of point estimates.

References

[1] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian Processes for Data-Efficient Learning in Robotics and Control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, stat.ML(2):408–423, February 2015.

[2] M. Kaiser, C. Otte, T. Runkler, and C. H. Ek. Bayesian alignments of warped multi-output gaussian processes. In *Advances in Neural Information Processing Systems 32, [NIPS Conference, Montreal, Quebec, Canada, December 3 - December 8, 2018]*, 2018.

[3] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[4] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. Bayesian optimization in alphago. *CoRR*, 2018.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] Neil D Lawrence. Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *Journal of Machine Learning Research*, 6:1783–1816, 2005.

[7] Keith Grochow, Steven L Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *SIGGRAPH '04: SIGGRAPH 2004 Papers*, August 2004.

[8] Raquel Urtasun, David J Fleet, and P. Fua. 3D people tracking with Gaussian process dynamical models. *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, 1:238–245, 2006.

[9] Steindór Sæmundsson, Katja Hofmann, and Marc Peter Deisenroth. Meta reinforcement learning with latent variable gaussian processes. *CoRR*, 2018.

[10] K. B. Petersen and M. S. Pedersen. The Matrix Cookbook, November 2012. Version 20121115.

[11] Iain Murray and Zoubin Ghahramani. A note on the evidence and Bayesian Occam's razor. Technical Report GCNU-TR 2005-003, August 2005.

[12] J. Močkus. On bayesian methods for seeking the extremum. In G. I. Marchuk, editor, *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*, pages 400–404, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

5.4 Gradients Calculations

In this section we will derive the gradients that we can use to find the maximum of the marginal likelihood. The derivations for the linear and for the non-linear case are very similar so the derivation will start by first deriving the linear solution and then from there we will derive the ones for the non-linear case. Remember that the only difference between the two objectives is that for one we integrate out the latent space and optimise the parameters of the mapping while in the other we do the opposite and optimise the latent representation directly.

The first thing that we need to do is to write up the objective function, what we want to do is to find,

$$\hat{\mathbf{W}} = \text{argmax}_{\mathbf{W}} p(\mathbf{Y}|\mathbf{W}). \tag{82}$$

Because we are only interested in the location of the optima not the value of the optima we can transform the objective using any monotonic function. As the probability contains an exponential function we therefore take the logarithm of the expression. Furthermore, it is common practice to minimise rather than maximise, we will therefore seek the minima of the negative logarithm of the objective as,

$$\hat{\mathbf{W}} = \text{argmin}_{\mathbf{W}} - \log\left(p(\mathbf{Y}|\mathbf{W})\right) = \mathcal{L}(\mathbf{W}). \tag{83}$$

If we write up the marginal likelihood of all the data $\mathbf{Y}$ we get something like this,

$$p(\mathbf{Y}|\mathbf{W}) = \prod_{i=1}^{N} p(\mathbf{y}_i|\mathbf{W}) = \prod_{i=1}^{N} \frac{1}{(2\pi)^{\frac{D}{2}}|\mathbf{C}(\mathbf{W})|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{y}_i^{\mathrm{T}}(\mathbf{C}(\mathbf{W}))^{-1}\mathbf{y}_i)} \tag{84}$$

$$= \frac{1}{(2\pi)^{\frac{DN}{2}}|\mathbf{C}(\mathbf{W})|^{\frac{D}{2}}} e^{-\frac{1}{2}\sum_{i=1}^{N}(\mathbf{y}_i^{\mathrm{T}}(\mathbf{C}(\mathbf{W}))^{-1}\mathbf{y}_i)}. \tag{85}$$

this leads to an objective function consisting of three terms which looks like this where we have multiplied both therm with two,

$$\mathcal{L}(\mathbf{W}) = \text{constant} + D\log|\mathbf{C}(\mathbf{W})| + \sum_{i}^{N} \mathbf{y}_i^{\mathrm{T}}(\mathbf{C}(\mathbf{W}))^{-1}\mathbf{y}_i \tag{86}$$

Now we need to compute the derivatives of $\mathbf{C}(\mathbf{W})$ with respect to $\mathbf{W}$. To do this lets first rewrite everything on matrix form so that we can remove the sum from the objective function. First note that,

$$\sum_i \mathbf{x}_i\mathbf{x}_i = \text{tr}\left(\begin{bmatrix} \leftarrow & \mathbf{x}_1 & \rightarrow \\ \leftarrow & \mathbf{x}_2 & \rightarrow \\ & \vdots & \\ \leftarrow & \mathbf{x}_N & \rightarrow \end{bmatrix} \begin{bmatrix} \leftarrow & \mathbf{x}_1 & \rightarrow \\ \leftarrow & \mathbf{x}_2 & \rightarrow \\ & \vdots & \\ \leftarrow & \mathbf{x}_N & \rightarrow \end{bmatrix}^{\mathrm{T}}\right), \tag{87}$$

this means that we can rewrite the objective function as,

$$\mathcal{L}(\mathbf{W}) = \text{constant} + D\log|\mathbf{C}(\mathbf{W})| + \text{tr}\left(\mathbf{Y}(\mathbf{C}(\mathbf{W}))^{-1}\mathbf{Y}^{\mathrm{T}}\right). \tag{88}$$

Now we have two terms we need to take derivatives of, both of these include the same matrix $\mathbf{C}$ in one form of the other so it will come in handy to take this derivative first,

$$\frac{\partial \mathbf{C}}{\partial \mathbf{W}_{ij}} = \frac{\partial \mathbf{W}\mathbf{W}^{\mathrm{T}}}{\partial \mathbf{W}_{ij}}. \tag{89}$$

We will now use the excellent Matrix Cookbook [10] to find the rules that we need to figure this one out. If you use the version from 2012 URL we can first use Eq. 37 to rewrite the the product,

$$\partial(\mathbf{X}\mathbf{Y}) = (\partial\mathbf{X})\mathbf{Y} + \mathbf{X}(\partial\mathbf{Y}). \tag{90}$$

We can then combine this with Eq. 32 which states,

$$\frac{\partial \mathbf{X}_{kl}}{\partial \mathbf{X}_{ij}} = \delta_{ik}\delta_{lj}, \tag{91}$$

where δ_{ij} is the kronecker delta function,

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \tag{92}$$

This leads to the following derivative,

$$\frac{\partial \mathbf{W}\mathbf{W}^{\mathrm{T}}}{\partial \mathbf{W}_{ij}} = \mathbf{W}\frac{\partial \mathbf{W}^{\mathrm{T}}}{\partial \mathbf{W}_{ij}} + \frac{\partial \mathbf{W}}{\partial \mathbf{W}_{ij}}\mathbf{W}^{\mathrm{T}} = \mathbf{W}\mathbf{J}_{ij} + \mathbf{J}_{ji}\mathbf{W}^{\mathrm{T}}, \tag{93}$$

where we $\mathbf{J}_{ij}$ is a matrix who has all zero entries except for $(\mathbf{J}_{ij})_{ij} = 1$.

Now lets strart by tackling the first term, the derivative of the log determinant.

$$\frac{\partial}{\partial \mathbf{W}_{ij}}\log|\mathbf{C}| \tag{94}$$

We will start by using Eq. 43 that states that,

$$\partial\log|\mathbf{X}| = \mathrm{tr}\left(\mathbf{X}^{-1}\partial\mathbf{X}\right). \tag{95}$$

We can now rewrite this as,

$$\frac{\partial}{\partial \mathbf{W}_{ij}}\log|\mathbf{C}| = \mathrm{tr}\left(\mathbf{C}^{-1}\frac{\partial \mathbf{C}}{\partial \mathbf{W}_{ij}}\right). \tag{96}$$

So now we have our first term and we can move on to the second. This term is a derivative of a trace we can now use Eq. 36 which states,

$$\partial\left(\mathrm{tr}(\mathbf{X})\right) = \mathrm{tr}\left(\partial\mathbf{X}\right). \tag{97}$$

This means that our second term becomes,

$$\frac{\partial}{\partial \mathbf{W}_{ij}}\mathrm{tr}\left(\mathbf{Y}(\mathbf{C})^{-1}\mathbf{Y}^{\mathrm{T}}\right) = \mathrm{tr}\left(\frac{\partial}{\partial \mathbf{W}_{ij}}\mathbf{Y}(\mathbf{C})^{-1}\mathbf{Y}^{\mathrm{T}}\right). \tag{98}$$

Now we want to break the quadratic form and we will do this by using the chain-rule to do the derivative,

$$\mathrm{tr}\left(\frac{\partial}{\partial \mathbf{W}_{ij}}\mathbf{Y}(\mathbf{C})^{-1}\mathbf{Y}^{\mathrm{T}}\right) = \mathrm{tr}\left(\frac{\partial}{\partial \mathbf{C}^{-1}}(\mathbf{Y}\mathbf{C}^{-1}\mathbf{Y}^{\mathrm{T}})\frac{\partial \mathbf{C}^{-1}}{\partial \mathbf{W}_{ij}}\right) = \mathrm{tr}\left((\mathbf{Y}\mathbf{Y}^{\mathrm{T}})^{\mathrm{T}}\frac{\partial \mathbf{C}^{-1}}{\partial \mathbf{W}_{ij}}\right). \tag{99}$$

Now we have isolated the derivative and we are nearly there. The last rule we will use is to use the derivative of a matrix inverse Eq. 40,

$$\partial\mathbf{X}^{-1} = -\mathbf{X}^{-1}(\partial\mathbf{X})\mathbf{X}^{-1}. \tag{100}$$

This means we can reach the derivative of the last term as,

$$\text{tr}\left((\mathbf{Y}\mathbf{Y}^{\text{T}})^{\text{T}}\frac{\partial \mathbf{C}^{-1}}{\partial \mathbf{W}_{ij}}\right) = \text{tr}\left(\mathbf{Y}^{\text{T}}\mathbf{Y}\left(-\mathbf{C}^{-1}\frac{\partial \mathbf{C}}{\partial \mathbf{W}_{ij}}\mathbf{C}^{-1}\right)\right), \tag{101}$$

where we know the derivative of the inner-most term as we computed it first.

So now we have the full derivative, as we can see the dimensionality makes sense, we take the derivative of our objective function which is a scalar with another scalar and therefore expect a scalar back, as both of our terms are traces of matrices this makes sense. This leads to the full derivative,

$$\frac{\partial}{\partial \mathbf{W}_{ij}}\mathcal{L}(\mathbf{W}) = D\text{tr}\left(\mathbf{C}^{-1}\frac{\partial \mathbf{C}}{\partial \mathbf{W}_{ij}}\right) + \text{tr}\left(\mathbf{Y}^{\text{T}}\mathbf{Y}\left(-\mathbf{C}^{-1}\frac{\partial \mathbf{C}}{\partial \mathbf{W}_{ij}}\mathbf{C}^{-1}\right)\right) \tag{102}$$

$$\frac{\partial \mathbf{C}}{\partial \mathbf{W}_{ij}} = \mathbf{W}\mathbf{J}_{ij} + \mathbf{J}_{ji}\mathbf{W}^{\text{T}} \tag{103}$$

Now we want to change the calculations to the non-linear case. The objective function now stays pretty much the same with only two differences. The first one is the order of the dimensions in the marginal likelihood. In the linear regression case the exponent was,

$$\sum_{i}^{N}(\mathbf{Y}_{i,:}\mathbf{C}(\mathbf{W})^{-1}\mathbf{Y}_{i,:}^{\text{T}}), \tag{104}$$

i.e. the covariance was between the dimensions in the data. Now for the non-parametric case it becomes the other way around, the covariance is between the data-points instead,

$$\sum_{d}^{D}(\mathbf{Y}_{:,d}^{\text{T}}\mathbf{K}^{-1}\mathbf{Y}_{:,d}), \tag{105}$$

So writing the objective now instead becomes,

$$\mathcal{L}(\mathbf{X}) = \text{constant} + N\text{log}|\mathbf{K}| + \text{tr}\left(\mathbf{Y}^{\text{T}}\mathbf{K}^{-1}\mathbf{Y}\right). \tag{106}$$

The second change is that the inner derivative of the covariance matrix is now a function of the latent variables rather than a weight matrix. This means that we can replace $\mathbf{C}(\mathbf{W})$ with $k(\mathbf{X}, \mathbf{X}) = \mathbf{K}$ instead. This means that the only alterations that we need to do is to change the inner derivative, and take the transpose of the data matrix, everything else stays the same. Leading to the following derivative,

$$\frac{\partial}{\partial \mathbf{x}_{id}}\mathcal{L}(\mathbf{X}) = N\text{tr}\left(\mathbf{K}^{-1}\frac{\partial \mathbf{K}}{\partial \mathbf{x}_{id}}\right) + \text{tr}\left(\mathbf{Y}\mathbf{Y}^{\text{T}}\left(-\mathbf{K}^{-1}\frac{\partial \mathbf{K}}{\partial \mathbf{x}_{id}}\mathbf{K}^{-1}\right)\right). \tag{107}$$

Now lets compute the derivative of the kernel matrix.

$$\frac{\partial \mathbf{K}}{\partial x_{id}} = \frac{\partial}{\partial x_{id}}\begin{bmatrix} k_{11} & k_{12} & \dots & k_{1N} \\ k_{21} & k_{22} & \dots & k_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ k_{N1} & k_{N2} & \dots & k_{NN} \end{bmatrix} \tag{108}$$

$$k_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) = \theta_0 e^{-\frac{1}{\theta_1}(\mathbf{x}_i - \mathbf{x}_j)^{\text{T}}(\mathbf{x}_i - \mathbf{x}_j)} \tag{109}$$

This means that we can compute the derivative for the kernel as,

$$\frac{\partial k_{ij}}{\partial x_{id}} = \frac{\partial}{\partial x_{id}}\theta_0 e^{-\frac{1}{\theta_1}(\mathbf{x}_i - \mathbf{x}_j)^{\text{T}}(\mathbf{x}_i - \mathbf{x}_j)} \tag{110}$$

$$= \theta_0 e^{-\frac{1}{\theta_1}(\mathbf{x}_i - \mathbf{x}_j)^{\text{T}}(\mathbf{x}_i - \mathbf{x}_j)}\frac{\partial}{\partial x_{id}}\left(-\frac{1}{\theta_1}(\mathbf{x}_i - \mathbf{x}_j)^{\text{T}}(\mathbf{x}_i - \mathbf{x}_j)\right) \tag{111}$$

$$= k_{ij}\cdot\left(-\frac{1}{\theta_1}\right)\frac{\partial}{\partial x_{id}}(\mathbf{x}_i^{\text{T}}\mathbf{x}_i - \mathbf{x}_i^{\text{T}}\mathbf{x}_j - \mathbf{x}_j^{\text{T}}\mathbf{x}_i + \mathbf{x}_j^{\text{T}}\mathbf{x}_j) \tag{112}$$

$$= -\frac{1}{\theta_1}k_{ij}\cdot 2(x_{id} - x_{jd}) \tag{113}$$

We can also compute the transpose element of the kernel by flipping the indices which means that,

$$\frac{\partial k_{ji}}{\partial x_{id}} = -\frac{1}{\theta_1} k_{ij} \cdot 2(x_{jd} - x_{id}) = -\frac{\partial k_{ij}}{\partial x_{id}}. \tag{114}$$

All the other elements of the kernel matrix does not contain x_{id} which means that they will be zero. Now we can use the regular structure of the derivative to write everything on matrix form as,

$$\frac{\partial k(\mathbf{X}, \mathbf{x}_i)}{\partial x_{id}} = \frac{1}{\theta_1} k(\mathbf{X}, \mathbf{x}_i) \odot \begin{bmatrix} x_{1d} - x_{id} \\ \vdots \\ x_{Nd} - x_{id} \end{bmatrix}, \tag{115}$$

where $\odot$ is the Hadamar product defined as,

$$(\mathbf{A} \odot \mathbf{B})_{ij} = (\mathbf{A})_{ij}(\mathbf{B})_{ij}. \tag{116}$$

Now we have the gradients for one column of the kernel matrix and can use this result to get the results also for a row by using the symmetry in the derivatives. With that we have the gradients that we need.

6 Evidence

Abstract

Over the last few weeks we looked at models, we have gone from simple models to figure out if a coin is biased or not to much more complicated infinite dimensional objects to place distributions over the space of functions. We followed the same procedure in each of these tasks, formulate a likelihood and a prior and try to get to the posterior. This lab we are not going to introduce a new model but instead look at the one part of the probabilistic framework that we have so far ignored, the *evidence* or *marginal likelihood*. Hopefully after doing this lab you will see that this object is not just an annoying object that we try to do our best to avoid working with, no its actually the most important of all the probabilistic objects that we have in our arsenal. So lets get aquainted with the evidence.

In this lab we will look at the role the evidence or the marginal likelihood plays in machine learning. We will follow the excellent paper [11]. The evidence is the probability distribution that is left when we have integrated out everything except for the data. Say that we have observed a set of data $\mathcal{D}$ and we have built up a model of this parameterised by a set of parameter θ the evidence is,

$$p(\mathcal{D}) = \int p(\mathcal{D}|\theta)p(\theta)\mathrm{d}\theta. \tag{117}$$

This means that the evidence is the distribution over the data space that is created if we average *all" of the possible hypothesis that we have relative to how probable we think that they are. So lets try and get an intuition for this. Lets say that we have a modelling scenario where we have a Gaussian model, and we do not know the mean nor the variance, now to make it simple we have an hypothesis space that only includes three different possible settings of the parameters. This would mean the evidence is an average over these three Gaussians as,

$$p(\mathcal{D}) = \sum_{\boldsymbol{\theta}} p(\mathcal{D}|\theta)p(\theta). \tag{118}$$

In the code below I have written up an example of this and the result can be seen in Figure 11.

Code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

x = np.linspace(-6,6,200)
pdf1 = norm.pdf(x,0,1)
pdf2 = norm.pdf(x,1,3)
pdf3 = norm.pdf(x,-2.5,0.5)

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)

ax.plot(x,pdf1,color='r',alpha=0.5)
ax.fill_between(x,pdf1,color='r',alpha=0.3)
ax.plot(x,pdf2,color='g',alpha=0.5)
ax.fill_between(x,pdf2,color='g',alpha=0.3)
ax.plot(x,pdf3,color='b',alpha=0.5)
ax.fill_between(x,pdf3,color='b',alpha=0.3)

pdf4 = 0.3*pdf1 + 0.2*pdf2 + 0.5*pdf3
ax.plot(x, pdf4, color='k', alpha=0.8, linewidth=3.0, linestyle='--')
ax.fill_between(x, pdf4, color='k', alpha=0.5)

# REMOVE THIS
plt.tight_layout()
plt.savefig(path, transparent=True)
return path
```

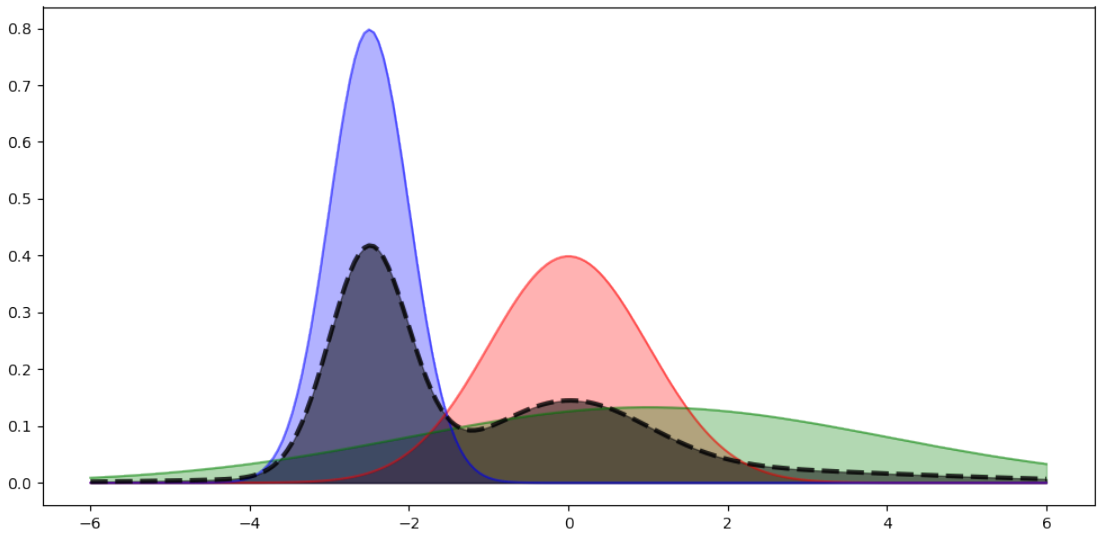

Figure 11: The above figure shows three different parameter settings of a model, *red*, *green* and *blue*. We now marginalise out the parameter according to our belief $p(\theta = \text{red}) = 0.3$, $p(\theta = \text{green}) = 0.2$ and $p(\theta = \text{blue}) = 0.5$ which leads to the evidence in black.

So we should think of the evidence how a model and our beliefs places probability mass over the space where we can later observe data. So now if we would observe some data $\mathbf{Y}$ we can evaluate this under the evidence and say, *what is the evidence that this model is the "right" one?*. Now this becomes very interesting when we have several models. So lets pick another model where instead of Gaussian distribution we have a model using a Laplace distribution giving rise to the evidence plot in Figure 12. So clearly this model and our beliefs in this model places distribution slightly differently across the space.

Code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import laplace

x = np.linspace(-6,6,200)
pdf1 = laplace.pdf(x,0,1)
pdf2 = laplace.pdf(x,-1,1)
pdf3 = laplace.pdf(x,-2.5,0.5)

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)

ax.plot(x,pdf1,color='r',alpha=0.5)
ax.fill_between(x,pdf1,color='r',alpha=0.3)
ax.plot(x,pdf2,color='g',alpha=0.5)
ax.fill_between(x,pdf2,color='g',alpha=0.3)
ax.plot(x,pdf3,color='b',alpha=0.5)
ax.fill_between(x,pdf3,color='b',alpha=0.3)

pdf4 = 0.3*pdf1 + 0.2*pdf2 + 0.5*pdf3
ax.plot(x, pdf4, color='k', alpha=0.8, linewidth=3.0, linestyle='--')
ax.fill_between(x, pdf4, color='k', alpha=0.5)

# REMOVE THIS
plt.tight_layout()
plt.savefig(path, transparent=True)
return path
```

So how is this useful, well, so far we have not seen any data, lets say that we now are observing some data $\mathbf{Y}$ which is all centered around -1 as. If we now evaluate the evidence for this data under the two different models we will see that the data is more probably under the model using the Laplace distribution compared to the Gaussian distribution. Simply because the latter model places more of its probability mass just there. This gives us the following relationship,

$$p_{\text{Gaussian}}(\mathcal{D} = \mathbf{Y}) < p_{\text{Laplace}}(\mathcal{D} = \mathbf{Y}), \tag{119}$$

therefore if we would have to choose a model to use to represent this data we would say *There is more or higher evidence for the Laplace model compared to the Gaussian model, we would therefore choose the Laplace model.* This is a really powerful statement as it allows us to test different hypothesis about *model* not just *parameters* of models using this evidence.

This line of reasoning have lead to one of the more famous plots in machine learning, which I like to call the MacKay plot after David Mackay. David was a very influential person in the machine learning community and I would argue that he has a lot to do with the prominent position the UK has played in the development of this field. Slightly outside this topic he also wrote an excellent book on global warming called Without Hot Air which is a fantastic read about the challenges of energy. But back on track, David put this Figure 13 of the evidence in his PhD thesis.

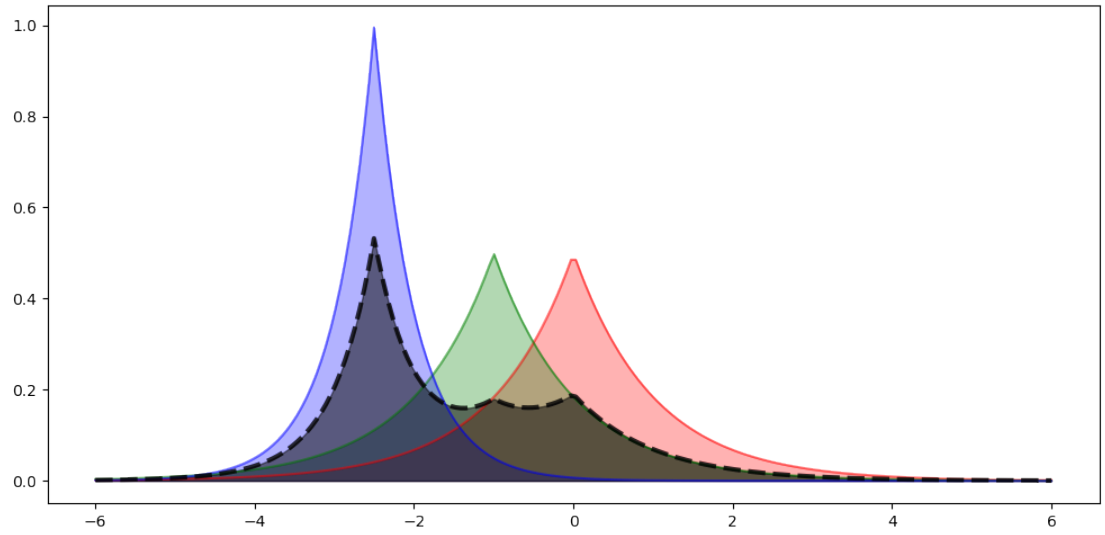

Figure 12: The evidence compute for a slightly different model where we have Laplace distributions that we do not know the parameters of.

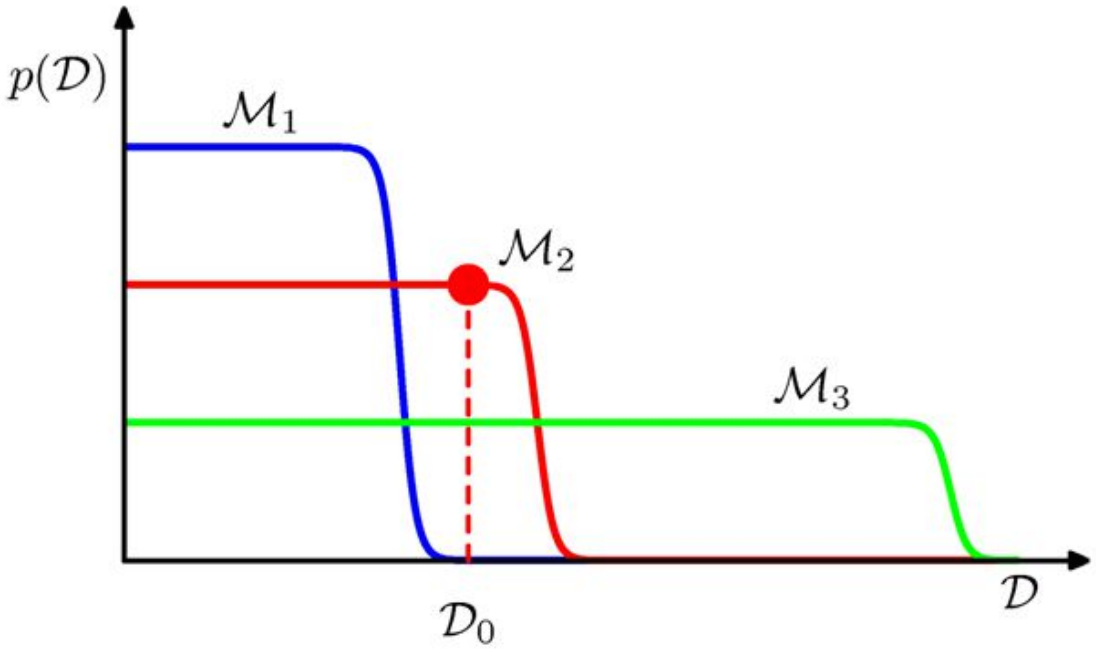

Figure 13: This is the famous Mackay plot, the idea is that if you compute the evidence under three different models, the green, red and blue. The data that you actually observe is $\mathcal{D}_0$ which is evaluated under the three different distributions.

The idea of the plot is that if you sort the data-space such that the simpler the data is the further left it is. Now this would mean that the "simpler" the model is then it will place probability mass further left. Now if we are now to choose model between the three above we can use the argument of Occams Razor which simply states that you should *pick the simplest model possible that explains your data*. Now as probability distributions have to sum to 1 the model that has the highest probability when evaluated at the data will thefore have to have placed less probability mass elsewhere and therefore contain less explanations. Therefore picking the model with the maximum evidence would therefore be choosing the model according to Occams Razor. In Figure 13 model $\mathcal{M}_1$ is too simple and $\mathcal{M}_3$ is too complicated and explains too many things while $\mathcal{M}_2$ is just right and this is exactly what the evidence would encode as $p_{\mathcal{M}_2}(\mathcal{D} = \mathcal{D}_0)$ would be higher than both $p_{\mathcal{M}_1}(\mathcal{D} = \mathcal{D}_0)$ and $p_{\mathcal{M}_3}(\mathcal{D} = \mathcal{D}_0)$. Think about this, does it make sense?

Hopefully you think that the argument above makes sense and see that this can be quite a powerful technique. I think its a really strong argumentations and a nice codification of Occam's Razor. Say that you meet someone that believes that the earth is flat, most likely they will present you with evidence that they think supports their theory and most likely[19] the examples that they will give and the explanations that they give will fit. But still, you don't believe them, you think that their arguments are so convoluted, so complicated so really you just go about your way believing the earth is a sphere and then move on. So really, you decided that their model didn't fit because the model that choose was too *complicated* so you choose another model that was *simpler* but importantly both provided an equivivalent explanation of the data. This means you have acted according to Occams Razor. However, if this is the case, why does anyone believe that the earth is flat[20] simply because the notion of simple is not universal. We all have a different idea of what is simple, and because of this the implementation of Occam's Razor is subjective. You first have to define what simple is in order to use the concept, again, *there is no free-lunch*. What we will now do for the rest of the worksheet is to implement a scenario where we can actually test these concepts and make our understanding of this a bit more clear. We will do so by repeating the experiments of [11]. This paper is a really good read, what I especially like about it is that it doesn't provide an answer, it just raises more questions.

6.1 A Note on the Evidence and Bayesian Occam's Razor

The reason that the unit initially have focused on conjugate models is because we wanted to avoid computing Baye's Rule to reach the posterior distribution. Conjugacy allowed us to avoid computing the denominator and simply multiply prior and likelhood, then identify the terms to be able to normalise the posterior. For most models this is not possible and we are required to actually perform the full computation and solve an often intractable integral to reach the posterior distribution. Now the object that we want to look at is the evidence so avoiding computation of it is rather pointless. Because of this we are going to choose a discrete data domain whos cardinality is so small so that we can actually evaluate the evidence for all possible data-sets. We will start off by first creating the data-set then we will move on and create a set of models that we can compute the evidence under.

6.1.1 Data

Consider a very simple data domain $\mathcal{D} = \{y^i\}_{i=1}^9$ where $y^i \in \{-1, 1\}$. This data is structured according to a grid whos locations can be parametrised by $\mathcal{X} = \{\mathbf{x}^i\}_{i=1}^9$ where $\mathbf{x}^i = (\{-1, 0, +1\}, \{-1, 0, +1\})$. This means that our data domain $\mathcal{D}$ contains $2^9 = 512$ different elements which is small enough for us to reason about but still complicated enough that it requires a sensible model.

We will now generate all possible data-sets so that we can evaluate the evidence. The code below will generate a list that you can iterate through with all the possible data points.

[19] otherwise they will be a bit stupid

[20] or that Bristol Rovers is a good football team

Code

```
import itertools as it

def generate_data(N=3):
    D2 = np.array(list(it.product([-1,1],repeat=N*N)))
    D = [];
    for i in range(0,len(D2)):
        d = D2[i]
        D.append(d.reshape([N,N]))
        x = [];
    for i in range(-(N-2),N-1):
        for j in range(-(N-2),N-1):
            x.append(np.array([float(i),float(j)]))
    return (D,x)
```

6.1.2 Models

Given the data defined above we wish to create a model, i.e. something that will explain the statistical variations that are possible in $\mathcal{D}$. The simplest model that (I) can think of is something that simply takes all its probability mass and places it uniformly over the whole data space,

$$p(\mathcal{D}|M_0, \boldsymbol{\theta}_0) = \frac{1}{512}. \tag{120}$$

The first model Eq. 120 does not take any parameters at all which means it has no flexibility and uses no information about $\mathcal{D}$ except for its cardinality. We can use what we know about the data in order to specify something slightly more representative. If we assume that each y^i are independent we can factorise the model into 9 separate models,

$$p(\mathcal{D}|M_1, \boldsymbol{\theta}_1) = \prod_{n=1}^{9} p(y^i|M_1, \boldsymbol{\theta}_1), \tag{121}$$

where θ_i^j means the j:th element of the parameter vector for the t:th model. Each model can be expressed using an exponential function which relates the vaule y^i to its location $\mathbf{x}^i$,

$$p(\mathcal{D}|M_1, \boldsymbol{\theta}_1) = \prod_{n=1}^{9} \frac{1}{1 + e^{-y^n \theta_1^1 x_1^n}}, \tag{122}$$

Question 1

Explain how the each separate model works? In what way is this model more or less flexible compared to M_0? How does this model spread its probability mass over $\mathcal{D}$?}

We can continue to add more parameters and create further models,

$$p(\mathcal{D}|M_2, \boldsymbol{\theta}_2) = \prod_{n=1}^{9} \frac{1}{1 + e^{-y^n (\theta_2^1 x_1^n + \theta_2^2 x_2^n)}} \tag{123}$$

$$p(\mathcal{D}|M_3, \boldsymbol{\theta}_3) = \prod_{n=1}^{9} \frac{1}{1 + e^{-y^n (\theta_3^1 x_1^n + \theta_3^2 x_2^n + \theta_3^3)}}, \tag{124}$$

Now we can implement the three different models such that we can return the probability for a specific data-point.

Code

```
def model0(theta,x,y):
    return 1.0/(pow(2.,len(x)))

def model1(theta,x,y):
    model = 1.0
    for i in range(len(x)):
        model *= 1.0/(1+exp(-y[i]*theta[0]*x[i][0]))
    return model

def model2(theta,x,y):
    model = 1.0
    for i in range(len(x)):
        model *= 1.0/(1+exp(-y[i]*(theta[0]*x[i][0]+theta[1]*x[i][1])))
    return model

def model3(theta,x,y):
    model = 1.0
    for i in range(len(x)):
        model *= 1.0/(1+exp(-y[i]*(theta[0]*x[i][0]+theta[1]*x[i][1]+theta[2])))
    return model
```

Question 2

How have the choices we made above restricted the distribution of the model? What data sets are each model suited to model? What does this actually imply in terms of uncertainty? In what way are the different models more flexible and in what way are they more restrictive?

Now we have both the models and the data and its time to move on to the actual computation of the evidence.

6.1.3 Evidence

The evidence of a model M_i is the distribution $p(\mathcal{D}|M_i)$. This distribution tells us how and where the model spreads its probability mass. Occam's razor can be interpreted in terms of the evidence such as we should choose a model which places most of its mass where we will see data and as little as possible elsewhere. In the previous section we have defined a small simple data domain $\mathcal{D}$ and we will now evaluate where the different models defined above places their probability mass.

In order to "reach" the evidence of a model we need to first remove the dependency of the variable $\boldsymbol{\theta}$. This can be done by marginalising out the parameters from the model,

$$p(\mathcal{D}|M_i) = \int_{\forall \boldsymbol{\theta}} p(\mathcal{D}|M_i, \boldsymbol{\theta}) p(\boldsymbol{\theta}) \mathrm{d}\boldsymbol{\theta}. \tag{125}$$

The marginalisation above requires one more object that we haven't seen before $p(\boldsymbol{\theta}|M_i)$. This is the prior over the parameters of the model. Being Bayesian implies that you need to take uncertainty into account in all steps of your calculations this is true for the data but also true for the parameters. As we do not really know much at all about the parameters we would like to be very uncertain and allow for a large range of possible values of $\boldsymbol{\theta}$. One prior would be to choose a simple Gaussian with zero mean and a very

large variance,

$$
\begin{aligned}
p(\boldsymbol{\theta}|M_i) &= \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \\
\boldsymbol{\mu} &= \mathbf{0} \\
\boldsymbol{\Sigma} &= \sigma^2 \boldsymbol{I} \\
\sigma^2 &= 10^3
\end{aligned}
\tag{126}
$$

Now when we have defined the prior $p(\boldsymbol{\theta})$ we just need to perform the marginalisation in Eq. 125 to be able to evaluate the evidence. However, this integration is rather tricky to do analytically which means that we will here use an approximate integral using a naive Monte Carlo approach,

$$
p(\mathcal{D}|M_i) \approx \frac{1}{S} \sum_{s=1}^{S} p(\mathcal{D}|M_i, \boldsymbol{\theta}^s), \tag{127}
$$

$$
\boldsymbol{\theta}^s \sim p(\boldsymbol{\theta}|M_i) \tag{128}
$$

where s indexes the samples from the prior of the parameters. Do not worry too much about this proceedure right now, we will later in the unit talk about sampling in more detail and we will do a lab specifically on this topic. Right now, just see this as a black-box proceedure that we are doing.

The code that you need in order to generate the evidence for the different models is

Code

```
# generate the parameter vector for the samples
def generate_parameters(N,d,mu,sigma):
    return sigma*np.random.randn(N,d)+mu

# generate the evidence
def compute_evidence(y,x,theta,model):
    evidence = 0.0
    for i in range(len(theta)):
        evidence += model(theta[i],x,y)
    return evidence/len(theta)
```

Now we have our models, we have our data and we have an approach to reach the evidence for each model. It is now time to run some experiments and see where this leads to. If you have set things up correctly you should be able to run something similar to,

Code

```
N = 3;
nr_samples = pow(10,2)
sigma = pow(10,1.5)
mu = 0
[D, x] = generate_data(N)
[theta, theta_prior] = generate_parameters(nr_samples,3,mu,sigma)

evidence = np.zeros([4,len(D)])

for i in range(len(D)):
    evidence[0,i] = compute_evidence(D[i].ravel(),x,theta,model0)
    evidence[1,i] = compute_evidence(D[i].ravel(),x,theta,model1)
    evidence[2,i] = compute_evidence(D[i].ravel(),x,theta,model2)
    evidence[3,i] = compute_evidence(D[i].ravel(),x,theta,model3)
```

What we now have is the evidence compute under each of the different models. We can now look at how the distribute probability mass over $\mathcal{D}$. The big question is how to sort the data-set, what order should the data have? One thing that you can try is to sort the data according to one model and plot all model using that one. You can get the indices that sorts an array by using `np.argsort()`. So try something like this,

Code

```
index = np.argsort(evidence[1,:])
ax.plot(evidence[0,index], 'r')
ax.plot(evidence[1,index], 'g')
ax.plot(evidence[2,index], 'b')
ax.plot(evidence[3,index], 'k')
```

When you have got everything running, go back to the arguments we did to motivate the evidence. Does it make sense, do you feel that this is supported in the experiments?

6.2 Summary

Hopefully you have reached the end of this lab and quite possibly you are a bit confused at this point. What am I actually supposed to have learnt from this? Lets go and think about it from the start of the machine learning unit, we argued that it is impossible to make any type of learning without making assumptions or having beliefs. Now we have just taken this to its extreme and made an argument that this is also true for Occam's Razor, this is truly a subjective argument as it relies on the concept of simple. The second argument is looking at the plots seeing how the different models distribute their probablity mass, some models represent certain types of data well and seeing that when building models it is alway a choice, if you are good at something you always pay the price for being bad at something else, the important thing is therefore to choose the model which is good at the relevant thing, the thing that you are interested in.

References

[1] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian Processes for Data-Efficient Learning in Robotics and Control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, stat.ML(2):408–423, February 2015.

[2] M. Kaiser, C. Otte, T. Runkler, and C. H. Ek. Bayesian alignments of warped multi-output gaussian processes. In *Advances in Neural Information Processing Systems 32, [NIPS Conference, Montreal, Quebec, Canada, December 3 - December 8, 2018]*, 2018.

[3] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[4] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. Bayesian optimization in alphago. *CoRR*, 2018.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] Neil D Lawrence. Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *Journal of Machine Learning Research*, 6:1783–1816, 2005.

[7] Keith Grochow, Steven L Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *SIGGRAPH '04: SIGGRAPH 2004 Papers*, August 2004.

[8] Raquel Urtasun, David J Fleet, and P. Fua. 3D people tracking with Gaussian process dynamical models. *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, 1:238–245, 2006.

[9] Steindór Sæmundsson, Katja Hofmann, and Marc Peter Deisenroth. Meta reinforcement learning with latent variable gaussian processes. *CoRR*, 2018.

[10] K. B. Petersen and M. S. Pedersen. The Matrix Cookbook, November 2012. Version 20121115.

[11] Iain Murray and Zoubin Ghahramani. A note on the evidence and Bayesian Occam's razor. Technical Report GCNU-TR 2005-003, August 2005.

[12] J. Močkus. On bayesian methods for seeking the extremum. In G. I. Marchuk, editor, *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*, pages 400–404, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

7 Bayesian Optimisation

Abstract

This week we are going to make use of our previous knowledge in modelling and use it for black-box optimisation. The idea here is that we have a function that we want to optimise, i.e. find the minima[21], the tricky thing is that we do not know the form of the function but we can evaluate it.

Lets assume that we have a function $f(\mathbf{x})$ that is *explicitly* unknown that we want to find the minima of. We will further assume that it is possible to evaluate the function but that each evaluation is expensive. This means that the problem that we have on our hands is to *search* the input domain for the extreme point but do so in a manner that we minimise the number of evaluation that we make of the function. One approach to address this type of problem is to use a technique called *Bayesian Optimisation* and that is the focus of this lab.

Searching for the extremum of explicitly unknown functions is problem that appears in many applications. The first use of Bayesian methods for approaching this is usually attributed to [12] a Lithuanian mathematician. This important work was slightly overlooked at the time but recently it has gotten the attention that it deserves. The reason for this is that with increasingly complicated models with enormous cost for training being able to efficiently utilise the data have become very important. The use of Bayesian optimisation for learning how to set parameters in complicated unstructured models[22] is often attributed to [3] who really put these types of techniques at the forefront of modern machine learning. Even though the are used everywhere this is often not reported particularly well, as an example it took several years for the authors of AlphaGo to properly publish and discuss the importance of Bayesian optimisation for their task [4].

The main part of a Bayesian optimisation system is a loop where we in an iterative manner decided on new locations to evaluate the objective function. The two components of the loop are a surrogate model of the function which describes how we believe the function looks in every part of the domain and a acquisition function which decides based on our current belief of what the function is where to sample next. Importantly this makes it key to have uncertainty in our system as we need to have a belief about what the function value is everywhere. As we have already looked at Gaussian processes as a rich class of function priors we will use them for our surrogate model. Lets begin by making the problem more concrete.

Lets assume that we want to find the minima of a function $f(x)$,

$$x_* = \text{argmin}_x f(x) \tag{129}$$

we will at each time have observed a set of values of the function at specific function locations, we will refer to this as the data $\mathcal{D} = \{\mathbf{x}, \mathbf{f}\}$. Furthermore at any point we have a current best estimate, we will refer to this location in the data space as x_* and its function value as $f(x_*)$. We will use a surrogate model to describe our beliefs about the function f. We will use a Gaussian process to do so which means that we have access to a distribution $p(f|x, \mathcal{D})$ which is the predictive posterior of the Gaussian process.

7.1 Surrogate Model

We will use a Gaussian process as a surrogate model for the function. In Lab 4 we looked at how to work with Gaussian processes. If you feel that you need a bit of a recap of this go back to that lab and make sure that you understand Eq. 10-12 and how Figure 2 was generated. What we need to know from our GP is given a set of data $\mathcal{D}$ what is our belief of what the function is at every other location in the input domain. This is the object that we call the *predictive posterior* of the Gaussian process,

$$p(\mathbf{f}_*|\mathcal{D}, \mathbf{x}_*, \theta) = \mathcal{N}(\mu_{\mathbf{x}_*|\mathbf{x}}, K_{\mathbf{x}_*|\mathbf{x}}) \tag{130}$$

$$\mu_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x})k(\mathbf{x}, \mathbf{x})^{-1}\mathbf{f} \tag{131}$$

$$K_{\mathbf{x}_*|\mathbf{x}} = k(\mathbf{x}_*, \mathbf{x}_*) - k(\mathbf{x}_*, \mathbf{x})k(\mathbf{x}, \mathbf{x})^{-1}k(\mathbf{x}, \mathbf{x}_*). \tag{132}$$

[21] we will usually refer to it as the minima, when we want to maximise, we just take the negative

[22] read neural networks

You will need to implement a function that can return the mean and the variance at a set of locations `x_star` of a Gaussian process parametrised using `theta`.

Code

```
def surrogate_belief(x,f,x_star,theta):

    return mu_star, varSigma_star
```

Now when we have our surrogate model set up it is time to move on to the second component, the acquisition function.

7.2 Aquisition Function

The idea of the aqusition function is that it encodes the strategy of how we should utilise the knowledge that we currently have in order to decide on where to query the function. The design of this function is where we balance the two important factors, *exploration* where we learn about new things, and *exploitation* where we utilise what we currently know. There are many different acquisition functions to use and we will here only look at one of them but in principle they all describe a *utility-value* across the whole input domain of how much we will "gain" by querying the function in this specific place.

7.2.1 Expected Improvement

The most commonly used acquisition function is *Expected Improvement*. The idea underlying expected improvement is that the utility of a location in the input domain is relative to how much lower we expect the function value at this point to be. This means that the utility function $u(x)$ can be defined as follows,

$$u(x) = \max(0, f(x_*) - f(x)) \tag{133}$$

This means that we have a reward for every location in the space where the function $f(x)$ is smaller than the current best estimate $f(x_*)$. Now as we do not know $f(x)$ we want to use our knowledge from the surrogate model f. This we can do by taking the expectation of the utility function over our belief in the function as,

$$\alpha(x) = \mathbb{E}\left[u(x)|x, \mathcal{D}\right] = \int_{-\infty}^{f(x_*)} (f(x_*) - f(x))\mathcal{N}(f|\mu(x), k(x,x))\mathrm{d}f. \tag{134}$$

One of the nice things about Expected improvement is that we can evaluate the expectation in closed form resulting in the following acquisition function,

$$\alpha(x) = (f(x_*) - \mu(x))\Psi(f(x_*)|\mu(x), k(x,x)) + k(x,x)\mathcal{N}f(x_*)|\mu(x), k(x,x)) \tag{135}$$

$$\Psi(f(x_*)|\mu(x), k(x,x)) = \int_{-\infty}^{f(x_*)} \mathcal{N}(f|\mu(x), k(x,x))\mathrm{d}f. \tag{136}$$

The function Ψ is the *cumulative density function* or *cdf* of the Gaussian which has the following form,

$$\frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right), \tag{137}$$

where $\mathrm{erf}(\cdot)$ is the *error-function*[23]. Now we want to choose points in the input domain that will maximise the acquisition function. Looking at the function that we have derived we can see that it includes two terms, the first term can be increased by picking an x value such that the difference between $f(x_*) - \mu(x)$ is large. In effect this is *exploiting* the knowledge that we currently have about the function. The second term can be increased by finding a location in the input domain such that $k(x,x)$ is large, i.e. the variance at this location is high. In effect this is *exploration* as we are looking for locations where we are uncertain of what

[23] `https://en.wikipedia.org/wiki/Error_function`

the value is. As you can see these two terms formulates a specific balancing between the two key aspects of search, *exploration* and *exploitation*.

Now we need to write an implementation of the acquisition function we are going to need something looking like this,

Code

```
from scipy.stats import norm
def expected_improvement(f_*, mu, varSigma, x):

    # norm.cdf(x, loc, scale) evaluates the cdf of the normal distribution

    return alpha
```

where `mu` and `varSigma` is the mean and the variance of the predictive posterior of the surrogate model at locations `x` which is the set of candidates for where to pick the next function evaluation from.

We now have all the parts that we need in order to implement our Bayesian optimisation loop, the surrogate model using a Gaussian process and the acquisition function using expected improvement.

7.3 Experiments

We will now write up the Bayesian optimisation loop that we will iterate through. The first thing we need is a function to evaluate. As we want to be able to play around with the function a bit we will add a set of possible arguments. The functions was taken from this excellent blog post on Bayesian optimisation.

Code

```
def f(x, beta=0, alpha1=1.0, alpha2=1.0):
    return np.sin(3.0*x) - alpha1*x + alpha2*x**2 + beta*np.random.randn(x.shape[0])
```

The next thing that we will do is to decide on a finite set of possible evaluations of the function. The function that we are using is a function in $\mathbb{R}$ what we will do is to divide up this space into a finite set of locations and then our aim is to find at which one of these points we have the minimal value of the function. If we call this set $\mathbf{X}$ we will now start our loop by taking a random set of starting points, compute the predictive posterior over the remaining points, compute the acquisition for all the points not included in the model, pick the location with the highest acquisition and include this into the modelling set. A hand-wavy structure should look something like this.

Algorithm 1 Bayesian Optimisation

procedure BO($f(x), \alpha(x), \mathbf{X}$)
  $x_{\text{start}} \subset \mathbf{X}$ ▷ Pick a random set of start-points
  $x \leftarrow x_{\text{start}}$
  $f' \leftarrow \text{argmin}_{x' \in x} f(x')$
  while iter **do** ▷ Loop until we have reached max number of iterations
    Evaluate $\mu_{\mathbf{X}|\mathbf{x}}$ and $K_{\mathbf{X}|\mathbf{x}}$ ▷ Predictive Posterior of Surrogate
    Evaluate $\alpha(\mathbf{X})$ ▷ Aquisition Function
    $x' = \text{argmax}_{\hat{x} \in \mathbf{X}} \alpha(\hat{x})$ ▷ Pick "best" candidate to evaluation set
    $x = x \cup x'$ ▷ Add element x' to the set
    if $f(x') < f'$ **then** ▷ Update current minima
      $f' = f(x')$
  return f'

One useful way to code this things is to keep two sets of points, you first start with an array with all locations that you can evaluate the function at, then you pick a random subset from this and move them to

another set. Then for each evaluation you keep removing points from the initial set. This can easily be done with numpy arrays like this,

Code

```
# remove points from an array
x_2 = np.arange(10)
index = np.random.permutation(10)
x_1 = x_2[index[0:3]]
x_2 = np.delete(x_2, index[0:3])

# remove largest element
ind = np.argmax(x_2)
x_1 = np.append(x_1, x_2[ind])
x_2 = np.delete(x_2, ind)
```

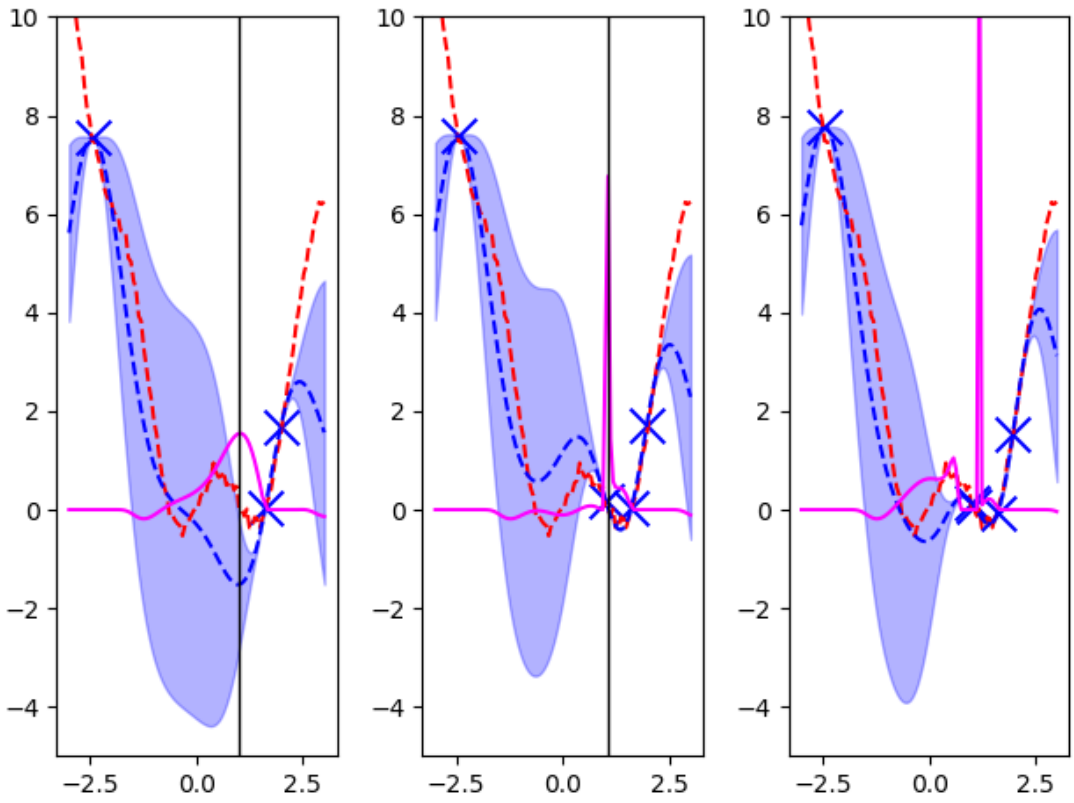

Figure 14: The image below shows the aqcuisition function in magenta, the true function in red and the surrogate models belief in blue. The left-most pane is the first iteration, at each iteration we add in the location of the highest acuisition and update the surrogate model. The image on the far right shows the third iteration.

When you have gotten the loop implemented you can try and see how good result you generally get in a fixed number of iterations. Then you compare this result with taking the same number of locations uniformly at random from the index set and evaluating them. If you compare the runs how often do you get a better value with the Bayesian optimisation approach compared to the random search? Now we can alter

this question slightly, given that you have a current best estimate using BO, how many random samples do you need in order to get an equally good result?

7.4 Summary

Hopefully you have seen that having the concept of uncertainty can be really useful in order to direct a sequential search strategy as in Bayesian optimisation. The example that we evaluated here was in one dimension for us to be able to visualise the results. In general the improvements that you achieve will only improve with increasing dimension[24].

References

[1] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian Processes for Data-Efficient Learning in Robotics and Control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, stat.ML(2):408–423, February 2015.

[2] M. Kaiser, C. Otte, T. Runkler, and C. H. Ek. Bayesian alignments of warped multi-output gaussian processes. In *Advances in Neural Information Processing Systems 32, [NIPS Conference, Montreal, Quebec, Canada, December 3 - December 8, 2018]*, 2018.

[3] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[4] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. Bayesian optimization in alphago. *CoRR*, 2018.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] Neil D Lawrence. Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *Journal of Machine Learning Research*, 6:1783–1816, 2005.

[7] Keith Grochow, Steven L Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *SIGGRAPH '04: SIGGRAPH 2004 Papers*, August 2004.

[8] Raquel Urtasun, David J Fleet, and P. Fua. 3D people tracking with Gaussian process dynamical models. *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, 1:238–245, 2006.

[9] Steindór Sæmundsson, Katja Hofmann, and Marc Peter Deisenroth. Meta reinforcement learning with latent variable gaussian processes. *CoRR*, 2018.

[10] K. B. Petersen and M. S. Pedersen. The Matrix Cookbook, November 2012. Version 20121115.

[11] Iain Murray and Zoubin Ghahramani. A note on the evidence and Bayesian Occam's razor. Technical Report GCNU-TR 2005-003, August 2005.

[12] J. Močkus. On bayesian methods for seeking the extremum. In G. I. Marchuk, editor, *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*, pages 400–404, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

[24] after a while they will start to do the opposite but thats a different question

8 Sampling

Abstract

In the first set of labs we looked at how we can create models that allows us to parametrise and factorise a distribution over the observed data domain. We saw that we can make decisions from the posterior distribution of the model which we reached by combining the model with observed data. For many different models it is not feasible to compute the posterior in closed form most commonly because the marginal likelihood or the evidence is not analytically or computationally tractable[25]. So how do we proceed? Well one possibility is to look for a point estimate rather than the full distribution and proceed with a Maximum Likelihood or a Maximum-a-Posteriori estimate. However, this should really be our last resort as these methods will at best tell us what it believes the "best" approach is but will not at all quantify what "best" means. Further our assumptions in this case does not reach the data which means they are at best regularisers. This means that for such an inference scheme we cannot make a choice if we should trust the model or not, nor can we make a choice on how well the model actually describes the data. The more sensible approach is to try and approximate the intractable integrals and here there are two main approaches, either stochastic or deterministic methods. They both have their benefits and negatives. A stochastic approach is simple to formulate but importantly it is hard to assess how well we are doing and in many ways it is considered one of the "black arts" of machine learning. Deterministic approaches are usually very efficient but they will never be exact. In the final two labs of this unit we will look at both of these approaches applied to the same problem. This week we will look at the stochastic approximations and next week we will look at the deterministic methods.

The task of inference in a machine learning model is the task of combining our assumptions with the observed data. In specific we have a set of observed data $\mathbf{Y}$ which have been parametrised by a variable $\boldsymbol{\theta}$ the task requires us to use Bayes rule to reach the posterior $p(\boldsymbol{\theta}|\mathbf{Y})$,

$$p(\boldsymbol{\theta}|\mathbf{Y}) = \frac{p(\mathbf{Y}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathbf{Y})}.$$

The challenging part of the relationship above is the marginal likelihood or the evidence, which is the probability of the observed data when *all* assumptions have been propagated through and integrated out,

$$p(\mathbf{Y}) = \int p(\mathbf{Y}, \boldsymbol{\theta}) \mathrm{d}\boldsymbol{\theta}.$$

In the first labs we looked at situations where we can avoid calculating the marginal likelihood by exploiting conjugacy, however, for certain cases it is simply not possible as this integral is intractable, either computationally but quite often it is analytically intractable. In order to proceed we have to make sacrifices and approximate this integral. But in order for the lab to get underway we need to have a model to play around with that will exemplify the different approaches.

In this part we are going to look at the rather useful task of image restoration, in specific we are going to work with binary images which have been corrupted by noise and we are supposed to clean them up. The task is exactly the same if you want to perform image segmentation rather than denoising.

8.1 The Model

Images are one of the most interesting and easily available sources of data, images contain a lot of information and they can be acquired in an unitrusive manner with very cheap sensors. Our task here is to build a model of images, in specific of binary or black-and-white images. Images are normally represented as a grid of pixels y_i however the images we observe are noisy and rather will be a realisation of an underlying latent pixel representation x_i. Now to make our computations a bit easier, lets say that white is encoded by $x_i = 1$ and black with $x_i = -1$ and that the grey-scale values that we observed $y_i \in (0, 1)$. We will write our likelihood on this form,

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z_1} \prod_{i=1}^{N} e^{L_i(x_i)}, \tag{138}$$

[25] think about how many elements you had in the summation for such a simple problem as the one we looked at in the Evidence lab.

where $L_i(x_i)$ is a function which generates a large value if x_i is likely to have generated y_i and Z_1 is a factor that ensures that $p(\mathbf{y}|\mathbf{x})$ is a distribution. We have further assumed that the pixels in the image are conditionally independent given the latent variables $\mathbf{x}$.

The next part is to think what a sensible prior would be, what do we actually know about images? One important aspect of images that makes them, well images is that there is a significant correlation between neighbouring pixels. What do we know about this relationship? Well, lets say that we see one white pixel, what do we believe the most likely colour of the pixel to the right to be? If I had to guess I would probably say white as I think that images have more contious segments of one colour compared to switches between colours. So this is now prior information, an assumption that we want to quantify in terms of a probability. We can write down this as follows,

$$p(\mathbf{x}) = \frac{1}{Z_0} e^{E_0(\mathbf{x})}, \tag{139}$$

where again $E_0(\mathbf{x})$ is a function that is large the configuration of $\mathbf{x}$ is something that we believe is likely and small otherwise and Z_0 a normalising term to ensure that $p(\mathbf{x})$ is a distribution. If we follow our previous reasoning and say that a pixel depends on its neighbouring pixels only we can write $E_0(\mathbf{x})$ as function of the following form,

$$E_0(\mathbf{x}) = \sum_{i=1}^{N} \sum_{j \in \mathcal{N}(i)} w_{ij} x_i x_j, \tag{140}$$

where $\mathcal{N}(i)$ specifies the set of nodes that are neighbours to node i. Remember that $x_i \in [-1, 1]$ this means that $x_i x_j$ will be 1 if the nodes have the same label and -1 if the nodes have different labels. The scalars w_{ij} are our parameters that we can control the strength of our prior with, where a large value w_{ij} implies that node $x_i x_j$ are nodes that we really believe should have the same label. Now we have our final model and can describe the joint distribution,

$$p(\mathbf{x}, \mathbf{y}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x}) = \frac{1}{Z_1} \prod_{i=1}^{N} e^{L_i(x_i)} \frac{1}{Z_0} e^{\sum_{j \in \mathcal{N}(i)} w_{ij} x_i x_j}. \tag{141}$$

We can also write up the graphical model for the model which is shown in Figure 15. The model that we just have described is referred to as a Markov Random Field with a Ising prior. This model was initially described in physics[26] to study nearby magnets where the latent variable was their "direction". However, it turns out that they are very good models for images in many tasks.

8.2 Inference

The task we will study in this paper is to given a noisy observation $\mathbf{y}$ recover the latent variables $\mathbf{x}$ that have generated the observations. This means that we want to reach the posterior distribution $p(\mathbf{x}|\mathbf{y})$ to do so we have to compute Baye's rule,

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})}.$$

The denominator could be computed as follows,

$$p(\mathbf{y}) = \sum_{\mathbf{x}} p(\mathbf{y}|\mathbf{x})p(\mathbf{x}).$$

For any type of sensible size of image this summation is not computationally tractable. What we want to sum over is all possible values that $\mathbf{x}$ can actually take, i.e. we want to test **all** possible binary images. If

[26] An interesting note of machine learning researchers is that very few comes from a computer science background, much more common are physicists, engineers and of course statisticians. Maybe it is therefore not surprising to see a lot of physics motivated models in use for rather different tasks commpared to what they where initially designed.

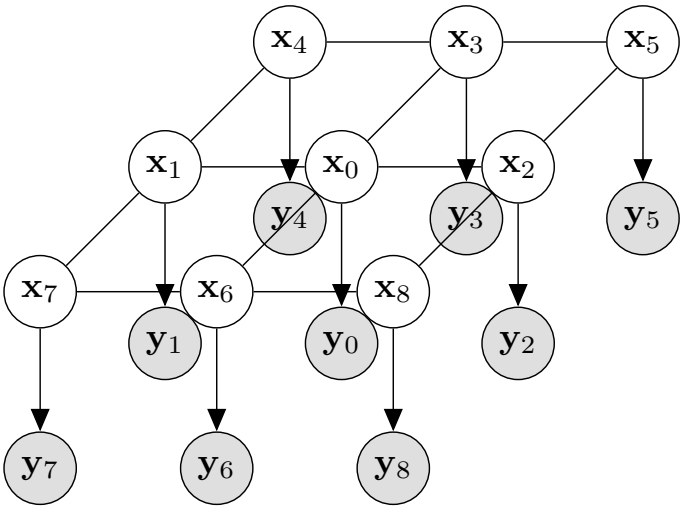

Figure 15: *Above is the graphical model of the MRF we will use for the images. Note that we have connected the latent variables with lines and not arrows, that is because we do not specify these as conditional probabilities but rather joint probabilities.*

we have an image of size 10 it consists of 100 different pixels. The number of combinations that they can take is therefore 2^{100}. This is the number of terms in the marginalisation above and its a *big* number. This means that it is simply computationally intractable to compute Baye's rule for any sensibly sized image, say something with 3-4 megapixels, and we need to perform some form of approximation to proceed. As a side-note think back on the previous labs where we often could use conjugacy to avoid these calculations, we summed over a space with *infinite* number of terms to reach the posterior when we did Gaussian processes without actually directly computing the evidence. Now think how nice conjugacy is.

We will now proceed to look at three different approaches to inferring the true pixel values in the above the first method is just a simple coordinate-wise gradient approach while the two others are more principled approximations. The lab should be fairly straight forward to code up but what I want you to try and do, where I think you will learn the most, is by playing around with the parameters, initialisation etc. so that you get an intuitive understanding for what is going on. Right lets get started!

8.3 Data

First we need some data to work with, you can use any image that you want as a starting point, if in doubt I can definitely recommend images of pugs, they work great. To make our life a bit easier we use grey-scale images rather than colour. You can use the Imagemagic to convert between colour and grey-scale and resize the image to something sensible.

Code

```
convert -resize 128x <image-in> <image-out>
convert <image-in> -set colorspace Gray -auto-level -threshold 50% <image-out>
```

Once the image is converted we can load it into python and create a noisy version of it. The code below has two different types of noise, either Gaussian noise or 'salt-and-pepper' noise which flips the pixel values[27].

[27]These noise distributions are very simple, you can also try to code up something more interesting. How about drawing random lines across the image in black or white? When you got your code up and running try to extend the noise models as this will give you a better idea of how the inference actually works.

Code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import imread

def add_gaussian_noise(im,prop,varSigma):
    N = int(np.round(np.prod(im.shape)*prop))

    index = np.unravel_index(np.random.permutation(np.prod(im.shape))[1:N],im.shape)
    e = varSigma*np.random.randn(np.prod(im.shape)).reshape(im.shape)
    im2 = np.copy(im).astype('float')
    im2[index] += e[index]

    return im2
def add_saltnpeppar_noise(im,prop):
    N = int(np.round(np.prod(im.shape)*prop))
    index = np.unravel_index(np.random.permutation(np.prod(im.shape))[1:N],im.shape)
    im2 = np.copy(im)
    im2[index] = 1-im2[index]

    return im2

# proportion of pixels to alter
prop = 0.7
varSigma = 0.1

im = imread('text.png')
im = im/255
fig = plt.figure()
ax = fig.add_subplot(131)
ax.imshow(im,cmap='gray')

im2 = add_gaussian_noise(im,prop,varSigma)
ax2 = fig.add_subplot(132)
ax2.imshow(im2,cmap='gray')
im2 = add_saltnpeppar_noise(im,prop)
ax3 = fig.add_subplot(133)
ax3.imshow(im2,cmap='gray')
```

In order to process the images we are also likely to need some helper code to access the image. In specific our prior requires us to compute the neighbours of a specific node, the code below computes the *4-neighbourhood* of a node so it does not include the diagonals. You might try to extend the code below and include the diagonals as well as this should improve the results.

Code

```
def neighbours(i,j,M,N,size=4):
      if size==4:
            if (i==0 and j==0):
                  n=[(0,1), (1,0)]
            elif i==0 and j==N-1:
                  n=[(0,N-2), (1,N-1)]
            elif i==M-1 and j==0:
                  n=[(M-1,1), (M-2,0)]
            elif i==M-1 and j==N-1:
                  n=[(M-1,N-2), (M-2,N-1)]
            elif i==0:
                  n=[(0,j-1), (0,j+1), (1,j)]
            elif i==M-1:
                  n=[(M-1,j-1), (M-1,j+1), (M-2,j)]
            elif j==0:
                  n=[(i-1,0), (i+1,0), (i,1)]
            elif j==N-1:
                  n=[(i-1,N-1), (i+1,N-1), (i,N-2)]
            else:
                  n=[(i-1,j), (i+1,j), (i,j-1), (i,j+1)]

            return n
      if size==8:
            print('Not yet implemented\n')

            return -1
```

Now we have most of the useful code we need and its time to move on the the specific inference algorithms.

8.4 Iterative Conditional Modes (ICM)

The first approach we should take is something called Iterative Conditional Modes you can read about it in Section 8.3.3 in [5]. This approach works like this, we will first initialise the latent variables to something, then we will fix all variables except for one and see what is the most likely state for this one to be in given that we assume all others to be correct. We will then iteratively do this for all the nodes and if we manage to go through one pass of all nodes without changing then we have reached a local minima. However, just to get some control of things, I'm going to run it a fixed set of iterations in the code below, but I do recommend that you keep a flag for changes so that you can bail early, or know if you have found a local minima. You can see the algorithm in Algorithm 2 where $\mathbf{x}_{\neg i}$ implies all $\mathbf{x}$ except x_i.

Algorithm 2 Iterative Conditional Modes for Ising Model

procedure IMAGE DENOISING WITH ICM
  $\mathbf{x} \leftarrow$ initialisation
  for $\tau = 1 \dots T$ **do**
    for $i = 1 \dots N$ **do**
      if $p(x_i = 1, \mathbf{x}_{\neg i}, \mathbf{y}) > p(x_i = -1, \mathbf{x}_{\neg i}, \mathbf{y})$ **then**
        $x_i = 1$
      else
        $x_i = -1$
  return $\mathbf{x}$

Question 3

Implement ICM for a binary image that you have corrupted with noise, show the result for a set of images with different noise levels. How many passes through the nodes do you need until you are starting to get descent results?

8.5 Stochastic Inference

Now we should pick up a bit more advanced algorithm to try and find the latent variables from data. What we will do here is to implement a simple Gibbs sampler Ch 11.3 [5]. Gibbs sampling is often quite easy to implement so it is often one of the first approaches you try to get something out of a model and see if it is worth developing a specific taylored inference scheme.

8.5.1 Basic Sampling

The idea behind sampling is that we want to compute and expectations over a function where the closed form is intractable,

$$\mathbb{E}_{p(\mathbf{z})}[f] = \int f(\mathbf{z})p(\mathbf{z})\mathrm{d}\mathbf{z}.$$

We now try to convert the integral to a discrete sum of values that are draw from $p(\mathbf{z})$ as,

$$\hat{f} = \frac{1}{L}\sum_{l=1}^{L} f(\mathbf{z}^{(l)}) \tag{142}$$

$$\mathbf{z}^{(l)} \sim p(\mathbf{z}). \tag{143}$$

The important thing to note here is that the approximation will depend on us drawing "good" samples, this is really what sampling is about different strategies to get informative samples.

8.5.2 Markov Chain Monte Carlo

One of the strategies to get more efficient sampling is referred to as Markov Chain Monte Carlo methods or MCMC for short. You will see them pop up in many different topics, if you are taking computer graphics in TB2 you will bump into them as high fidelity graphics is a lot about intractable integrals. MCMC was developed as a part of the Manhattan project[28] in the development of the first nuclear bomb where nummerical solutions where sought to complicated or intractable problems. The idea behind MCMC is to let the sequence of samples come from a Markov chain such that when we draw a new sample we take the previous evaluations into consideration. Note that this does not mean that the samples are not independent according to the distribution we want to sample from. In specific we will use a proposal distribution $q(\mathbf{z}|\mathbf{z}^{(\tau)})$ to draw samples from where $\mathbf{z}^{(\tau)}$ is the current state of our sampling chain.

8.5.3 Gibbs Sampling

Gibbs sampling is probably the most straight-forward type of MCMC method. It is widely used and very easy to implement. The idea behind a Gibbs sampler is if we have a distribution $p(\mathbf{z})$ that we wish to draw samples from we will draw samples from each dimension in turn where we condition on the other dimensions. In specific we will sample from,

$$p(z_i|\mathbf{z}_{\neg i}),$$

where $\mathbf{z}_{\neg i}$ is all dimensions of $\mathbf{z}$ except for i. We will then replace z_i with our samples and "rotate/cycle" through the variables. The idea behind a Gibbs sampler is outlined in Algorithm 3. Now lets try and relate this to our specific task that of image denoising.

[28] https://en.wikipedia.org/wiki/Manhattan_Project

Algorithm 3 Gibbs Sampling

procedure GIBBS SAMPLER FOR $p(\mathbf{z})$
  $\mathbf{z} \leftarrow$ initialisation
  for $\tau = 1 \dots T$ **do**
    $z_1^{(\tau+1)} \approx p(z_1 | z_2^{(\tau)}, \dots, z_N^{(\tau)})$
    $z_2^{(\tau+1)} \approx p(z_2 | z_1^{(\tau)+1}, z_3^{(\tau)}, \dots, z_N^{(\tau)})$
    $\vdots$
    $z_N^{(\tau+1)} \approx p(z_2 | z_1^{(\tau)+1}, z_2^{(\tau+1)}, \dots, z_{N-1}^{(\tau+1)})$
  return z

8.5.4 Gibbs Sampling in an Ising Model

We have now described how to sample from a general multivariate distribution $p(\mathbf{z})$ now we want to try and use this scheme to our MRF Ising model. In specific we are interested in sampling from the "unreachable" posterior $p(\mathbf{x}|\mathbf{y})$. The key thing underpinning Gibbs sampling is that even though the multivariate posterior might be unreachable, it should be much simpler to get the posterior over a single variable. If this is possible then we can run Gibbs sampling by rotating through the posterior over a single variable x_i. To begin lets formulate this distribution over a general node x_i,

$$p(x_i|\mathbf{x}_{\neg i}, \mathbf{y}) = \frac{p(\mathbf{x}, \mathbf{y})}{p(\mathbf{x}_{\neg i}, \mathbf{y})}. \tag{144}$$

To proceed we need to compute the marginal likelihood above, this turns out to be rather easy as we are working with binary data,

$$p(\mathbf{x}_{\neg i}, \mathbf{y}) = \int p(\mathbf{x}, \mathbf{y}) \mathrm{d}x_i = \sum_{x_i \in [1,-1]} p(x_i, \mathbf{x}_{\neg i}, \mathbf{y}) \tag{145}$$

$$= p(x_i = 1, \mathbf{x}_{\neg i}, \mathbf{y}) + p(x_i = -1, \mathbf{x}_{\neg i}, \mathbf{y}). \tag{146}$$

So now lets say that we want to compute the posterior over $x_i = 1$ we can write this up as,

$$p(x_i = 1|\mathbf{x}_{\neg i}, \mathbf{y}) = \frac{p(x_i = 1, \mathbf{x}_{\neg i}, \mathbf{y})}{p(x_i = 1, \mathbf{x}_{\neg i}, \mathbf{y}) + p(x_i = -1, \mathbf{x}_{\neg i}, \mathbf{y})}, \tag{147}$$

which we can evaluate as we know $\mathbf{y}$ and $\mathbf{x}$. However, what we want to do is to sample from the posterior over x_i but as we do not even know what form the distribution is we are going to do a simple trick. We will evaluate $p(x_i = 1|\mathbf{x}_{\neg i}, \mathbf{y})$ and then we will draw a random number z uniformly from $(0, 1)$ and then we will pick x_i from this distribution where we use the posteriors as proportions. Below is a couple of example that hopefully explains things clearly,

$p(x_i = 1\|\mathbf{x}_{\neg i}, \mathbf{y})$	z	x_i^{sample}
0.5	0.7	-1
0.5	0.2	1
0.1	0.05	1
0.1	0.2	-1

You could now go straight on to implement the Gibbs sampler but it would most likely be very slow as you would have to evaluate distributions with very many parameters every iterations. To speed things up

Algorithm 4 Gibbs Sampling Ising Model

procedure GIBBS SAMPLER FOR $p(\mathbf{x}|\mathbf{y})$
  $\mathbf{x}^{(0)} \leftarrow$ initialisation
  for $\tau = 0 \dots T$ **do**
    for $i = 1 \dots N$ **do**
      $p_i = p(x_i = 1 | \{x_j^{(\tau+1)}\}_{j=1}^{i-1}, \{x_j^{(\tau)}\}_{j=i+1}^{N}, \mathbf{y})$
      $t \sim \text{Uniform}(0, 1)$
      if $p_i > t$ **then**
        $x_i^{\tau+1} = 1$
      else
        $x_i^{\tau+1} = -1$
  return $\mathbf{x}^{(T)}$

we can use the structure that exists in the problem. Lets write up our posterior,

$$p(x_i = 1|\mathbf{x}_{\neg i}, \mathbf{y}) = \frac{p(x_i, \mathbf{x}_{\neg_i}, \mathbf{y})}{p(\mathbf{x}_{\neg i}, \mathbf{y})} \tag{148}$$

$$= \frac{p(y_i|x_i = 1) \prod_{j \neq i} p(y_j|x_j) p(x_i = 1, \mathbf{x}_{\neg i})}{p(y_i|x_i = 1) \prod_{j \neq i} p(y_j|x_j) p(x_i = 1, \mathbf{x}_{\neg i}) + p(y_i|x_i = -1) \prod_{j \neq i} p(y_j|x_j) p(x_i = -1, \mathbf{x}_{\neg i})} \tag{149}$$

$$= \frac{p(y_i|x_i = 1) p(x_i = 1, \mathbf{x}_{\neg i})}{p(y_i|x_i = 1) p(x_i = 1, \mathbf{x}_{\neg i}) + p(y_i|x_i = -1) p(x_i = -1, \mathbf{x}_{\neg i})} \tag{150}$$

$$= \frac{p(y_i|x_i = 1) p(x_i = 1, \mathbf{x}_{\mathcal{N}(i)})}{p(y_i|x_i = 1) p(x_i = 1, \mathbf{x}_{\mathcal{N}(i)}) + p(y_i|x_i = -1) p(x_i = -1, \mathbf{x}_{\mathcal{N}(i)})} \tag{151}$$

where $\mathbf{x}_{\mathcal{N}(i)}$ is the set of nodes that are in the neighbourhood of x_i. This is a much smaller computation where we have exploited the structure in the problem in two ways, first that the likelihood factorises, this means we only have to compute one single term and secondly that the prior also factorises into the Markov blanket of x_i. Now we are ready to implement the Gibbs sampler for our Ising model. The code that you need to write should follow the Algorithm 4.

Question 4

Implement the Gibbs sampler for the image denoising task. Generate images with different amount of noise and see where it fails and where it works. My result is shown in Figure 16.

Question 5

There is nothing saying that you should cycle through the nodes in the graph index by index, you can pick any different order and you do not have to visit each node equally many times either. Alter your sampler so that it picks and updates a random node each iteration. Are the results different? Do you need more or less iterations? To get reproduceable results fix the random seed in your code with `np.random.seed(42)`.

Question 6

What effect does the number of iterations we run the sampler have on the results? Try to run it for different times, does the result always get better?

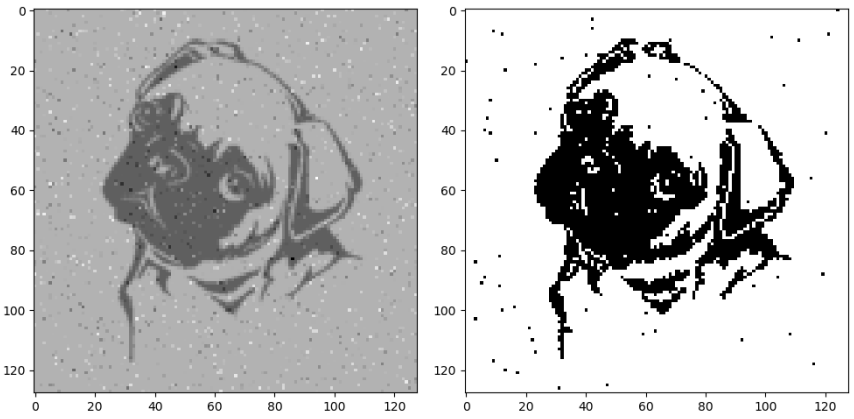

Figure 16: *The result of running the Gibbs sampling approach in the Ising model. The left image is the noisy version while the rightmost is the cleaned up version*

8.6 Summary

In this lab you have seen how we can approach an intractable computation using an approximative method. The method that you have done can easily be extended to work with colour images but then you have to alter the likelihood function to something a bit more interesting. Next week we will use the same model but perform a deterministic approximation instead.

References

[1] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian Processes for Data-Efficient Learning in Robotics and Control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, stat.ML(2):408–423, February 2015.

[2] M. Kaiser, C. Otte, T. Runkler, and C. H. Ek. Bayesian alignments of warped multi-output gaussian processes. In *Advances in Neural Information Processing Systems 32, [NIPS Conference, Montreal, Quebec, Canada, December 3 - December 8, 2018]*, 2018.

[3] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[4] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. Bayesian optimization in alphago. *CoRR*, 2018.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] Neil D Lawrence. Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *Journal of Machine Learning Research*, 6:1783–1816, 2005.

[7] Keith Grochow, Steven L Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *SIGGRAPH '04: SIGGRAPH 2004 Papers*, August 2004.

[8] Raquel Urtasun, David J Fleet, and P. Fua. 3D people tracking with Gaussian process dynamical models. *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, 1:238–245, 2006.

[9] Steindór Sæmundsson, Katja Hofmann, and Marc Peter Deisenroth. Meta reinforcement learning with latent variable gaussian processes. *CoRR*, 2018.

[10] K. B. Petersen and M. S. Pedersen. The Matrix Cookbook, November 2012. Version 20121115.

[11] Iain Murray and Zoubin Ghahramani. A note on the evidence and Bayesian Occam's razor. Technical Report GCNU-TR 2005-003, August 2005.

[12] J. Močkus. On bayesian methods for seeking the extremum. In G. I. Marchuk, editor, *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*, pages 400–404, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

9 Variational Inference

Abstract

Now we will move on to a deterministic approximation of the Ising Model. What we will do is to specify a surrogate model and try to fit this model so that it is as close as possible to the actual model. We will first go through the idea of Variational Bayes and then proceed to go through and look at the specific approach we will use for the Ising model.

Math

This derivation is long, I go through the full proof of, variational Bayes, mean-field and mean-field variational Bayes for the Ising model. I tried to be as complete as possible so everything should be self-contained. However, there is likely to be blunders in there so if you find something strange point it out to me. Importantly, you do not need to know any of this, its here if you are really interested in machine learning but none of the questions on the exam will be related to this. If you want to skip the derivations read 9.1 and then move ahead to 9.4. However, if I am allowed to say it myself this is quite beautiful so if you enjoy math I suggest you continue reading.

9.1 Variational Bayes

Inference is the task of fitting our model to some observed data, what we often do is to try to choose the model that maximises the evidence. If we have been given some data $\mathbf{y}$ and have some parameters θ to fit we wish to pick them such that,

$$\hat{\theta} = \text{argmax}_\theta p(\mathbf{y}).$$

As we know the evidence is often intractable to compute but lets see what we can do,

$$\log p(\mathbf{Y}) = \log \int p(\mathbf{Y}, \mathbf{X}) \mathrm{d}\mathbf{X} = \log \int p(\mathbf{X}|\mathbf{Y}) p(\mathbf{Y}) \mathrm{d}\mathbf{X} \tag{152}$$

$$= \log \int \frac{q(\mathbf{X})}{q(\mathbf{X})} p(\mathbf{X}|\mathbf{Y}) p(\mathbf{Y}) \mathrm{d}\mathbf{X}. \tag{153}$$

The strange thing here is the second row where we have added a distribution $q(\mathbf{X})$ to the equation. This is just a general distribution and because we add it in this form it will not change the integral at all. What we will do now is try to formulate a bound on this integral, in specific we will use something called the Jensen inequality. The Jensen inequality states that a line between two points on the curve will always be above the curve see Figure 17.

$$\lambda f(x_0) + (1-\lambda) f(x_1) \geq f(\lambda x_0 + (1-\lambda) x_1)$$
$$x \in [x_{min}, x_{max}]$$
$$\lambda \in [0, 1]]$$

Even though it might seem trivial it is a very useful property for dealing with probabilities. When we are marginalising variables from our model we are computing expectations, if we are computing an expectation of a convex function, the expectation of the function will always be an upper bound on the function applied to the expectations,

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]) \tag{154}$$

$$\int f(x) p(x) \mathrm{d}x \geq f\left(\int x p(x) \mathrm{d}x \right) \tag{155}$$

Where this is specifically important is when the function is a logarithm. As a logarithm is a concave function the inequality just flips around as can be seen in Figure **??**. This means that the logarithm of an

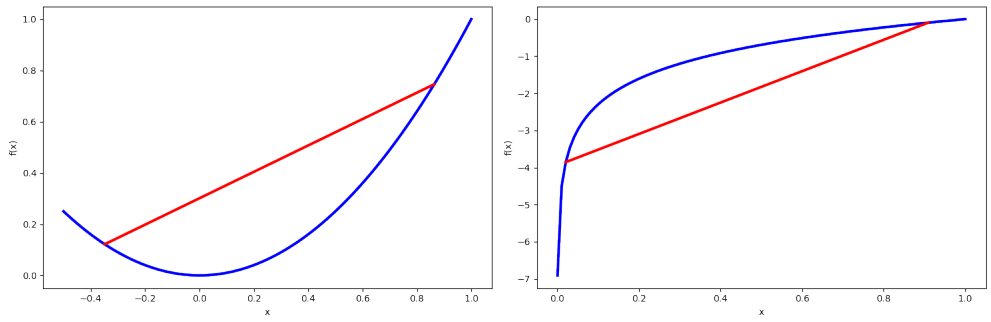

Figure 17: *The plot on the left shows a convex function in blue and line connecting two of the points on the function. The Jensen inequality implies that if you "move" a point along the red line it will always be above the blue. On the right the blue function is a logarithm and we can see that the reverse behaviour is true, the red line will always be below the blue. This means that we can say that, "the red line will always be a lower bound on the blue".*

integral is a upper bound on the log of the integral,

$$\int \log(x)p(x)\mathrm{d}x \leq \log\left(\int xp(x)\mathrm{d}x\right). \tag{156}$$

We will now exploit this result to try and find a bound on the intractable evidence,

$$\log\, p(\mathbf{Y}) = \log \int \frac{q(\mathbf{X})}{q(\mathbf{X})} p(\mathbf{X}|\mathbf{Y})p(\mathbf{Y})\mathrm{d}\mathbf{X} = \tag{157}$$

$$\geq \int q(\mathbf{X})\log\frac{p(\mathbf{X}|\mathbf{Y})p(\mathbf{Y})}{q(\mathbf{X})}\mathrm{d}\mathbf{X} \tag{158}$$

$$= \int q(\mathbf{X})\log\frac{p(\mathbf{X}|\mathbf{Y})}{q(\mathbf{X})}\mathrm{d}\mathbf{X} + \int q(\mathbf{X})\mathrm{d}\mathbf{X}\log\, p(\mathbf{Y}) \tag{159}$$

$$= -\mathrm{KL}\left(q(\mathbf{X})||p(\mathbf{X}|\mathbf{Y})\right) + \log\, p(\mathbf{Y}). \tag{160}$$

The important part of the computation above is where we move the logarithm inside the integral by exploiting the bound. Importantly the first term after having split the integral into two is what is known as the Kullback-Leibler divergence Ch 1.6.1 [5]. This is a measure of "simiarity" between probability distributions. It is not a metric, its for example not symmetric, but importantly it is only 0 if the two distributions are the same and positive in all other cases. This leads us to an important observation, if $q(\mathbf{X}) = p(\mathbf{X}|\mathbf{Y})$ then the bound is tight. Importantly in order to reach the posterior distribution $p(\mathbf{X}|\mathbf{Y})$ we would have to compute the evidence. So this leads us to the central intuition of Variational Bayes, if we can pick a distribution $q(\mathbf{X})$ such that it is as close as possible to the posterior $p(\mathbf{X}|\mathbf{Y})$ we will have a good surrogate model, if it is exact, it is the same model. Therefore lets try to minimise the KL-divergence between the two distributions.

$$\mathrm{KL}(q(\mathbf{X})||p(\mathbf{X}|\mathbf{Y})) = \int q(\mathbf{X})log\frac{q(\mathbf{X})}{p(\mathbf{X}|\mathbf{Y})}\mathrm{d}\mathbf{X} \tag{161}$$

$$= \int q(\mathbf{X})log\frac{q(\mathbf{X})}{p(\mathbf{X},\mathbf{Y})}\mathrm{d}\mathbf{X} + log\, p(\mathbf{Y}) \tag{162}$$

$$= H(q(\mathbf{X})) - \mathbb{E}_{q(\mathbf{X})}\left[log\, p(\mathbf{X},\mathbf{Y})\right] + log\, p(\mathbf{Y}). \tag{163}$$

What we have done above is to write up the divergence as an expectation over the joint distribution and a term that only dependes on $q(\mathbf{X})$ and the evidence. If we now move the evidence over on the other side of

the expression we will get this formulation,

$$log\ p(\mathbf{Y}) = \mathrm{KL}(q(\mathbf{X})||p(\mathbf{X}|\mathbf{Y})) + \underbrace{\mathbb{E}_{q(\mathbf{X})}\left[log\ p(\mathbf{X},\mathbf{Y})\right] - H(q(\mathbf{X}))}_{\text{ELBO}} \tag{164}$$

$$\geq \mathbb{E}_{q(\mathbf{X})}\left[log\ p(\mathbf{X},\mathbf{Y})\right] - H(q(\mathbf{X})) = \mathcal{L}(q(\mathbf{X})). \tag{165}$$

As we know the KL-divergence has to be positive the remaining term is a lower-bound on the evidence. This is why this is referred to as the *ELBO-Evidence Lower BOund*.

So that was a whole lot of math but what does it all mean, what does this lead us to, has this actually solved anything? Well if we look at the formula above, what we want to find is $q(\mathbf{X})$ if we do find this, we know that it is an approximation of the true posterior $p(\mathbf{X}|\mathbf{Y})$ this is really useful, further the bound is specified by the computation of an expectation over the *joint* distribution of the data. 1) if we cannot formulate the joint distribution we are in bigger trouble and 2) we are allowed to choose the distribution $q(\mathbf{X})$ that we have to take the expectation over. Clearly this should be simpler to do.

In the next part we will derive a specific family of approximations often referred to as mean-field approximations[29]. They are often not particularly exact but they do work in most cases.

9.2 Mean Field Approximation

The mean-field approximation assumes that the approximative posterior factorises over all the variables as,

$$q(\mathbf{X}) = \prod_i q_i(\mathbf{x}_i).$$

We will proceed by deriving the bound related to this type of approximative distribution.

$$\mathcal{L}(q) = \int q(\mathbf{X}) \log \frac{p(\mathbf{Y},\mathbf{X})}{q(\mathbf{X})} \mathrm{d}\mathbf{X} = \int \prod_i q_i(\mathbf{x}_i) \log \frac{p(\mathbf{Y},\mathbf{X})}{\prod_k q_k(\mathbf{x}_k)} \mathrm{d}\mathbf{X} \tag{166}$$

$$= \int \prod_i q_i(\mathbf{x}_i) \left(\log p(\mathbf{Y},\mathbf{X}) - \sum_k \log q_k(\mathbf{x}_k) \right) \tag{167}$$

For many types of models we want to update several distributions. What we will do now is to derive a scheme where we update one component in turn. Therefore we would like to re-write $\mathcal{L}(q)$ in such a manner that we can single out a single component,

$$\mathcal{L}(q) = \mathcal{L}(q_j) + \mathcal{L}(q_{\neg j}),$$

where $\neg j$ means all the components except for j.

[29] Again these are models that initially was suggested in physics to model phase transitions first described by Pierre Curie. If you read the literature on Variational Bayes you will often hear the ELBO referred to as the variational free energy, and the first terms the bound as *energy* as the second term corresponds to the *entropy* of the approximating distribution. So don't forget your theromdynamics.

$$\mathcal{L}(q) = \int \prod_i q_i(\mathbf{x}_i) \left(\log p(\mathbf{Y}, \mathbf{X}) - \sum_k \log q_k(\mathbf{x}_k) \right) \tag{168}$$

$$= \int_j \int_{\neg j} q_j(\mathbf{x}_j) \prod_{i \neq j} q_i(\mathbf{x}_i) \left(\log\, p(\mathbf{X}, \mathbf{Y}) - \sum_k \log\, q_k(\mathbf{x}_k) \right) \mathrm{d}\mathbf{x}_{\neg j} \mathrm{d}\mathbf{x}_j \tag{169}$$

$$= \int_j q_j(\mathbf{x}_j) \underbrace{\int_{\neg j} \prod_{i \neq j} q(\mathbf{x}_i) \log\, p(\mathbf{Y}, \mathbf{X}) \mathrm{d}\mathbf{x}_{\neg j}}_{\log\, f_j(\mathbf{x}_j)} \mathrm{d}\mathbf{x}_j$$

$$- \int_j q_j(\mathbf{x}_j) \int_{\neg j} \prod_{i \neq j} q(\mathbf{x}_i) \left(\log\, q_j(\mathbf{x}_j) + \sum_{k \neq j} \log\, q_k(\mathbf{x}_k) \right) \mathrm{d}\mathbf{x}_{\neg j} \mathrm{d}\mathbf{x}_j \tag{170}$$

$$= \int_j q_j(\mathbf{x}_j) \log\, f_j(\mathbf{x}_j) \mathrm{d}\mathbf{x}_j$$

$$- \int_j q_j(\mathbf{x}_j) \left(\log\, q_j(\mathbf{x}_j) \underbrace{\int_{\neg j} \prod_{i \neq j} q_i(\mathbf{x}_i) \mathrm{d}\mathbf{x}_{\neg j}}_{=1} + \underbrace{\int_{\neg j} \prod_{i \neq j} q_i(\mathbf{x}_i) \sum_{k \neq j} \log\, q_k(\mathbf{x}_k) \mathrm{d}\mathbf{x}_{\neg j}}_{\text{constant w.r.t. } q_j} \right) \tag{171}$$

$$= \int_j q_j(\mathbf{x}_j) \log\, f_j(\mathbf{x}_j) \mathrm{d}\mathbf{x}_j - \int_j q_j(\mathbf{x}_j) \log\, q_j(\mathbf{x}_j) \mathrm{d}\mathbf{x}_j + \text{const.} \cdot \underbrace{\int_j q_j(\mathbf{x}_j) \mathrm{d}\mathbf{x}_j}_{=1} \tag{172}$$

$$= \int_j q_j(\mathbf{x}_j) \log\, \frac{f_j(\mathbf{x}_j)}{q_j(\mathbf{x}_j)} \mathrm{d}\mathbf{x}_j + \text{const.} \tag{173}$$

$$= - \int_j q_j(\mathbf{x}_j) \log\, \frac{q_j(\mathbf{x}_j)}{f_j(\mathbf{x}_j)} \mathrm{d}\mathbf{x}_j + \text{const.} \tag{174}$$

$$= -\mathrm{KL}(q_j(\mathbf{x}_j) || f_j(\mathbf{x}_j)) + \text{const.} \tag{175}$$

The above derivation ends up somewhere rather intuitive, to maximise the lower bound on the evidence with respect to $q_j(\mathbf{x}_j)$ we want to minimise the KL-divergence between the factor $q_j(\mathbf{x}_j$ and the distribution when all *other* factors have been averaged out. Have a look at what the term $f_j(\mathbf{x}_j)$ to see if this makes sense.

As we know that the KL-divergence is always positive and as we are free to choose $q_j(\mathbf{x}_j)$ as we wish we can simply set,

$$q_j(\mathbf{x}_j) = f_j(\mathbf{x}_j) \tag{176}$$

$$\log\, f_j(\mathbf{x}_j) = \int_{\neg j} \underbrace{\prod_{i \neq j} q_i(\mathbf{x}_i)}_{q_{\neg j}(\mathbf{x}_{\neg j})} \log\, p(\mathbf{Y}, \mathbf{X}) \mathrm{d}\mathbf{x}_{\neg j} \tag{177}$$

$$= \mathbb{E}_{q_{\neg j}(\mathbf{x}_{\neg j})} \left[\log\, p(\mathbf{Y}, \mathbf{X}) \right] \tag{178}$$

So in order to use the mean-field variational bayes we need to pick the approximate distribution $q(\mathbf{X})$ in such a way that we can compute the expectation above. Now we have derived both variational bayes and the mean field approximation we are ready to move back to our model and work specifically with the Ising model we have defined.

9.3 Mean Field Variational Bayes in Ising Model

Now let us formulate the mean field approximation for the Ising model, lets first remind ourselves of the model. We specified a prior of the form,

$$p(\mathbf{x}) = \frac{1}{Z_0} e^{E_0(\mathbf{X})} \tag{179}$$

$$E_0(\mathbf{X}) = \sum_i^N \sum_{j \in (i)} w_{ij} x_i x_j \tag{180}$$

If we look at the term $w_{ij}x_ix_j$ we can see that it will be postive if the latent values are the same and negative otherwise. The larger the value the higher the probability which fits well with our Ising model. The other term we need is the likelhood,

$$p(\mathbf{Y}|\mathbf{X}) = \prod_i p(y_i|x_i) = \frac{1}{Z_1} \prod_i e^{L_i(x_i)}, \tag{181}$$

where the function L_i should give a large value if it is likely that x_i have generated y_i. Now we are ready to formulate our approximate distribution. We will use a full mean-field approximation so we will assume that the approximate distribution over each latent variable is independent,

$$q(\mathbf{x}) = \prod_i q(x_i, \mu_i),$$

where we have introduced μ_i as a variational parameter that parametrises this distribution. In specific μ_i will be $\mathbb{E}_{q_i}[x_i]$. The first thing we need to get is the joint distribution of the model. We will through out the task work in log-space which gives us,

$$\log p(\mathbf{x}, \mathbf{y}) = \log p(\mathbf{y}|\mathbf{x}) p(\mathbf{x}) \tag{182}$$

$$= \log \left(\prod_i e^{L_i(x_i)} \frac{1}{Z_0} e^{\sum_{j \in \mathcal{N}(i)} w_{ij} x_i x_j} \right) \tag{183}$$

$$= \sum_i \left(L_i(x_i) + \sum_{j \in \mathcal{N}(i)} w_{ij} x_i x_j \right) + \text{const.} \tag{184}$$

As we have choosen a fully factorised approximative distribution $q(\mathbf{x})$ we will compute each expectation in the bound in turn so we want to write up the joint distribution where we only consider one variable. This means,

$$\log p(\mathbf{x}, \mathbf{y}) = L_i(x_i) + x_i \sum_{j \in \mathcal{N}(i)} w_{ij} x_j + \text{const.},$$

where we have included all the term over the remaining latent variables in the constant term. We are now ready to compute the expectation to get the approximative posterior,

$$\log q_i(x_i) = \log f_i(x_i) = \int \prod_{j \neq i} q_j(x_j) \log p(\mathbf{x}, \mathbf{y}) \mathrm{d}x_{\neg i} \tag{185}$$

$$= \int \prod_{j \neq i} q_j(x_j) (L_i(x_i) + \sum_{k \in \mathcal{N}(i)} w_{ik} x_i x_k + \text{const.}) \mathrm{d}x_{\neg i} \tag{186}$$

$$= \underbrace{\int \prod_{j \neq i} q_j(x_j) \mathrm{d}x_{\neg i}}_{=1} L_i(x_i) + \int \prod_{j \neq i} q_j(x_j) \sum_{k \in \mathcal{N}(i)} w_{ik} x_i x_k \mathrm{d}x_{\neg i} + \text{const.} \tag{187}$$

The first integral will compute to one as $q_j(x_j)$ is a distribution. The second term is a bit trickier to deal with so we are going to deal with it on its own.

$$\int \prod_{j\neq i} q_j(x_j) \sum_{k\in\mathcal{N}(i)} w_{ik} x_i x_k \mathrm{d}x_{\neg i} = \int \prod_{j\neq i} q_j(x_j) x_i \sum_{k\in\mathcal{N}(i)} w_{ik} x_k \mathrm{d}x_{\neg i} \tag{188}$$

$$= \int x_i \sum_{k\in\mathcal{N}(i)} w_{ik} \left(\prod_{j\neq i} q_j(x_j)\right) x_k \mathrm{d}x_{\neg i} = x_i \sum_{k\in\mathcal{N}(i)} w_{ik} \int \left(\prod_{j\neq i} q_j(x_j)\right) x_k \mathrm{d}x_{\neg i}. \tag{189}$$

We will now expand the integration over each term and find something rather beautiful,

$$x_i \sum_{k\in\mathcal{N}(i)} w_{ik} \int \left(\prod_{j\neq i} q_j(x_j)\right) x_k \mathrm{d}x_{\neg i} = \tag{190}$$

$$x_i \sum_{k\in\mathcal{N}(i)} w_{ik} \int \left(q_1(x_1)q_2(x_2)\cdot\ldots\cdot q_N(x_N)\right) x_k \mathrm{d}x_1 \mathrm{d}x_2 \cdot\ldots\cdot \mathrm{d}x_N \tag{191}$$

$$= x_i \sum_{k\in\mathcal{N}(i)} w_{ik} \underbrace{\int q_1(x_1)\mathrm{d}x_1}_{=1} \underbrace{\int q_2(x_2)\mathrm{d}x_2}_{=1} \cdot\ldots\cdot \int q_k(x_k)x_k\mathrm{d}x_k \cdot\ldots\cdot \underbrace{\int q_N(x_N)\mathrm{d}x_N}_{=1} \tag{192}$$

$$= x_i \sum_{k\in\mathcal{N}(i)} w_{ik} \int q_k(x_k)x_k\mathrm{d}x_k = x_i \sum_{k\in\mathcal{N}(i)} w_{ik} \mathbb{E}_{q_k(x_k)}[x_k] = \tag{193}$$

$$= x_i \sum_{k\in\mathcal{N}(i)} w_{ik}\mu_k \tag{194}$$

Now we are ready to tidy things up and write out the approximative posterior for x_i by combining our terms,

$$\log\, q_i(x_i) = \log\, f_i(x_i) = L_i(x_i) + x_i \underbrace{\sum_{k\in\mathcal{N}(i)} w_{ik}\mu_k}_{m_i} + \text{const.} \tag{195}$$

$$= L_i(x_i) + x_i \cdot m_i + \text{const.} \tag{196}$$

The expression above does make sense, we have one term which relates to the observations at x_i and one term which relates to the prior term relating the expectations of the nearby latent locations. This means that we can write our approximative distribution as,

$$q(\mathbf{x}) \propto \prod_i q_i(x_i) \propto e^{x_i m_i + L_i(x_i).}$$

We need to make sure that the approximation that we have is an actual distribution therefore making sure that it integrates to 1. In this case this is really simply as $x_i \in [1, -]$ and therefore we can write it up as,

$$\hat{q}_i(x_i = 1) = \frac{1}{q(x_i = 1) + q(x_i = -1)} q(x_i = 1) = \frac{e^{m_i + L_i(1)}}{e^{m_i + L_i(1)} + e^{-m_i + L_i(-1)}} \tag{197}$$

$$= \left\{ \begin{array}{c} \text{Simplification:} \\ \frac{e^a}{e^a + e^b} = \frac{1}{e^{-a}(e^a + e^b)} = \frac{1}{1 + e^{b-a}} \end{array} \right\} \tag{198}$$

$$= \frac{1}{1 + e^{-2m_i - L_i(1) + L_i(-1)}} = \frac{1}{1 + e^{-2\underbrace{(m_i + \frac{1}{2}L_i(1) - \frac{1}{2}L_i(-1))}_{a_i}}} \tag{199}$$

$$= \frac{1}{1 + e^{-2a_i}} = \text{sigm}(2a_i). \tag{200}$$

As the probability for x_i taking value 1 is equal to a sigmoid the probability for the other case is trivial,

$$q_i(x_i = -1) = \text{sigm}(-2a_i)$$

Importantly the proposal distribution is completely defined by its expected value μ_i as a last step we now want to find a way to update this parameter. This is easy to do by going through its definition,

$$\mu_i = \mathbb{E}_{q_i(x_i)}[x_i] = \sum_{x_i \in [1,-1]} x_i q_i(x_i) = (+1)q_i(x_i = 1) + (-1)q_i(x_i = 1) \tag{201}$$

$$= \frac{1}{1 + e^{-2a_i}} - \frac{1}{1 + e^{2a_i}} = \frac{e^{a_i}}{e^{a_i} + e^{-a_i}} - \frac{e^{-a_i}}{e^{-a_i} + e^{a_i}} \tag{202}$$

$$= \frac{e^{a_i} - e^{-a_i}}{e^{a_i} + e^{-a_i}} = \tanh(a_i) = \tanh\left(m_i + \frac{1}{2}(L_i(1) - L_i(-1))\right). \tag{203}$$

So now we are there, we have the approximative distribution for the mean field approximation of an Ising model and we have equation that tells us how to update the parameters of this distribution. Now before we implement this, let us see if it makes sense.

$$q_i(x_i = 1|\mu_i) = \text{sigm}(2a_i) = \text{sigm}\left(2(m_i + \frac{1}{2}(L_i(1) - L_i(-1)))\right) \tag{204}$$

$$\mu_i = \tanh(a_i) = \tanh\left(m_i + \frac{1}{2}(L_i(1) - L_i(-1))\right) \tag{205}$$

$$m_i = \sum_{j \in \mathcal{N}(i)} w_{ij}\mu_j \tag{206}$$

We are in effect trying to find the probability of a latent variable that can take two different values, this means that a sigmoid function makes perfect sense as an approximative distribution. The update of the variational parameter μ_i is updated as a $\tanh(2ai)$ function this also makes sense as we have $x_i \in [-1, 1]$ we now have an updated which is bounded between those two values. Below we have plotted the two functions,

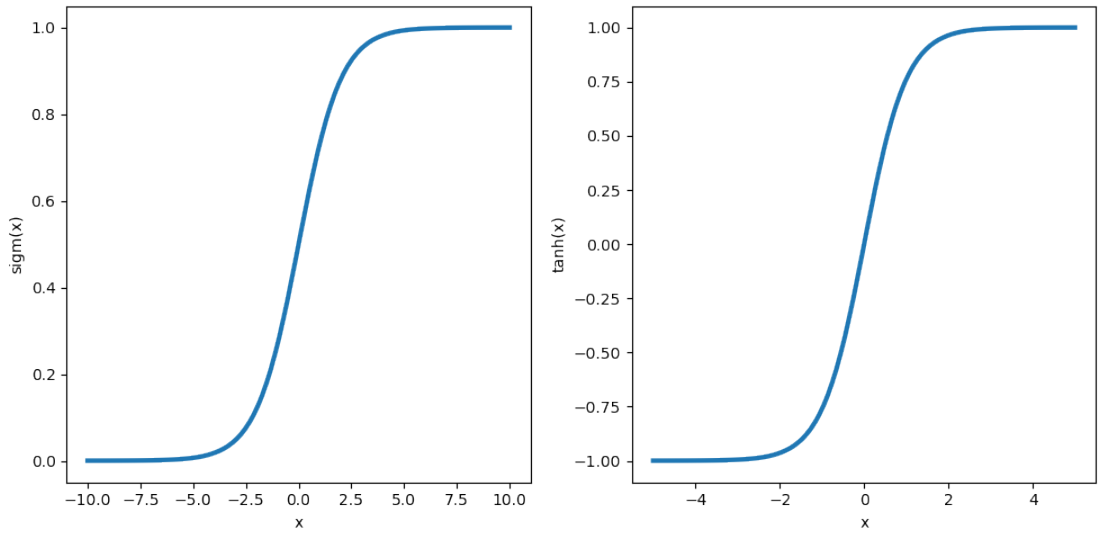

Figure 18: *The above plot shows on the left a sigmoid function and on the right a hyperbolicus tangent function. They look very similar but observe that the sigmoid has its asymptots at 0 and 1 while the tanh is at -1 and 1.*

Algorithm 5 Variational Bayes for Ising Model

procedure MEAN FIELD VARIATIONAL BAYES
  $\boldsymbol{\mu} \leftarrow$ initialise variational distributions
  $\boldsymbol{x} \leftarrow$ initialise latent variables
  for $\tau = 1 \dots T$ **do**
    for $i = 1 \dots N$ **do**
      $m_i^{\tau+1} = \sum_{j \in \mathcal{N}(i)} w_{ij} \mu_j^{\tau} \leftarrow$ compute parameter
      $\mu_i^{\tau+1} = \tanh\left(m_i + \frac{1}{2}(L_i(1) - L_i(-1))\right) \leftarrow$ update variational parameter
  return $q(\mathbf{x})$

Except for the asymptotic vaules, does the posterior and the variational update make sense, both are functions of a_i as,

$$a_i = m_i + \frac{1}{2}(L_i(1) - L_i(-1)) = \sum_{j \in \mathcal{N}(i)} w_{ij}\mu_j + \frac{1}{2}(L_i(1) - L_i(-1)).$$

The first term in the above expression relates to the nodes that are neighbours of i. In effect it is a weighting of the expected values of the posteriors of these nodes where the weights w_{ij} are what encodes our prior assumption in how neighbours interact. As the weights w_{ij} are all positive if all neighbours are in agreement, i.e. have the same sign, then the first term will "push" towards either -1 or $+1$. For simplicity lets assume that all neighours have $\mu_j = 1$ then m_i will be a positive value and vice versa. If it is close to zero that means that the neighbours are all in disagreement **or** that we are very uncertain of their values, i.e. μ_j is close to zero. The second term is a difference between that comes from the likelihood terms, if it is positive this means that it is much more likely that $x_i = 1$ describes the data y_i compared to $x_i = -1$. So for the extremes, if all neighbours are $\mu_j = 1$ and $L_i(1) - L_i(-1) > 0$ then a_i will be a large positive value, i.e. the posterior $q_i(x_i)$ will be large and μ_i will get a value close to one. So these equations do make sense.

9.4 Implementation

Now we are ready to finally implement our variational Bayes inference scheme. What we will do is to start off with some initial value for our variational parameter and then we will update each of the distributions in turn. You can try and start with different values and see how it changes the way we reach a solution. The algorithm we should implement is outlined in Algorithm 5.

9.5 Summary

This lab implemented a deterministic approximation to approximate inference. Importantly, with this approach we will never reach the true solution, but importantly we will get to an approximate solution really quickly. Also because we now have a parametrised form of our posterior we can do lots of interesting things with it as it is just like any other distribution we can just treat it as it is the true posterior. Variational methods are really useful in practice but they do take a bit more effort to formulate compared to sampling and I believe that this task gave you a good example of this.

References

[1] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian Processes for Data-Efficient Learning in Robotics and Control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, stat.ML(2):408–423, February 2015.

[2] M. Kaiser, C. Otte, T. Runkler, and C. H. Ek. Bayesian alignments of warped multi-output gaussian processes. In *Advances in Neural Information Processing Systems 32, [NIPS Conference, Montreal, Quebec, Canada, December 3 - December 8, 2018]*, 2018.

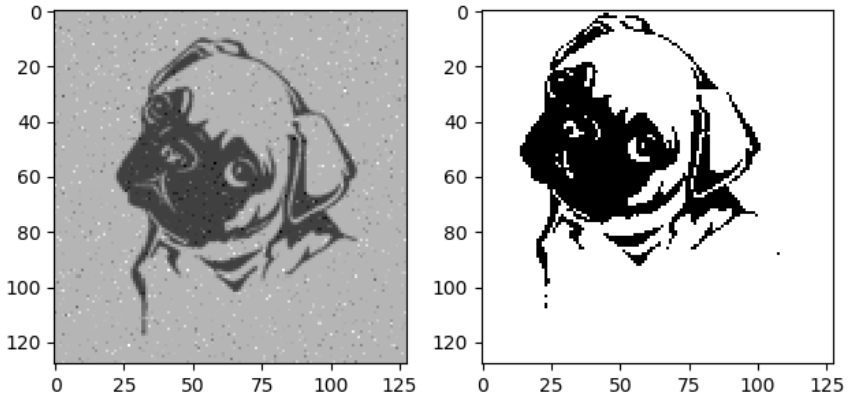

Figure 19: *This figure shows the result of my implementation of variational inference for the image denoising example. As you can see the ising prior cleans up the image rather nicely and we are left with a lovely black and white pug*

[3] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[4] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. Bayesian optimization in alphago. *CoRR*, 2018.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] Neil D Lawrence. Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *Journal of Machine Learning Research*, 6:1783–1816, 2005.

[7] Keith Grochow, Steven L Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *SIGGRAPH '04: SIGGRAPH 2004 Papers*, August 2004.

[8] Raquel Urtasun, David J Fleet, and P. Fua. 3D people tracking with Gaussian process dynamical models. *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, 1:238–245, 2006.

[9] Steindór Sæmundsson, Katja Hofmann, and Marc Peter Deisenroth. Meta reinforcement learning with latent variable gaussian processes. *CoRR*, 2018.

[10] K. B. Petersen and M. S. Pedersen. The Matrix Cookbook, November 2012. Version 20121115.

[11] Iain Murray and Zoubin Ghahramani. A note on the evidence and Bayesian Occam's razor. Technical Report GCNU-TR 2005-003, August 2005.

[12] J. Močkus. On bayesian methods for seeking the extremum. In G. I. Marchuk, editor, *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*, pages 400–404, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.