Week 4 - 5: Bayesian neural networks

Introduction

This week, we will review Bayesian inference and Bayesian neural networks taking into account MCMC methods and probability distributions. We will cover Bayesian logistic regression and Bayesian neural networks where we will use MCMC methods. We note that your textbook does not feature the lesson for this week and hence we have to rely on other materials for further information.

Additional Reading material

1. Andrieu, C., De Freitas, N., Doucet, A., & Jordan, M. I. (2003). An introduction to MCMC for machine learning. *Machine learning*, *50*(1-2), 5-43. https://www.cs.ubc.ca/~arnaud/andrieu_defreitas_doucet_jordan_intromontecarlomachinelearning.pdf
2. http://www.columbia.edu/~mh2078/MachineLearningORFE/MCMC_Bayes.pdf
3. Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)* (pp. 681-688). http://people.ee.duke.edu/~lcarin/398_icmlpaper.pdf
4. Neal, R. M. (2011). MCMC using Hamiltonian dynamics. *Handbook of markov chain monte carlo*, *2*(11), 2. https://arxiv.org/pdf/1206.1901.pdf%20http://arxiv.org/abs/1206.1901.pdf
5. Chandra, R., & Simmons, J. (2024). Bayesian neural networks via MCMC: a Python-based tutorial. *IEEE Access: https://ieeexplore.ieee.org/abstract/document/10530647*

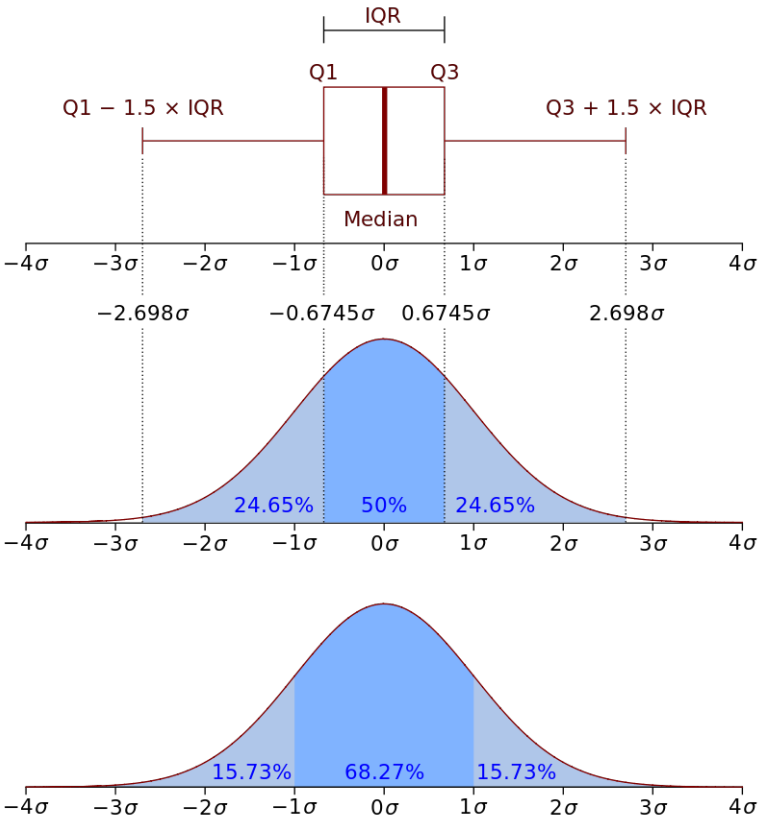
Programming support

1. https://www.r-tutor.com/elementary-statistics/probability-distributions
2. https://web.stanford.edu/class/archive/cs/cs109/cs109.1198/handouts/pythonForProbability.html

Probability distribution

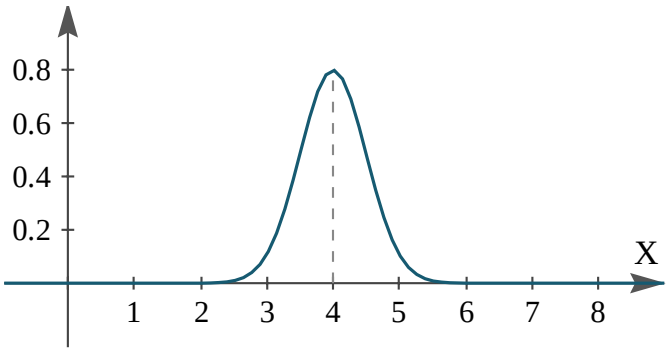
Gaussian (Normal) Distribution

A normal probability density or distribution can be visualised as follows where Q1, Q2 and Q3 refer to respective quartiles and IQR refers to the inter-quantile range.

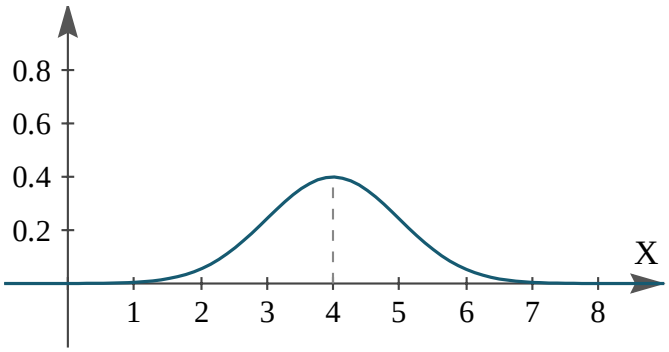


Source: https://en.wikipedia.org/wiki/Probability_density_function

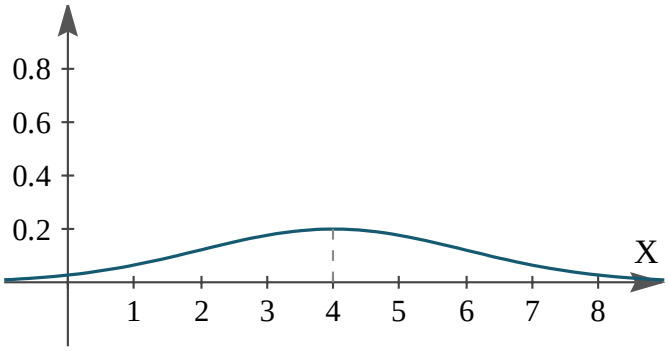
Let us visualise what happens when standard deviation (std) changes and mean remains the same for the a distribution.



mean = 4, std = 0.5



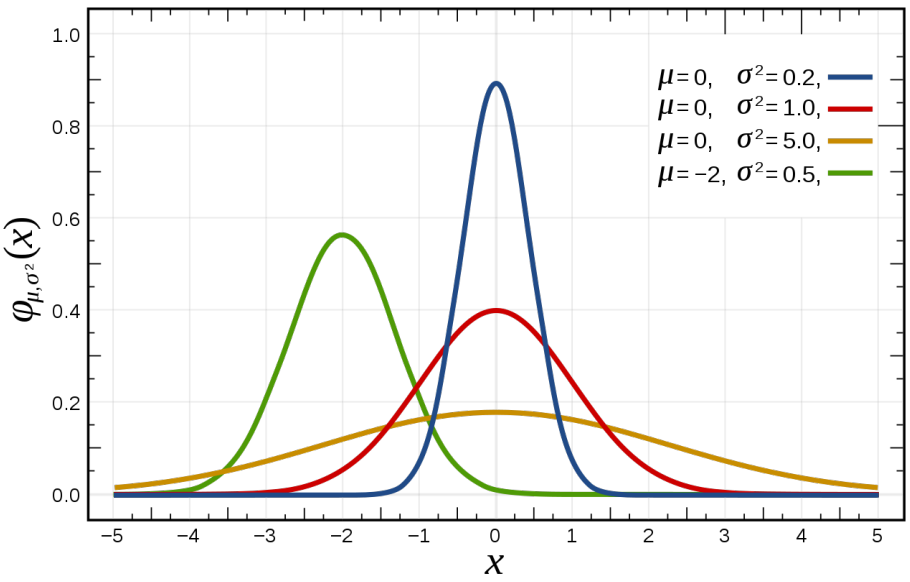
mean = 4, std = 1



mean = 4, std = 2

Source:https://www.intmath.com/counting-probability/11-probability-distributions-concepts.php

We see some more examples where changes to the mean and standard deviation gives us different shapes of the probability density function (PDF).



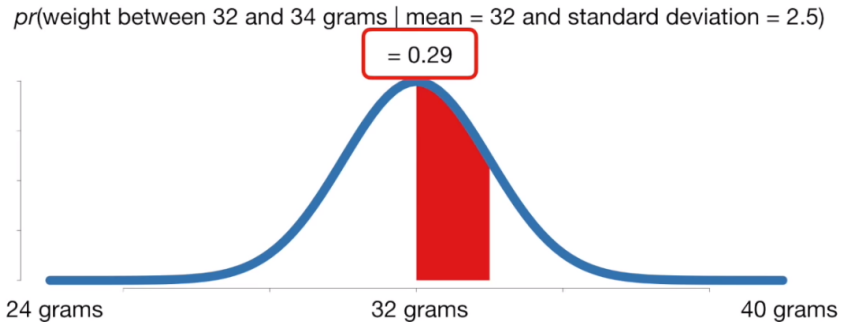
The equation for Gaussian of Normal PDF taking mean μ and standard deviation σ is given below:

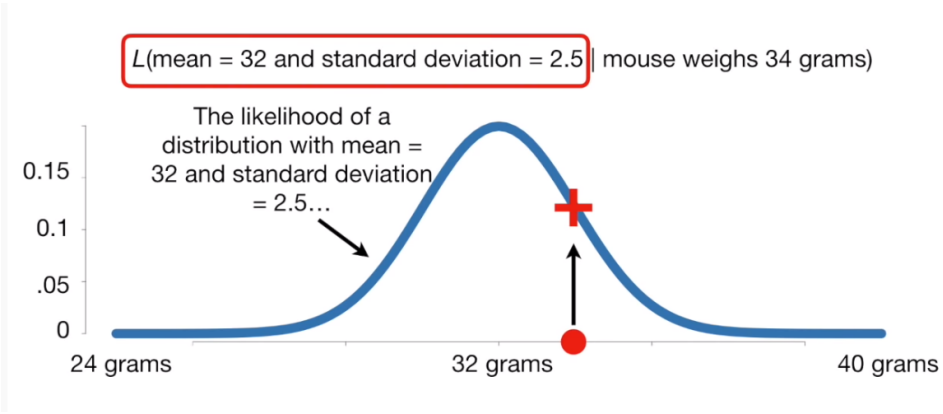
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

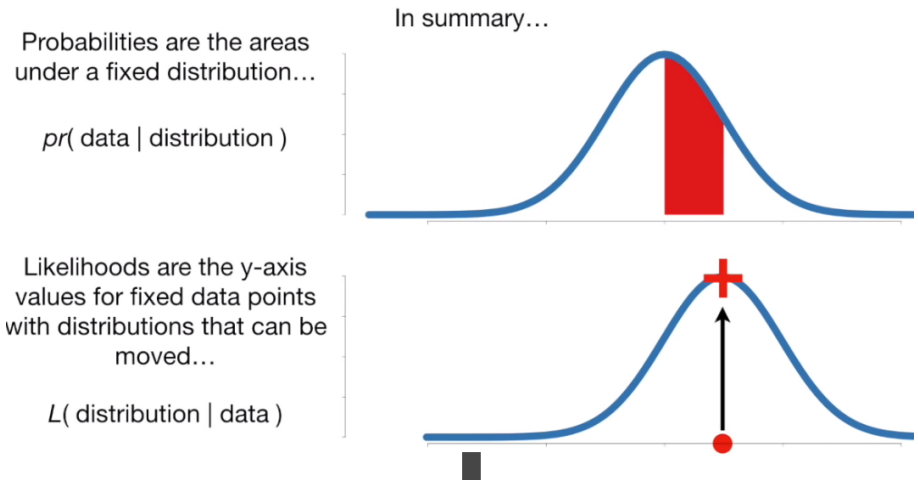
We note that σ^2 is the variance.

More information: https://en.wikipedia.org/wiki/Normal_distribution

Note that probability and likelihood are not the same in the field of statistics, while in everyday language they are used as if they are the same.







Source: Josh Starmer, "Probability is not Likelihood. Find out why!!!": https://www.youtube.com/watch?v=pYxNSUDSFH4&t=17s

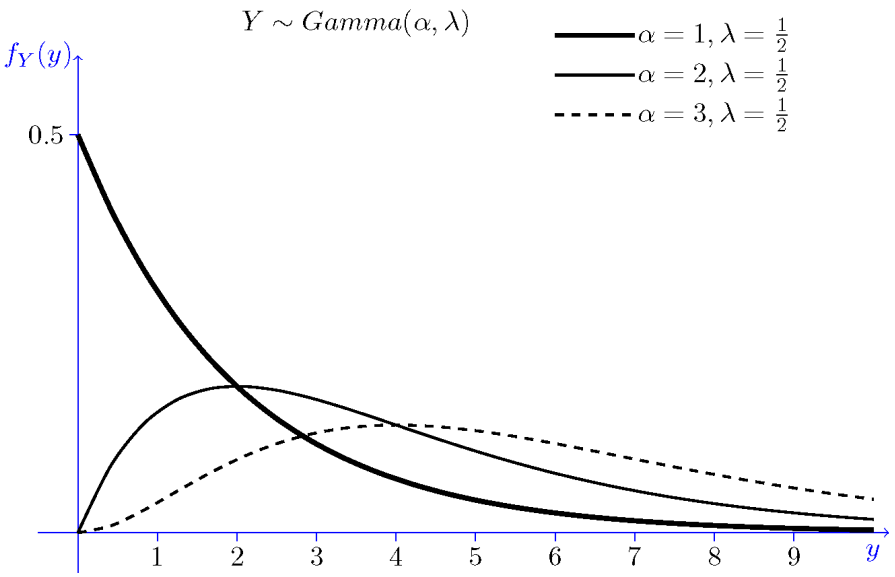
Gamma distribution

A continuous random variable x is said to have a *gamma* distribution with parameters α and β as shown below.

$$f(x;\alpha,\beta) = \frac{\beta^{\alpha} x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}$$

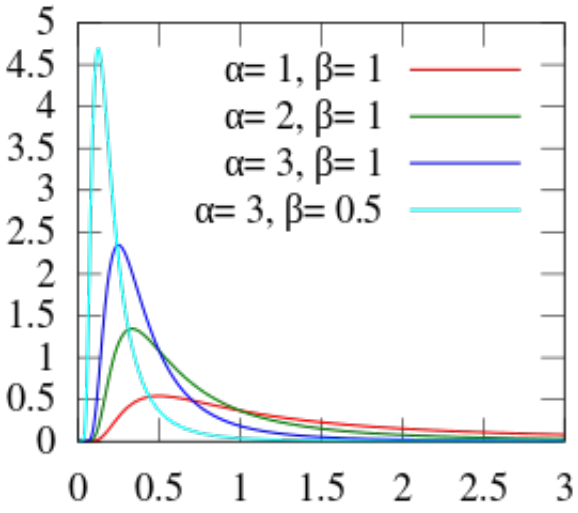
for $x > 0 \quad \alpha, \beta > 0$

where $\Gamma(n) = (n-1)!$



Note in figure above β is λ

Image source: https://en.wikipedia.org/wiki/Gamma_distribution



The **inverse-Gamma** distribution is given above: Source: https://en.wikipedia.org/wiki/Inverse-gamma_distribution

We review random number generation using Python:

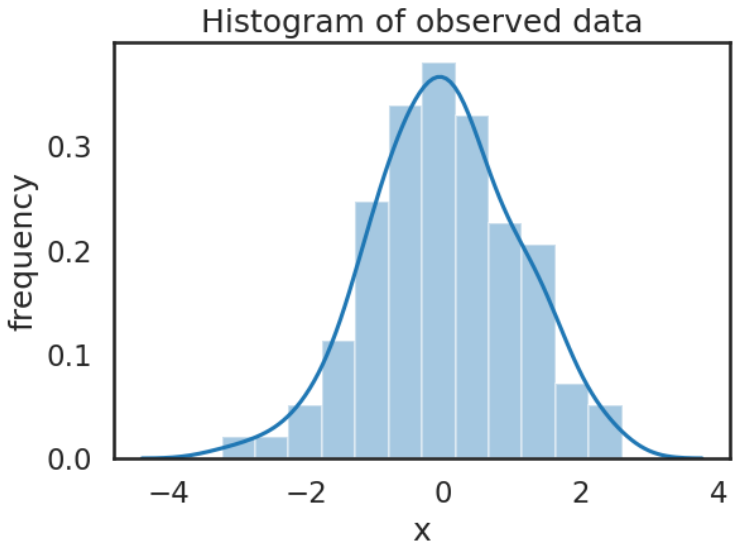
```
from numpy import random
#https://www.datacamp.com/community/tutorials/numpy-random
#https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.randint.html
#https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html
x = random.randint(100)
print(x, ' random.randint(100) ')

x = random.rand()
print(x, ' random.rand() ')

x=random.randint(10, size=(4))
print(x, ' x=random.randint(10, size=(4))')

x = random.randint(100, size=(3, 5))
print(x, ' random.randint(100, size=(3, 5))')

x = random.rand(5)
print(x, ' random.rand(5)')

x = random.rand(3, 4)
print(x, ' random.rand(3, 4)')


#https://numpy.org/doc/stable/reference/random/generated/numpy.random.binomial.html
# result of flipping a coin 10 times, tested 100 times.
n, p = 5, .5  # number of trials, probability of each trial
s = random.binomial(n, p, 10)
print(s, ' random.binomial(n, p, 10)')

#https://numpy.org/doc/stable/reference/random/generated/numpy.random.binomial.html
#"A real world example. A company drills 9 wild-cat oil exploration wells,
# each with an estimated probability of success of 0.1. All nine wells fail.
# What is the probability of that happening?"
#Let’s do 200 trials of the model, and count the number that generate zero positive results.
trial = random.binomial(9, 0.1, 200)
trialsum = sum(trial == 0)/200

print(trial, ' trials')
print(trialsum, ' is trialsum')


#Gamma distribution https://numpy.org/doc/stable/reference/random/generated/numpy.random.gamma.html
scale, size= 5, 50
gammagen= random.gamma(scale, size=size)
```

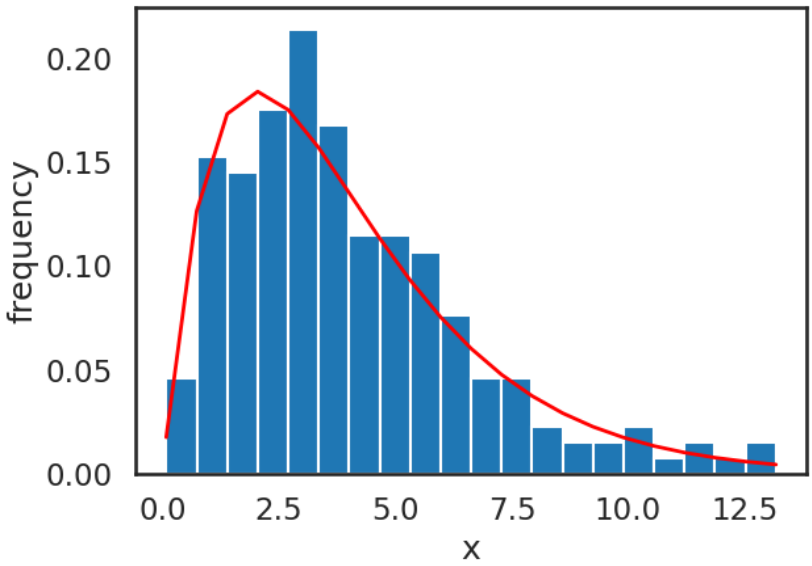
```
print(gammagen, ' gamma gen')


# using scipy https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html
#https://docs.scipy.org/doc/scipy/reference/tutorial/stats.html
from scipy.stats import norm
data_normal = norm.rvs(size=5,loc=0,scale=1)
print(data_normal, ' data normal')
```

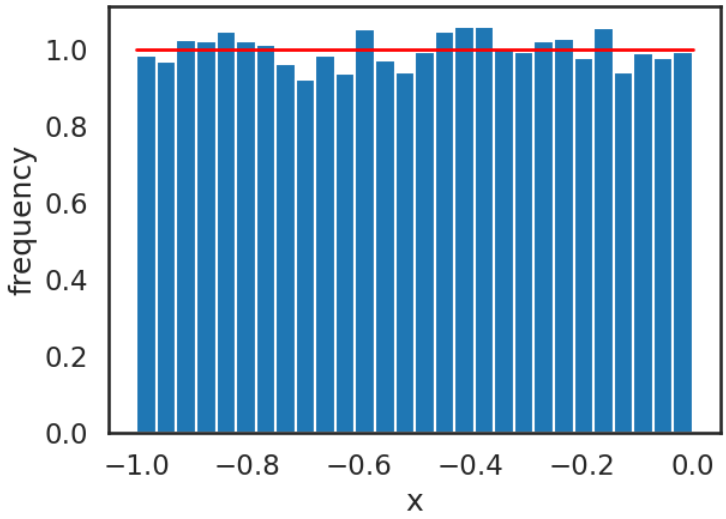
Next, we use Seaborn and Matplotlib python library:

```
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from scipy.stats import norm

sns.set_style('white')
sns.set_context('talk')

np.random.seed(123)

data = np.random.randn(200)

#histogram using seaborn (sns)
ax = plt.subplot()
sns.distplot(data, kde=True, ax=ax) # smooth using KDE: https://en.wikipedia.org/wiki/Kernel_densit
_ = ax.set(title='Histogram of observed data', xlabel='x', ylabel='frequency');
plt.tight_layout()
plt.savefig('histo_seaborn.png')
plt.clf()

#plot using matplotlib
# An "interface" to matplotlib.axes.Axes.hist() method
n, bins, patches = plt.hist(x=data, bins='auto', color='#0504aa', alpha=0.7, rwidth=0.85)
plt.grid(axis='y', alpha=0.75)
plt.xlabel('x')
plt.ylabel('frequency')
plt.tight_layout()
plt.savefig('histo_matplotlib.png')
plt.clf()

 #Gamma distribution https://numpy.org/doc/stable/reference/random/generated/numpy.random.gamma.htm
shape, scale = 2., 2.  # mean=2, std=2
s_gamma = np.random.gamma(shape, scale, 200)
#print(s)
```

```
import scipy.special as sps
count, bins, ignored = plt.hist(s_gamma, 20, density=True)
y = bins**(shape-1)*(np.exp(-bins/scale) / (sps.gamma(shape)*scale**shape))
plt.xlabel('x')
plt.ylabel('frequency')
plt.plot(bins, y, linewidth=2, color='r')
plt.savefig('gamma_dist.png')
plt.clf()

#Uniform distribution https://numpy.org/doc/stable/reference/random/generated/numpy.random.uniform.
s_uniform = np.random.uniform(-1,0,20000) # -1 and 0 give limits

count, bins, ignored = plt.hist(s_uniform, bins='auto', density=True)
plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
plt.xlabel('x')
plt.ylabel('frequency')
plt.savefig('uniform_dist.png')
plt.clf()
```







The above figure shows output given by Normal (top), Gamma(middle), and Uniform (bottom) distribution.

Now some examples in R

```
#Source: https://www.cyclismo.org/tutorial/R/probability.html
dnorm(0)
dnorm(0,mean=4)
dnorm(0,mean=4,sd=10)

v <- c(0,1,2)
dnorm(v)

x <- seq(-2,2,by=.1)
print(x)
y <- dnorm(x)
plot(x,y)
```

Multivariate Normal distribution

The multivariate normal distribution or joint normal distribution generalises univariate normal distribution to more variables or higher dimensions.

$$f_{\mathbf{x}}\left(x_1, \ldots, x_k\right) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k|\boldsymbol{\Sigma}|}}$$

where x is a real "k"-dimensional column vector and |Σ| is the determinant of symmetric covariance

matrix Σ which is positive definite. Multivariate normal distribution reducves to univariate normal distribution if Σ is a single real number.

Bivariate case

$$f(x,y) = \frac{1}{2\pi\sigma_X\,\sigma_Y\sqrt{1-\rho^2}}\,\exp\left(-\frac{1}{2(1-\rho^2)}\left[\frac{(x-\mu_X)^2}{\sigma_X^2} + \frac{(y-\mu_Y)^2}{\sigma_Y^2} - \frac{2\rho(x-\mu_X)(y-\right.\right.$$

where

ρ is the correlation between X and Y, given $\sigma_X > 0$ and $\sigma_Y > 0$.

$$\boldsymbol{\mu} = \begin{pmatrix}\mu_X \\ \mu_Y\end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix}\sigma_X^2 & \rho\sigma_X\,\sigma_Y \\ \rho\sigma_X\,\sigma_Y & \sigma_Y^2\end{pmatrix}.$$

More information: https://en.wikipedia.org/wiki/Multivariate_normal_distribution

Note that the covariance matrix gives the covariance between each pair of elements, where the diagonal represents the variance, which gives the covariance of each element with itself.

```
#Implementation: https://peterroelants.github.io/posts/multivariate-normal-primer/
def multivariate_normal(x, d, mean, covariance):
    """pdf of the multivariate normal distribution."""
    x_m = x - mean
    return (1. / (np.sqrt((2 * np.pi)**d * np.linalg.det(covariance))) *
            np.exp(-(np.linalg.solve(covariance, x_m).T.dot(x_m)) / 2))
```

```
#https://stackoverflow.com/questions/11615664/multivariate-normal-density-in-python
from numpy import *
import math
# covariance matrix
sigma = matrix([[2.3, 0, 0, 0],
           [0, 1.5, 0, 0],
           [0, 0, 1.7, 0],
           [0, 0,   0, 2]
          ])
# mean vector
mu = array([2,3,8,10])

# input
x = array([2.1,3.5,8, 9.5])

def norm_pdf_multivariate(x, mu, sigma):
    size = len(x)
    if size == len(mu) and (size, size) == sigma.shape:
```

```
        det = linalg.det(sigma)
        if det == 0:
            raise NameError("The covariance matrix can't be singular")

        norm_const = 1.0/ ( math.pow((2*pi),float(size)/2) * math.pow(det,1.0/2) )
        x_mu = matrix(x - mu)
        inv = sigma.I
        result = math.pow(math.e, -0.5 * (x_mu * inv * x_mu.T))
        return norm_const * result
    else:
        raise NameError("The dimensions of the input don't match")

print(norm_pdf_multivariate(x, mu, sigma))
```

```
                           # Install MASS package
library("MASS")                                       # Load MASS package
#https://statisticsglobe.com/bivariate-multivariate-normal-distribution-in-r



my_n2 <- 1000                                         # Specify sample size
my_mu2 <- c(5, 2, 8)                                  # Specify the means of the variables
my_Sigma2 <- matrix(c(10, 5, 2, 3, 7, 1, 1, 8, 3),  # Specify the covariance matrix of the variable
                    ncol = 3)
mvrnorm(n = my_n2, mu = my_mu2, Sigma = my_Sigma2)  # Random sample from bivariate normal distribut
```

Bernoulli distribution

Bernoulli distribution is a discrete probability distribution which takes the value 1 with probability p and the value 0 with probability $q = 1 - p$ which can be used for modelling binary classification problems. Hence, the probability mass function for this distribution over k possible outcomes is given by

$$f(k;p) = p^k(1-p)^{1-k} \quad \text{for } k \in 0,1$$

Binomial distribution

The probability of getting exactly "k" successes in "n" independent Bernoulli trials is given by

$$f(k,n,p) = \Pr(k;n,p) = \Pr(X = k) = \binom{n}{k}p^k(1-p)^{n-k}$$

for $k = 0, 1, 2, ..., n$, where

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

```
from scipy.stats import binom
# setting the values
# of n and p
n = 6
p = 0.6
# defining the list of r values
r_values = list(range(n + 1))
print(r_values, ' r_values')
# obtaining the mean and variance
mean, var = binom.stats(n, p)
# list of pmf values
dist = [binom.pmf(r, n, p) for r in r_values ]
# printing the table
print("r\tp(r)")
for i in range(n + 1):
    print(str(r_values[i]) + "\t" + str(dist[i]))
# printing mean and variance
print("mean = "+str(mean))
print("variance = "+str(var))
```

Multinomial distribution

We consider an experiment of extracting *n* balls of *k* different colours from a bag and replacing the extracted ball after each draw. The balls of the same colour are equivalent. The number of extracted balls of colour *i* (*i* = 1, ..., *k*) as X_i, and denote as p_i the probability that a given extraction will be in color i.

$$\begin{aligned} f(x_1,\ldots,x_k;n,p_1,\ldots,p_k) &= \Pr(X_1 = x_1 \text{ and } \ldots \text{ and } X_k = x_k) \\ &= \begin{cases} \dfrac{n!}{x_1!\cdots x_k!}p_1^{x_1} \times \cdots \times p_k^{x_k}, & \text{when } \sum_{i=1}^{k} x_i = n \\ 0 & \text{otherwise,} \end{cases} \end{aligned}$$

More information: https://en.wikipedia.org/wiki/Multinomial_distribution

```
#https://numpy.org/doc/stable/reference/random/generated/numpy.random.multinomial.html

import numpy as np

draw = np.random.multinomial(20, [1/6.]*6, size=1)
print(draw, ' first draw')
#array([[4, 1, 7, 5, 2, 1]]) # random
#It landed 4 times on 1, once on 2, etc.
```

```
#Now, throw the dice 20 times, and 20 times again:
draw = np.random.multinomial(20, [1/6.]*6, size=2)
print(draw, ' second  draw')
#array([[3, 4, 3, 3, 4, 3], # random
     #  [2, 4, 3, 4, 0, 7]])
```

More info: http://users.umiacs.umd.edu/~jbg/teaching/INST_414/04c.pdf

Supplementary material

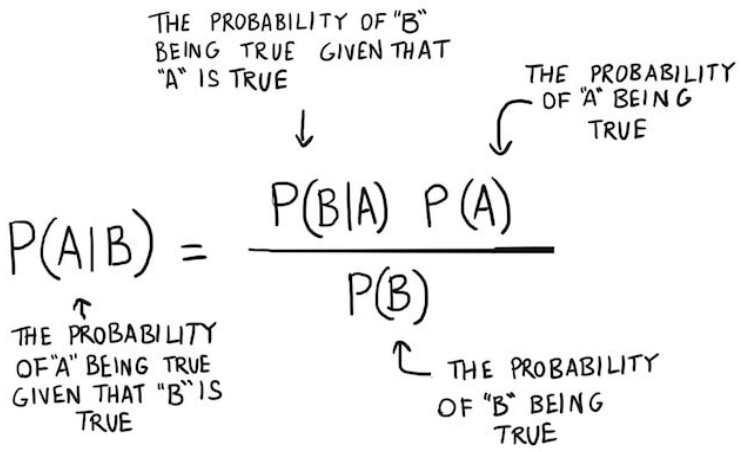
1. Josh Starmer, " Maximum Likelihood, clearly explained!!!", https://www.youtube.com/watch?v=XepXtl9YKwc&t=245s
2. Josh Starmer, "Probability is not Likelihood. Find out why!!!", https://www.youtube.com/watch?v=pYxNSUDSFH4&t=17s

Bayesian inference

Bayesian methods account for the uncertainty in prediction and decision making via the posterior distribution. Note that the posterior is the conditional probability determined after taking into account the prior distribution and the relevant evidence or data via sampling methods.

Bayesian methods can account for the uncertainty in parameters (weights) and topology by marginalisation over the predictive posterior distribution.

Hence, as opposed to conventional neural networks, Bayesian neural learning use probability distributions to represent the weights



Thomas Bayes is the guy behind Bayes' theorem which is the foundation of Bayesian inference.

Image Source: https://en.wikipedia.org/wiki/Thomas_Bayes

Let's review it again

Conditional Probability

- The probability of A given B:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \text{ if } P(B) > 0$$

- Multiplication rule:

$$P(A \cap B) = P(A)P(B|A)$$

- Bayes law:

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)}$$

Key probability rules are given here: http://www.milefoot.com/math/stat/prob-rules.htm

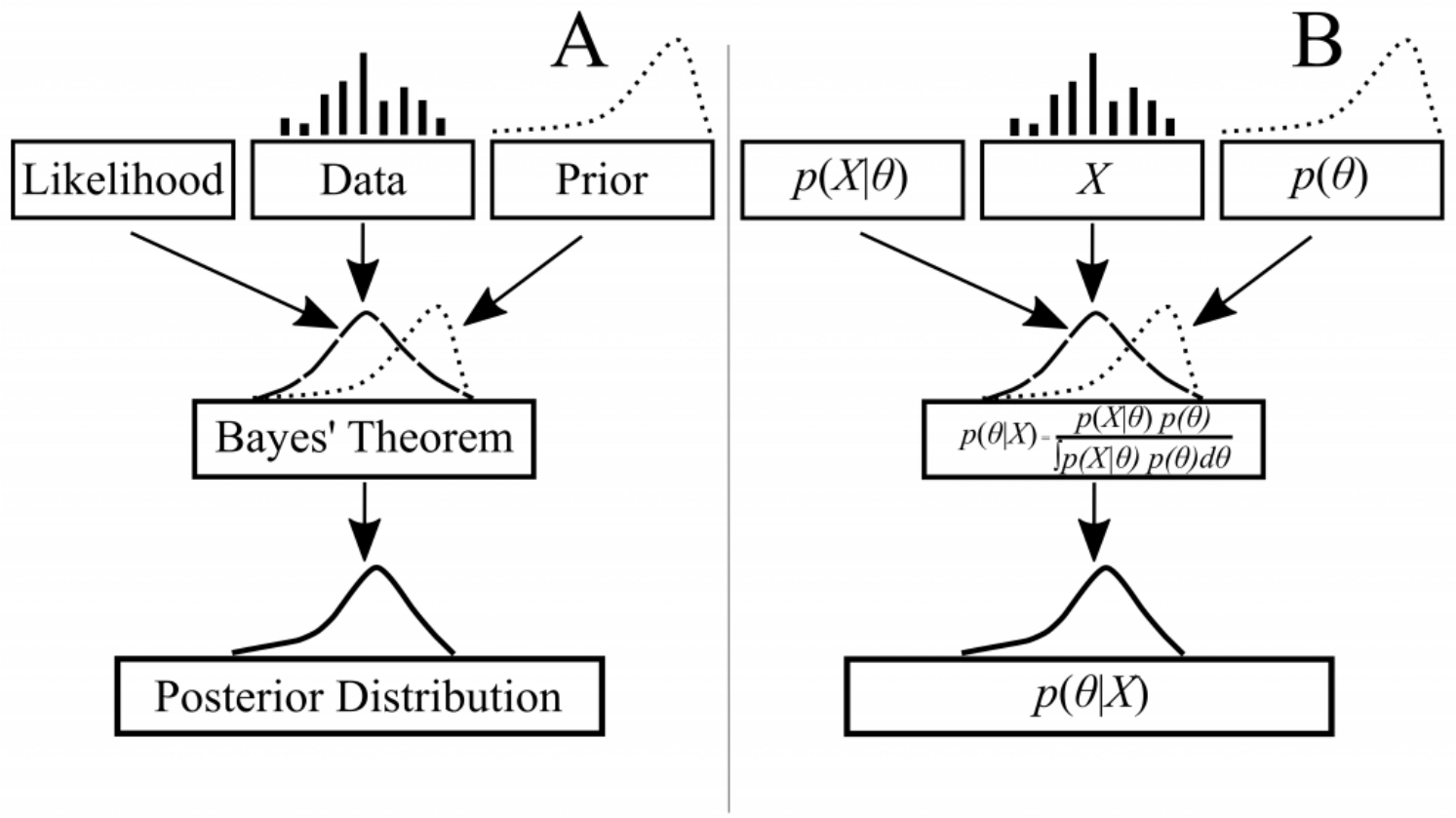


Figure Source: TBA

The figure above gives an overview of the Bayesian inference framework that uses data with prior and likelihood to construct or sample from the posterior distribution. This is the building block of the rest of the lessons that will feature Bayesian logistic regression and Bayesian neural networks.

Essentially, Bayesian inference refers to a principled way of estimating unknown variables using prior information or belief about the variable. Prior information is captured in the form of a distribution. A simple example of a prior belief is a distribution that has a positive real-valued number in some range, this essentially would imply that our result or posterior distribution would likely be a distribution of positive numbers in some range which would be similar to the prior but not the same.

If the posterior and prior are both same, this is known as conjugate priors and if the prior is helpful, it is known as informative prior.

Here is a nice overview of the history of the field.

Supplementary material

1. Bayesian vs Frequentist: https://stats.stackexchange.com/questions/22/bayesian-and-frequentist-reasoning-in-plain-english
2. Bayesian vs Frequentist video: https://www.youtube.com/watch?v=meivbbfHmK0

MCMC sampling

Bayesian inference

The need for efficient sampling methods to implement Bayesian inference has been the focus of research in computational statistics, especially for the case of multi-modal and irregular posterior distributions. Bayesian inference is typically implemented by **Markov Chain Monte Carlo (MCMC)** sampling methods which are used to update the probability for a hypothesis (proposal Θ) as more information becomes available. The hypothesis is given by a prior probability distribution that expresses one's belief about a quantity (or free parameter in a model) before some data is taken into account. MCMC methods enable to samples from a distribution iteratively using **proposal distribution**, **prior distribution** $P(\Theta)$ and a **likelihood function** to construct the **posterior distribution** $P(\Theta|data)$.

$$P(\Theta|\mathbf{data}) = \frac{\mathbf{P(d|\Theta)} \times \mathbf{P(\Theta)}}{\mathbf{P(d)}}$$

We note that $P(data|\Theta)$ could be seen as the **likelihood distribution** in disguise. $P(data)$ *is the* marginal distribution of the data and is often seen as a normalising constant and ignored. Hence, ignoring it, we can also express the above in this way

$$P(\Theta|\mathbf{data}) \propto \mathbf{P(data|\Theta)} \times \mathbf{P(\Theta)}$$

The likelihood function is a function of the parameters of a given model provided specific observed data. The likelihood function can be seen as a fitness measure of the proposals which are drawn from the proposal distribution.

The posterior distribution is constructed after taking into account the relevant evidence (data) and prior distribution with the likelihood that considers the proposal and the model. MCMC methods essentially implement Bayesian inference via a numerical approach that marginalize or integrate over the posterior distribution.

MCMC sampling

MCMC methods have seen much success in many applications, such as machine learning,

astrophysics, geoscientific inversions, Earth and environmental sciences, and any application that uses some form of model over data.

We note that a Markov process is uniquely defined by its transition probabilities $P(x' \mid x)$ which defines the probability of transitioning from any given state x to other given state x'. The Markov process has a unique stationary distribution $\pi(x)$ given the following two conditions are met.

- There must be the existence of stationary distribution given by the sufficient detailed balance condition that requires that each transition $x \rightarrow x'$ is reversible. This implies that for every pair of states x, x', the probability of being in state x and moving to state x' must be equal to the probability of being in state x' and moving to state x, hence,

$$\pi(x)P(x' \mid x) = \pi(x')P(x \mid x').$$

  *More information: http://prob140.org/sp17/textbook/ch14/Detailed_Balance.html

- The stationary distribution must be unique which is guaranteed by ergodicity of the Markov process. Ergodicity is guaranteed when every state is aperiodic where the system does not return to the same state at fixed intervals, and when every state is positive recurrent where the expected number of steps for returning to the same state is finite. In other words, an ergodic system is one that mixes well, i.e. you get the same result whether you average its values over time or over space.

*More information: Grazzini, J., 2012. Analysis of the emergent properties: Stationarity and ergodicity. *Journal of Artificial Societies and Social Simulation*, *15*(2), p.7. http://jasss.soc.surrey.ac.uk/15/2/7.html

Given that $\pi(x)$ is chosen to be $P(x)$, the condition of detailed balance becomes

$$P(x' \mid x)P(x) = P(x \mid x')P(x')$$

which is re-written as

$$\frac{P(x'|x)}{P(x|x')} = \frac{P(x')}{P(x)}$$

Here is a basic MCMC algorithm that samples till max_samples is reached for training data, $\mathbf{D}$.

for $i=1 **until** max_samples

1. Propose a value $x'|x_i \sim q(x_i)$, where $q(.)$ is the proposal distribution.
2. Given x', execute model $f(x', \mathbf{D})$ and compute the predictions (output y) and the log-liklihood
3. Calculate the acceptance probability
   $\alpha = \min\left(1, \frac{P(x')}{P(x_i)} \frac{q(x_i|x')}{q(x'|x_i)}\right)$

4. Generate from a uniform distribution $u \sim U(0, 1)$

if $\alpha < u$

accept by setting $x_i = x'$

else

reject by setting $x_i = x_{i-1}$

The algorithm above proceeds by proposing new values of the parameter (Step 1) from the selected proposal distribution, which is random-walk (multivariate) normal distribution $q(.)$ with user-defined mean (generally 0) and the step-size (standard deviation) ϕ or covariance matrix Σ. Conditional on these proposed values, the model $f(x', \mathbf{D})$ computes or predicts an output using proposal x' and data $\mathbf{D}$ (Step 2).

Using the predictions, the likelihood is computed, and then the Metropolis-Hasting criterion is used for determining whether to accept or reject the proposal (Step 3). If the proposal is accepted, the chain moves to this proposed value. If rejected, the chain stays at the current value (Step 4). The process is repeated until the convergence criterion is met, which in this case is the maximum number of samples (max_samples) defined by the user. We note that the proposal distribution can use gradients if available but the acceptance criterion will slightly change.

Below is a framework that gives an overview of MCMC for a simple data-driven model such as neural network or logistic regression.

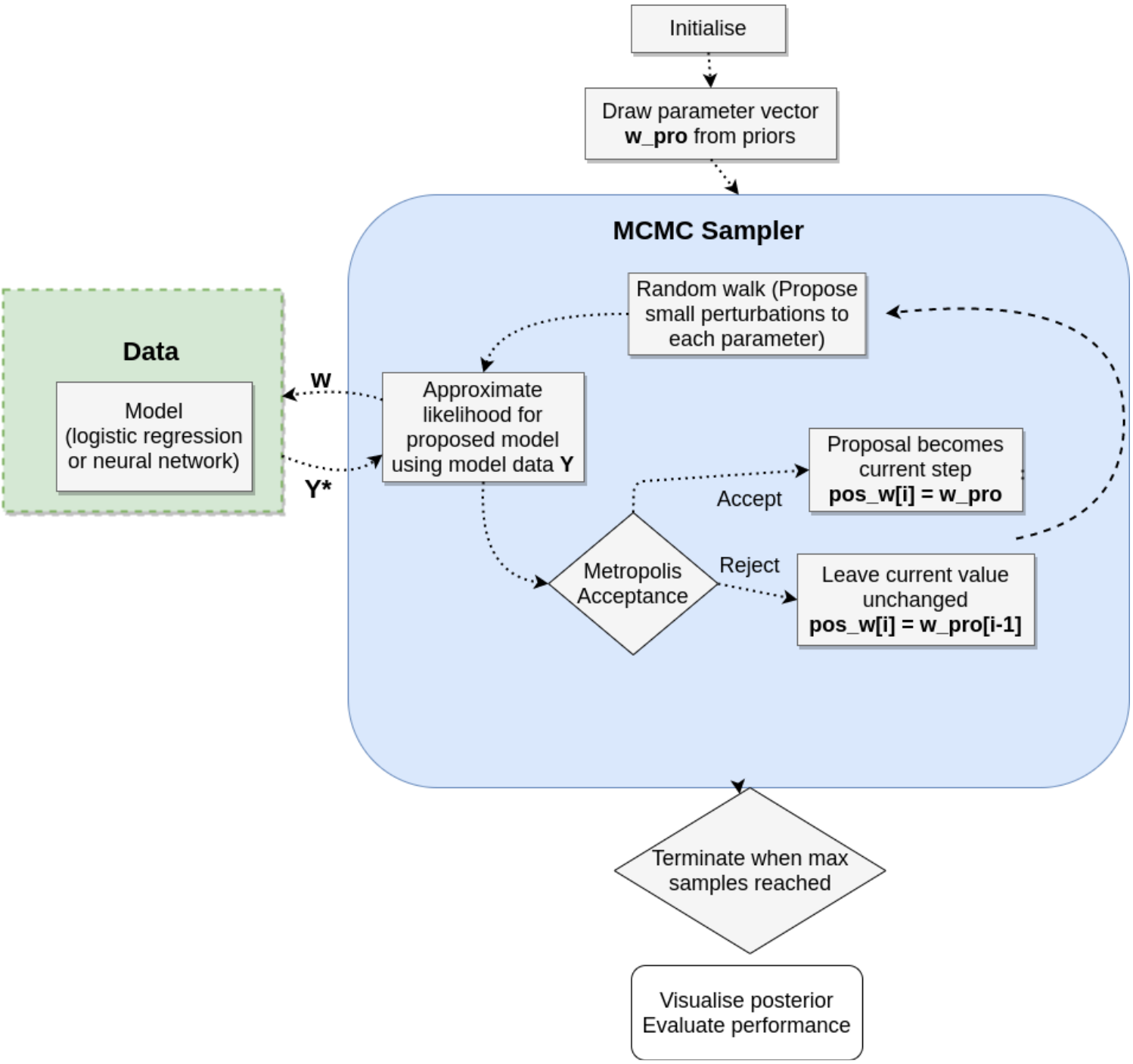
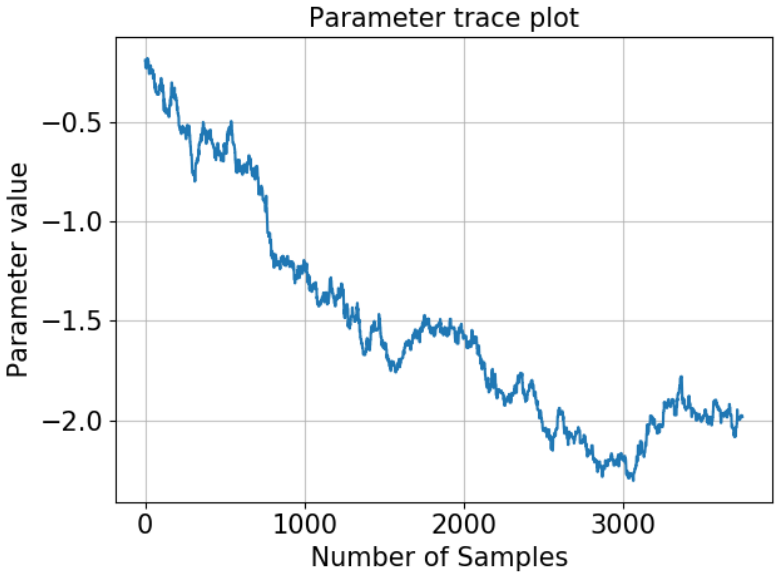
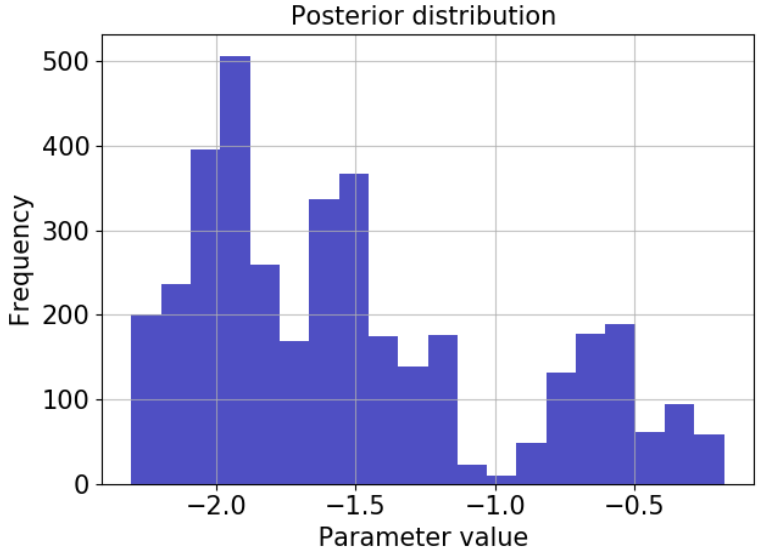
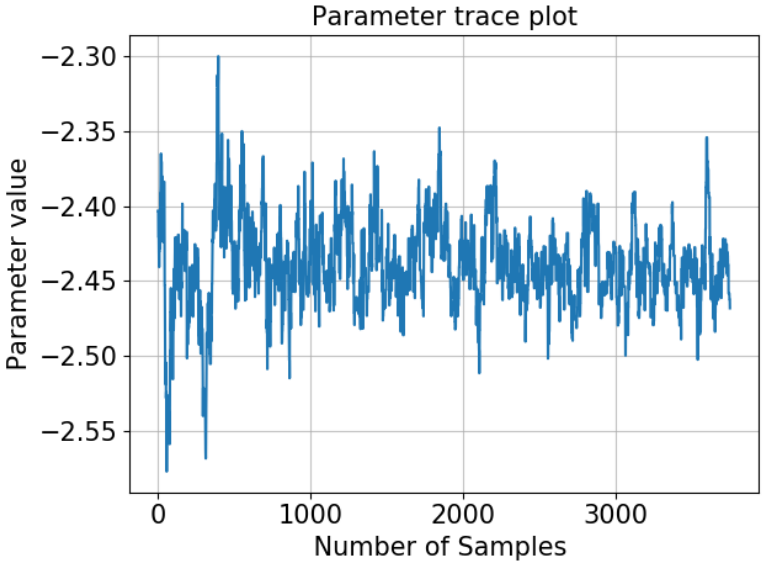


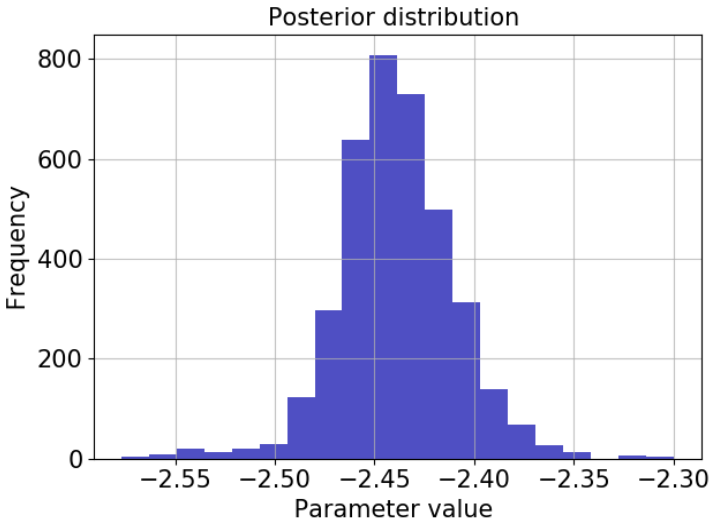
Figure Source: Edited by R. Chandra based on: Chandra R; Azam D; Müller RD; Salles T; Cripps S, 2019, 'Bayeslands: A Bayesian inference approach for parameter uncertainty quantification in Badlands', *Computers and Geosciences*, vol. 131, pp. 89 - 101, http://dx.doi.org/10.1016/j.cageo.2019.06.012

Below is an example of trace-plot and posterior for two selected variables in Bayesian logistic regression with MCMC.

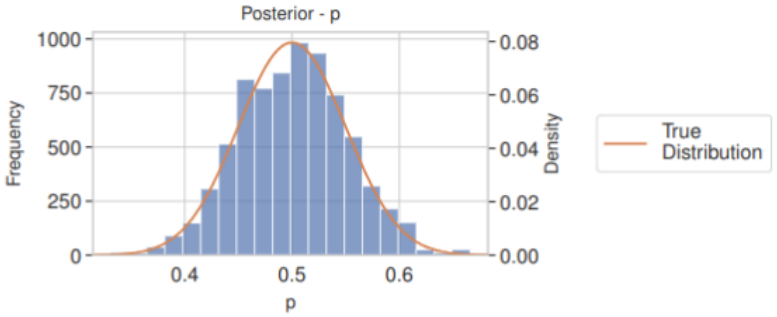

Parameter trace plot
Parameter value
−0.5
−1.0
−1.5
−2.0
0
1000
2000
3000
Number of Samples


Posterior distribution
Frequency
500
400
300
200
100
0
−2.0
−1.5
−1.0
−0.5
Parameter value

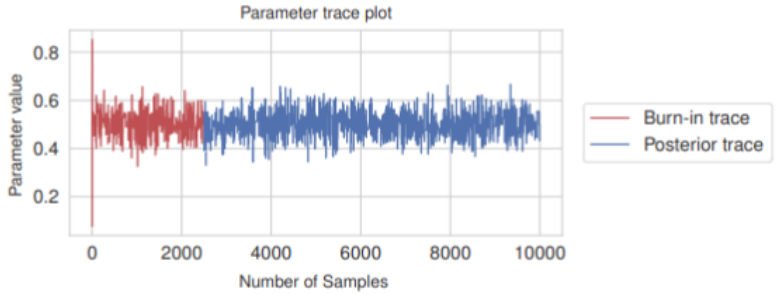

Parameter trace plot
Parameter value
−2.30
−2.35
−2.40
−2.45
−2.50
−2.55
0
1000
2000
3000
Number of Samples



Burn-in samples:



(a) Posterior distribution



(b) Trace plot

Lesson is based on: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647

Below is a video that can shed more light on MCMC sampling. Note they may not show results using R or Python, but in the coming lessons, we will get into more details with them.

Supplementary material

Additional notes about MCMC is here:

1. http://phylo.bio.ku.edu/slides/BayesianMCMC-2013.pdf
2. http://www.southampton.ac.uk/~sks/utrecht/mcmc.pdf
3. https://jellis18.github.io/post/2018-01-02-mcmc-part1/

Video on Erodicity: https://www.youtube.com/watch?v=1Vxe3LBykRI

Video on Detailed balance: https://www.youtube.com/watch?v=Bg7gaJzzPN0

Priors and likelihood derivation

Preliminaries

Log rules

Logarithm product rule

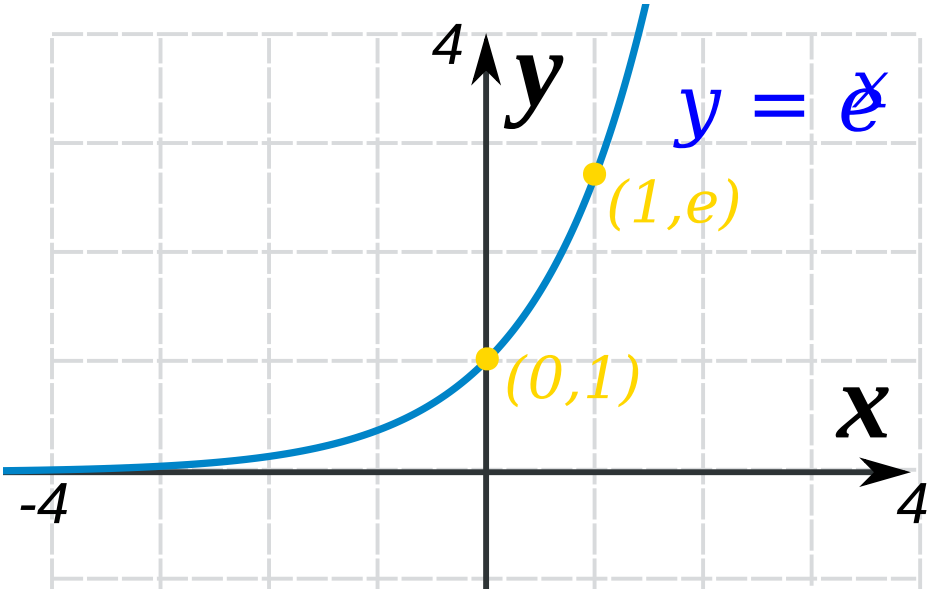
$$log_b(x \times y) = log_b(x) + log_b(y)$$

Logarithm quotient rule

$$log_b(x/y) = log_b(x) - log_b(y)$$

More infor: https://www.rapidtables.com/math/algebra/Logarithm.html

Log vs exp



Source: https://www.mathsisfun.com/algebra/exponents-logarithms.html

Log-likelihood function

It is more convenient to maximize the log of the likelihood function since the logarithm is monotonically increasing function of its argument, maximization of the log of a function is equivalent to maximization of the function itself.

The log-likelihood simplifies the subsequent mathematical analysis and also helps avoid numerical instabilities due to the product of a large number of small probabilities. In the log-likelihood, this is

resolved naturally by computing the sum of the log probabilities.

Given you have a set of cases

$X = \{x_1, x_2, \ldots, x_N\}$

The total likelihood would be the product of likelihood for each case

$p(X \mid \Theta) = \prod_{i=1}^{N} p(x_i \mid \Theta)$

where Θ

represents the parameters of the model (logistic regression/neural network).

$\ln p(X \mid \Theta) = \sum_{i=1}^{N} \ln p(x_i \mid \Theta)$

Our likelihood is built using the Gaussian distribution taking into account, variable x, mean u and standard deviation σ

$$g\left(x\right) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{\left(x-\mu\right)^2}{\sigma^2}\right)$$

Taking into account a model (logistic regression or neural network), we can construct the likelihood function using a set of weights and biases $\boldsymbol{\theta}$ for M network parameters, for S training instances.

We wish to model the dependency between a vector k of inputs, $\mathbf{x} = (x_1, \ldots, x_k)'$ and an output y. We assume that the relationship between inputs and outputs is a signal plus noise model where the signal depends upon a set of parameters θ and is denoted by $f(\mathbf{x}, \theta)$. The noise is assumed to be Gaussian with a mean of zero and a variance of τ^2, so that

$$y = f(\mathbf{x}, \theta) + e,\ e \sim N(0, \tau^2)$$

If we observed single sample of outputs, $\mathbf{y} = (y_1, \ldots, y_T)$ and corresponding inputs, $X = (\mathbf{x}_1, \ldots, \mathbf{x}_T)'$, then the likelihood is given by

$$p(\mathbf{y}|X, \boldsymbol{\theta}) = \frac{1}{(2\pi\tau^2)^{S/2}} \times \exp\left(-\frac{1}{2\tau^2}\sum_{t=1}^{S}\left(y_t - f(\mathbf{x}_t, \theta)\right)^2\right)$$

where y_S is the predictions with S samples, τ and θ are proposed values by the proposal distribution, $f(\bar{\mathbf{x}}_\mathbf{t}, \theta)$ is the model (logistic regression).

Here θ (represents all weights and biases) and τ (represents single noise parameter) are the parameters, L is number of parameters in the model, and σ is the standard deviation.

Hence, we will only look at examples that use log-likelihood for the rest of the lessons.

More information:

1. http://cs229.stanford.edu/section/gaussians.pdf
2. https://math.stackexchange.com/questions/892832/why-we-consider-log-likelihood-instead-of-likelihood-in-gaussian-distribution

Priors

The prior is typically information you have without looking at the data and considering only the model topology. The information can be based on past experiments or information regarding the posterior distribution of the model for related datasets.

An ***informative prior*** would give specific and definite information about a variable. If we consider the prior distribution for the temperature tomorrow evening, it would be reasonable to use a normal distribution with an expected value (as mean) of today evenings temperature with a standard deviation of the temperature during evening time for the entire season.

A ***weakly informative prior*** expresses partial information about a variable. In the case of the prior distribution of evening temperature, a weakly informative prior would consider day time temperature of the day (as mean) with a standard deviation of day time temperature for the whole year.

An **uninformative prior** or **diffuse prior** expresses vague about a variable such as the variable is positive or has some limit range. Typically uniform distribution can be used for uninformative prior.

If the case when the prior distribution comes from the same probability distribution family as the posterior distribution, the prior and posterior are then called conjugate distributions. The prior is called a **conjugate prior** for the likelihood function of the Bayesian model.

In case that the prior is based on the normal distribution for a logistic regression problem, we would need the user defined hypermeters, i.e mean and standard deviation. Essentially, these user defined values would be placed in the prior function that will also consider the number of model parameters.

$$p(\boldsymbol{\theta}) \propto \frac{1}{(2\pi\sigma^2)^{L/2}} \times \exp\left\{ -\frac{1}{2\sigma^2}\left(\sum_{i=1}^{M}\theta^2\right)\right\} \times \tau^{-2(1+\nu_1)}\exp\left(\frac{-\nu_2}{\tau^2}\right)$$

where θ (represents all weights and biases) and τ (represents single noise parameter) are the parameters, L is the number of parameters in the model, and σ is the standard deviation. ν_1 and ν_2 are user-defined parameters, which are 0 in our case. Note that the mean is 0.

Next, we show how we can use MCMC sampling as an alternative way to train the above model. Training here is also known as sampling and the number of training iterations (epochs) is known as samples in the MCMC game. Yes, this is the game of thrones via distributions!

Code jump into MCMC!

We give details of implementing **Bayesian linear regression** that uses Metropolis-Hastings MCMC with Random-Walk proposal distribution. Consider the equations and code for logistic regression model below.

The likelihood is given by

$$p(\mathbf{y_S}|\boldsymbol{\theta}) = -\frac{1}{(2\pi\tau^2)^{S/2}} \times \exp\left(-\frac{1}{2\tau^2}\sum_{t\in S}(\mathbf{y_t} - f(\bar{\mathbf{x}}_\mathbf{t}))^2\right)$$

Our log-likelihood is given by

$$logp(\mathbf{y} \mid \mathbf{x}, \theta, \tau^2) = -log((2\pi\tau^2)^{S/2}) - \frac{1}{2\tau^2}\sum_{t=1}^{S}(\mathbf{y}_t - f(\mathbf{x}_t, \theta))^2$$

We note we have two priors. The prior is given by 1. multivariate Gaussian (for weights and biases) and 2. inverse Gamma for τ^2. Lets revisit multivariate normal distribution.

$$f_{\mathbf{X}}(x_1, \ldots, x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k|\boldsymbol{\Sigma}|}}$$

Suppose that our $\mathbf{x}$ is our set of weights and biases, θ. Suppose our μ is a vector of zeros, then we get

$$f_{\mathbf{X}}(\theta_1, \ldots, \theta_k) = \frac{\exp\left(-\frac{1}{2}(\boldsymbol{\theta})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta})\right)}{\sqrt{(2\pi)^k|\boldsymbol{\Sigma}|}}$$

As the covariance matrix in this case is just a diagonal matrix with all values equal to σ^2 (scalar), so $\boldsymbol{\Sigma}^{-1}$ will become $\mathbf{I}/\sigma^2$ where I is an identity matrix (diagonal elements which are all 1s).

Hence, the numerator in above equation

$$(\boldsymbol{\theta})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta})$$

becomes

$$\frac{(\boldsymbol{\theta})^{\mathrm{T}}\mathbf{I}(\boldsymbol{\theta})}{\sigma^2}$$

We note that multiplying identity matrix with any other matrix is the matrix itself, hence finally we get $\boldsymbol{\theta}^2$ in numerator.

Now we see inverse gamma distribution

$$f(\tau^2;\nu_1,\nu_2) = \frac{\nu_1^{\nu_2}}{\Gamma(\nu_1)}\,(1/\tau^2)^{\nu_1+1}\exp\left(-\nu_2/\tau^2\right)$$

where ν_1 and ν_2 are parameters for inverge gamma. We note that the front part $\frac{\nu_1^{\nu_2}}{\Gamma(\nu_1)}$ is a constant which is dropped considering propotionality. Hence we have

$$p(\boldsymbol{\theta}) \propto \frac{1}{(2\pi\sigma^2)^{L/2}} \times \exp\left\{-\frac{1}{2\sigma^2}\left(\sum_{i=1}^{M}\theta_i^2\right)\right\} \times \tau^{-2(1+\nu_1)}\exp\left(\frac{-\nu_2}{\tau^2}\right)$$

Our log-prior is given by

$$\log p(\theta) \propto -\frac{M}{2}\log 2\pi\sigma^2 - \frac{1}{2\sigma^2}\left(\sum_{i=1}^{M}\theta^2\right) - (1+\nu_1)\log\tau^2 - \frac{\nu_2}{\tau^2}$$

Note the code below does not run. We will run them all together in the next session with data.

```
class MCMC:
    def __init__(self, samples, traindata, testdata, topology, regression):
                self.samples = samples  # NN topology [input, hidden, output]
                self.topology = topology  # max epocs
                self.traindata = traindata  #
                self.testdata = testdata
                random.seed()
                self.regression = regression # False means classification
```

```
        def rmse(self, predictions, targets):
                return np.sqrt(((predictions - targets) ** 2).mean())

        def likelihood_func(self, model, data, w, tausq):
                y = data[:, self.topology[0]]
                fx = model.evaluate_proposal(data, w)
                accuracy = self.rmse(fx, y) #RMSE
                loss = -0.5 * np.log(2 * math.pi * tausq) - 0.5 * np.square(y - fx) / tausq
                return [np.sum(loss), fx, accuracy]

        def prior_likelihood(self, sigma_squared, nu_1, nu_2, w, tausq):
                param = self.topology[0]  + 1 # number of parameters in model
                part1 = -1 * (param / 2) * np.log(sigma_squared)
                part2 = 1 / (2 * sigma_squared) * (sum(np.square(w)))
        multivariate_normal = part1 - part2
        inverse_gamma = -(1 + nu_1) * np.log(tausq) - (nu_2 / tausq)
                log_loss = multivariate_normal + inverge_gamma
                return log_loss
```

The code below is well known from previous lessons. Note that GD and SGD functions have been removed and the code currently does not have an algorithm/method to train the model.

```
 # by R. Chandra
 #https://github.com/rohitash-chandra/Bayesian_logisticregression

import numpy as np
import random
import math
import matplotlib.pyplot as plt
from math import exp

class lin_model:

    def __init__(self, num_epocs, train_data, test_data, num_features, learn_rate, activation):
                self.train_data = train_data
                self.test_data = test_data
                self.num_features = num_features
                self.num_outputs = self.train_data.shape[1] - num_features
                self.num_train = self.train_data.shape[0]
                self.w = np.random.uniform(-0.5, 0.5, num_features)  # in case one output class
                self.b = np.random.uniform(-0.5, 0.5, self.num_outputs)
                self.learn_rate = learn_rate
                self.max_epoch = num_epocs
                self.use_activation = activation #SIGMOID # 1 is  sigmoid , 2 is step, 3 is linear
                self.out_delta = np.zeros(self.num_outputs)
                self.activation = activation

        def activation_func(self,z_vec):
                if self.use_activation == False:
                        y =  1 / (1 + np.exp(z_vec)) # sigmoid/logistic
                else:
```

```
                        y = z_vec
                return y

        def predict(self, x_vec ):
                z_vec = x_vec.dot(self.w) - self.b
                output = self.activation_func(z_vec) # Output
                return output



        def squared_error(self, prediction, actual):
                return  np.sum(np.square(prediction - actual))/prediction.shape[0]# to cater more i

        def encode(self, w): # get  the parameters and encode into the model

                self.w =  w[0:self.num_features]
                self.b = w[self.num_features]

        def evaluate_proposal(self, data, w):  # BP with SGD (Stocastic BP)

                self.encode(w)  # method to encode w and b
                fx = np.zeros(data.shape[0])

                for s in range(0, data.shape[0]):
                                i = s #random.randint(0, data.shape[0]-1)  (we dont shuffle in this
                                input_instance  =  data[i,0:self.num_features]
                                actual  = data[i,self.num_features:]
                                prediction = self.predict(input_instance)
                                fx[s] = prediction

                return fx
```

Before sampling, we need to set up the sampler by generating the first sample and initialising arrays or matrices that will be capturing the posterior distribution, accuracy, and predictions as the sampling happens.

```
def sampler(self):

        # ------------------- initialize MCMC
                testsize = self.testdata.shape[0]
                trainsize = self.traindata.shape[0]
                samples = self.samples

                x_test = np.linspace(0, 1, num=testsize)
                x_train = np.linspace(0, 1, num=trainsize)
```

```
            #self.topology  # [input,   output]
            y_test = self.testdata[:, self.topology[0]]
            y_train = self.traindata[:, self.topology[0]]

            w_size = self.topology[0]  + self.topology[1]  # num of weights and bias (eg. 4 + 1

            pos_w = np.ones((samples, w_size))  # posterior of all weights and bias over all sa
            pos_tau = np.ones((samples, 1))

            fxtrain_samples = np.ones((samples, trainsize))  # fx of train data over all sample
            fxtest_samples = np.ones((samples, testsize))  # fx of test data over all samples
            rmse_train = np.zeros(samples)
            rmse_test = np.zeros(samples)

            w = np.random.randn(w_size)
            w_proposal = np.random.randn(w_size)

            step_w = 0.02;  # defines how much variation you need in changes to w
            step_eta = 0.01;
            # eta is an additional parameter to cater for noise in predictions (Gaussian likeli
            # note eta is used as tau in the sampler to consider log scale.
            # eta is not used in multinomial likelihood.


            model = lin_model(0 ,  self.traindata, self.testdata, self.topology[0], 0.1, self.r

            pred_train = model.evaluate_proposal(self.traindata, w)
            pred_test = model.evaluate_proposal(self.testdata, w)

            eta = np.log(np.var(pred_train - y_train))
            tau_pro = np.exp(eta)

            print('evaluate Initial w')

            sigma_squared = 5  # considered by looking at distribution of  similar trained  mod
            nu_1 = 0
            nu_2 = 0

            prior_likelihood = self.prior_likelihood(sigma_squared, nu_1, nu_2, w, tau_pro)  #

            [likelihood, pred_train, rmsetrain] = self.likelihood_func(model, self.traindata, w

            print(likelihood, ' initial likelihood')
            [likelihood_ignore, pred_test, rmsetest] = self.likelihood_func(model, self.testdat


            naccept = 0
```

The MCMC class has been created, but it has something important missing! That is the sampler! Note that the above code has likelihood functions needed for prediction/regression problem using the Gaussian likelihood. Note that the log-likelihood is used and hence the ratio of previous and current likelihood will need to consider log rules: https://www.rapidtables.com/math/algebra/Logarithm.html

See below the implementation of the sampler - first part below that calls or invokes the logistic regression class and calculates initial likelihood to begin. We are all set for sampling next!

for $i=1 **until** max_samples

1. Propose a value $x'|x_i \sim q(x_i)$, where $q(.)$ is the proposal distribution.
2. Given x', execute model $f(x', \mathbf{D})$ and compute the predictions (output y) and the log-liklihood
3. Calculate the acceptance probability
   $\alpha = \min\left(1, \frac{P(x')}{P(x_i)} \frac{q(x_i|x')}{q(x'|x_i)}\right)$
4. Generate from a uniform distribution $u \sim U(0, 1)$

if $\alpha < u$

accept by setting $x_i = x'$

else

reject by setting $x_i = x_{i-1}$

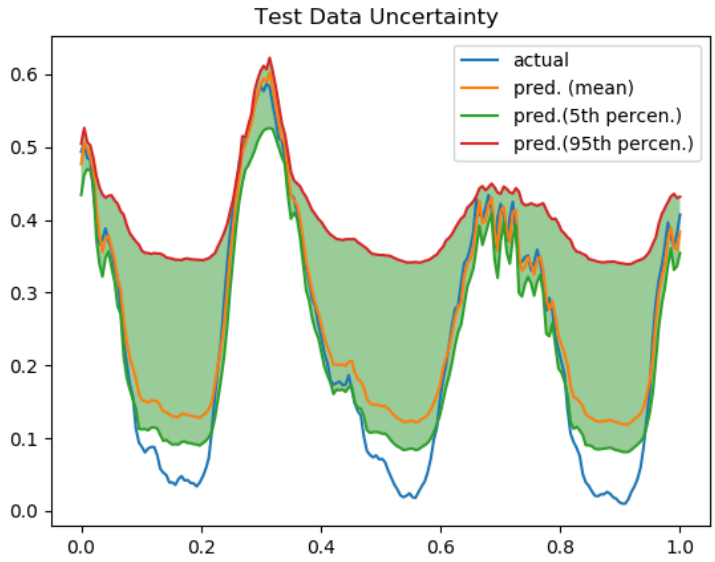
Finally we do sampling as shown below.

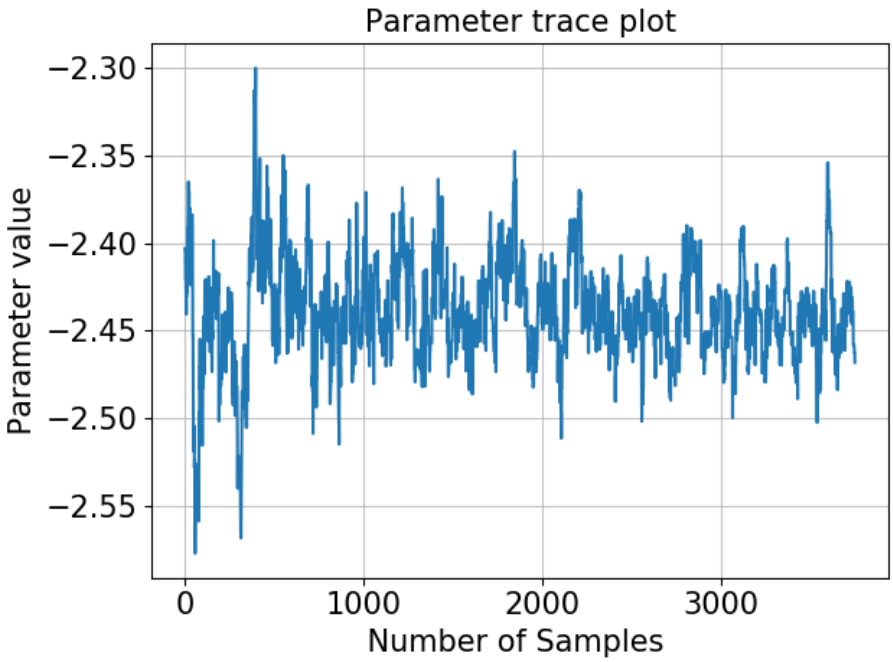
```
for i in range(samples - 1):

                w_proposal = w + np.random.normal(0, step_w, w_size)

                eta_pro = eta + np.random.normal(0, step_eta, 1)
                tau_pro = math.exp(eta_pro)

                [likelihood_proposal, pred_train, rmsetrain] = self.likelihood_func(model,
                [likelihood_ignore, pred_test, rmsetest] = self.likelihood_func(model, self

                # likelihood_ignore  refers to parameter that will not be used in the alg.

                prior_prop = self.prior_likelihood(sigma_squared, nu_1, nu_2, w_proposal, t

                diff_likelihood = likelihood_proposal - likelihood # since we using log sca
                diff_priorliklihood = prior_prop - prior_likelihood

                mh_prob = min(1, math.exp(diff_likelihood + diff_priorliklihood))

                u = random.uniform(0, 1)

                if u < mh_prob:
                        # Update position
                        #print    (i, ' is accepted sample')
                        naccept += 1
                        likelihood = likelihood_proposal
                        prior_likelihood = prior_prop
```

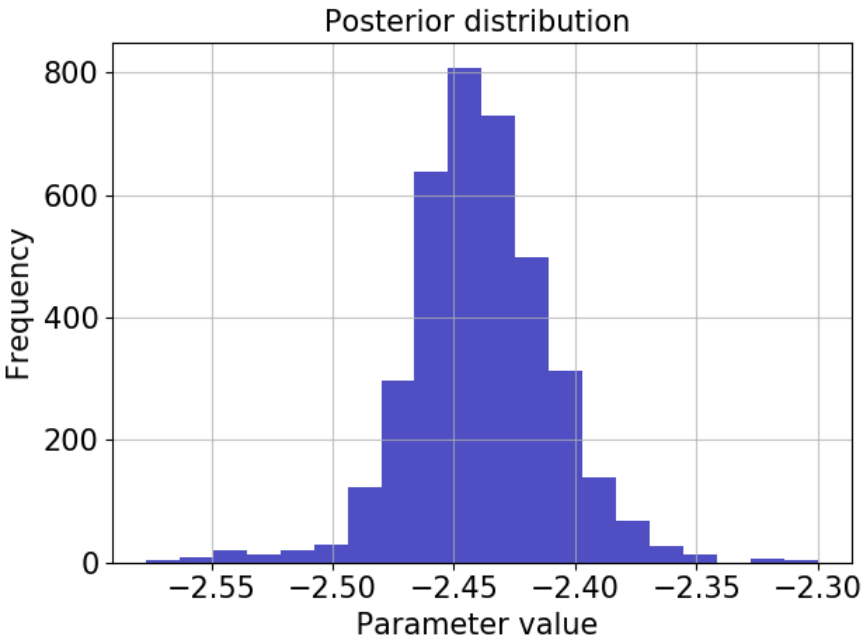
```
                        w = w_proposal
                        eta = eta_pro
                        rmse_train[i + 1,] = rmsetrain
                        rmse_test[i + 1,] = rmsetest


                        #print (likelihood, prior_likelihood, rmsetrain, rmsetest, w, 'acce

                        pos_w[i + 1,] = w_proposal
                        pos_tau[i + 1,] = tau_pro
                        fxtrain_samples[i + 1,] = pred_train
                        fxtest_samples[i + 1,] = pred_test

                else:
                        pos_w[i + 1,] = pos_w[i,]
                        pos_tau[i + 1,] = pos_tau[i,]
                        fxtrain_samples[i + 1,] = fxtrain_samples[i,]
                        fxtest_samples[i + 1,] = fxtest_samples[i,]
                        rmse_train[i + 1,] = rmse_train[i,]
                        rmse_test[i + 1,] = rmse_test[i,]


        accept_ratio = naccept / (samples * 1.0) * 100


        print(accept_ratio, '% was accepted')

        burnin = 0.25 * samples  # use post burn in samples

        pos_w = pos_w[int(burnin):, ]
        pos_tau = pos_tau[int(burnin):, ]
        rmse_train = rmse_train[int(burnin):]
        rmse_test = rmse_test[int(burnin):]


        rmse_tr = np.mean(rmse_train)
        rmsetr_std = np.std(rmse_train)
        rmse_tes = np.mean(rmse_test)
        rmsetest_std = np.std(rmse_test)
        print(rmse_tr, rmsetr_std, rmse_tes, rmsetest_std, ' rmse_tr, rmsetr_std, rmse_tes,
```

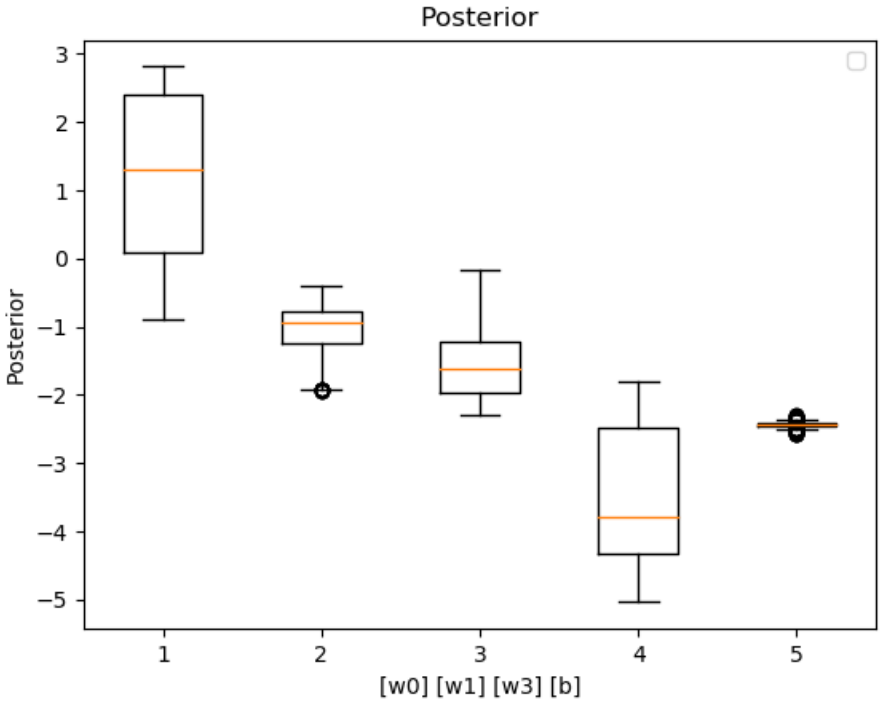
Burn-in: Note that in MCMC, a certain portion of the initial samples is discarded. The discarded samples are known as the burn-in period. At times, the burn-in can be 25 %, at times 50 % depending on the complexity of the model. If you use MCMC for large neural network architectures, 50 % burn-in would be required. Note that burn-in could be seen as the optimisation stage! Essentially you are discarding material that is not part of the posterior distribution. Your posterior distribution should feature good predictions, and that is what you get after your sampler goes towards convergence.

After the code runs, we will be able to see prediction plots from the trained logistic regression model.









Lesson is based on: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647

Bayesian Linear Regression

Bayesian linear regression for a single step ahead (eg. Sunspot time series). After running the code, you can see trace-plot and histogram of the posterior distribution.

Exercise 5.1

R Challenge

- Apply Bayesian logistic/linear regression for one-step-ahead COVID-19 prediction - prediction for China/Australia/USA/India.
- Try extending the R Logistic Regression code by using regression problems from UCI machine learning repository for Bayesian logistic and linear regression.

Python Challenge

- Apply Bayesian logistic/linear regression for one-step-ahead COVID-19 prediction - prediction for China/Australia/USA/India.
- Logistic Regression code by using regression problems from UCI machine learning repository for Bayesian logistic and linear regression.

Exercise 5.1 Solution

R Challenge

- Apply Bayesian logistic/linear regression for one-step-ahead COVID-19 prediction - prediction for Chine/Australia/USA/India.
- Try extending the R Logistic Regression code by using regression problems from UCI machine learning repository for Bayesian logistic and linear regression.

Python Challenge

- Apply Bayesian logistic/linear regression for one-step-ahead COVID-19 prediction - prediction for Chine/Australia/USA/India.
- Apply existing code by using regression problems from UCI machine learning repository for Bayesian logistic and linear regression.
- Apply existing code for multiple outputs (heating and cooling load) for the energy dataset.

Bayesian Neural Networks

A Bayesian neural network is essentially a probabilistic implementation of a standard neural network with the key difference being that the weights and biases are represented via the posterior probability distributions rather than single point values as shown in the figure below.

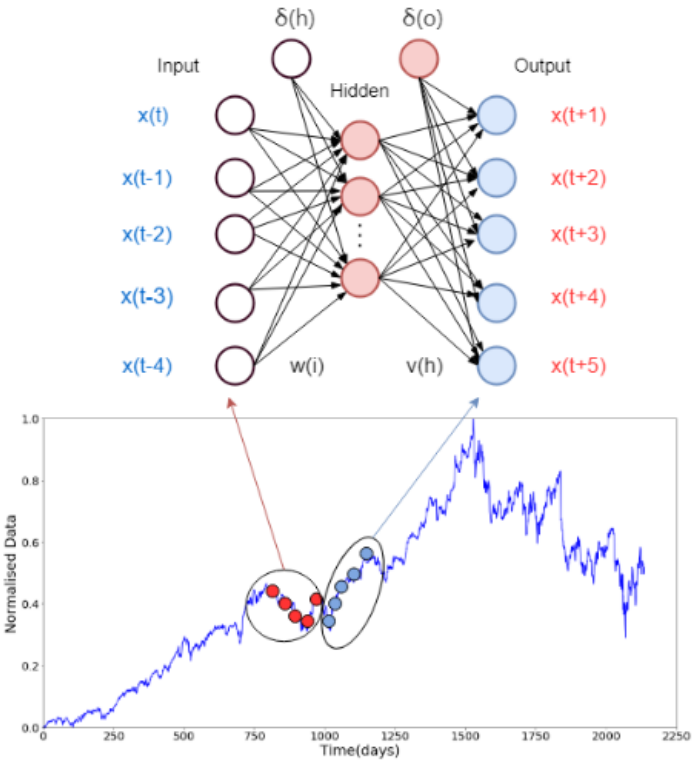
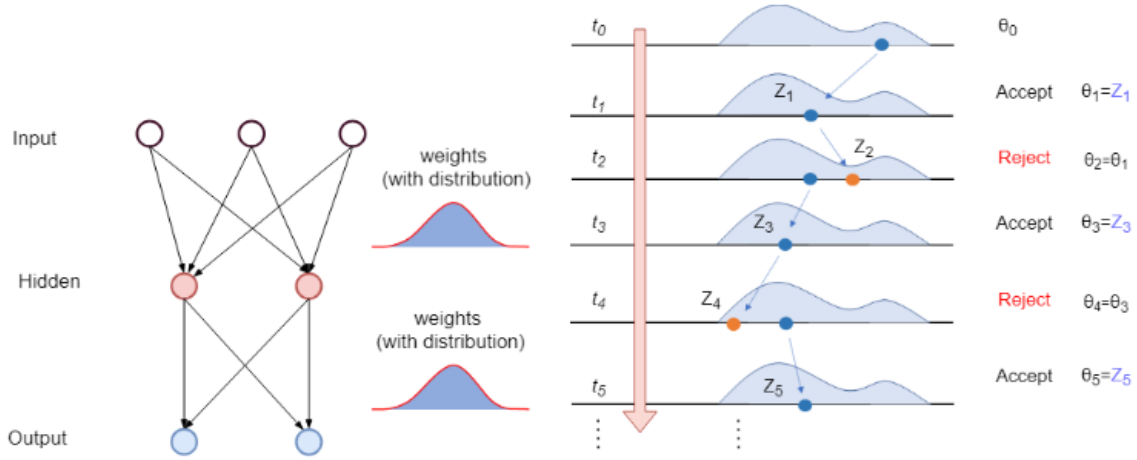
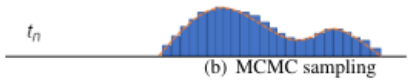


Figure 1: The time series (shown in red circles) is used as input for the neural network which predicts 5 steps-ahead in time (shown by blue circles). A sliding window approach is used to reconstruct the dataset in this way using Taken's theorem.





(a) Bayesian neural network (b) MCMC sampling

Figure 2: Bayesian neural network and MCMC sampling. Note that the posterior distribution is shown that represents weights in Panel (a)

Source: R. Chandra and Y. Xu, "Bayesian neural networks for stock market prediction before and during COVID-19", PLoS One, 2021.

Similarly to standard neural networks, Bayesian neural networks also have universal continuous function approximation capabilities. On the other hand, the probabilistic model directly specifies the model through the interaction between known parameters to generate data. The probabilistic neural network model employs the posterior distribution to provides uncertainty quantification on the predictions.

The challenge of Bayesian inference is to learn a posterior distribution of neural network weights and biases to represent the data. We begin inference with prior distributions over the weights and biases of the network with a sampling scheme and a likelihood function given training data.

Since non-linear activation functions exist in the network, the conjugacy of prior and posterior is lost and inference sample scheme is used to construct the posterior distribution using the prior distribution and the data.

Likelihood function

We use the same idea from Bayesian logistic regression that uses Metropolis-Hastings MCMC with Random-Walk proposal distribution. Consider the equations used initially for logistic regression to be used for the neural network model in the next lesson.

The likelihood is given by

$$p(\mathbf{y_S}|\boldsymbol{\theta}) = \frac{1}{(2\pi\tau^2)^{S/2}} \times \exp\left(-\frac{1}{2\tau^2}\sum_{t\in S}(\mathbf{y_t} - f(\bar{\mathbf{x}}_\mathbf{t}))^2\right)$$

Our log-likelihood is given by

$$logp(\mathbf{y} \mid \mathbf{x}, \theta, \tau^2) = -log((2\pi\tau^2)^{S/2}) - \frac{1}{2\tau^2}\sum_{t=1}^{S}(\mathbf{y}_t - f(\mathbf{x}_t, \theta))^2$$

The prior is given by

$$p(\boldsymbol{\theta}) \propto \frac{1}{(2\pi\sigma^2)^{L/2}} \times \exp\left\{ -\frac{1}{2\sigma^2}\left(\sum_{i=1}^{M}\theta\right)\right\} \times \tau^{-2(1+\nu_1)} \exp\left(\frac{-\nu_2}{\tau^2}\right)$$

Our log-prior is given by

$$\log p(\theta) \propto -\frac{M}{2}\log 2\pi\sigma^2 - \frac{1}{2\sigma^2}\left(\sum_{i=1}^{M}\theta^2\right) - (1+\nu_1)\log\tau^2 - \frac{\nu_2}{\tau^2}$$

L represents the number of weights and biases which will increase in case of neural networks.

Lesson is based on: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647

References:

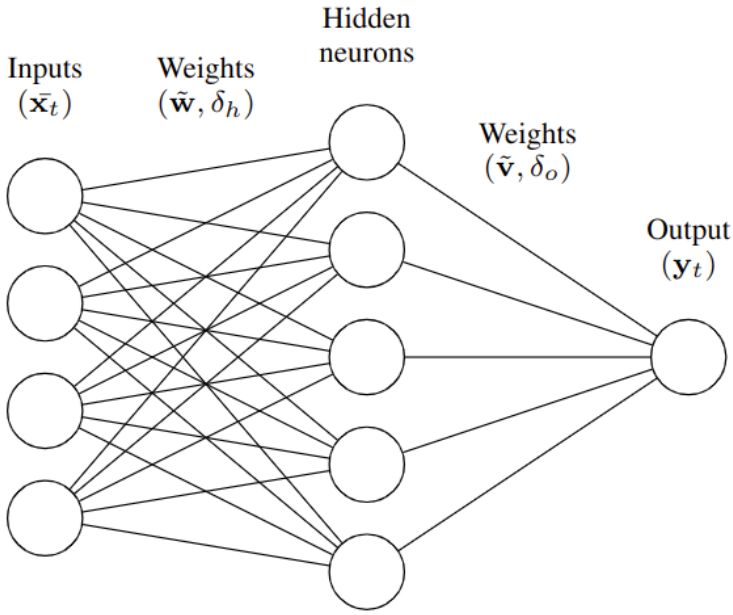
1. Vehtari, A., Sarkka, S., & Lampinen, J. (2000, July). On MCMC sampling in Bayesian MLP neural networks. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium* (Vol. 1, pp. 317-322). IEEE. https://ieeexplore.ieee.org/abstract/document/857855 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.6539&rep=rep1&type=pdf
2. Andrieu, C., De Freitas, N., Doucet, A., & Jordan, M. I. (2003). An introduction to MCMC for machine learning. *Machine learning*, *50*(1-2), 5-43. https://www.cs.ubc.ca/~arnaud/andrieu_defreitas_doucet_jordan_intromontecarlomachinelearning.pdf
3. Neal, R. M. (2011). MCMC using Hamiltonian dynamics. *Handbook of markov chain monte carlo*, *2*(11), 2. https://arxiv.org/pdf/1206.1901.pdf%20http://arxiv.org/abs/1206.1901.pdf
4. Song, J., Zhao, S., & Ermon, S. (2017). A-nice-mc: Adversarial training for mcmc. In *Advances in Neural Information Processing Systems* (pp. 5140-5150). http://papers.nips.cc/paper/7099-a-nice-mc-adversarial-training-for-mcmc.pdf
5. Sharaf, T., Williams, T., Chehade, A., & Pokhrel, K. (2020). BLNN: An R package for training neural networks using Bayesian inference. *SoftwareX*, *11*, 100432. https://www.sciencedirect.com/science/article/pii/S235271101930322X

MCMC Neural Network

Random-walk MCMC for Bayesian neural network for time series prediction problem. Note that no Langevin-gradients are used in this version.

Langevin-gradient Bayesian neural networks

We use a feedforward neural network with a single hidden layer:



We draw the proposed values for the parameters (θ^p) according to a one-step (epoch) gradient, as shown in below:

$$\theta^p \sim \mathcal{N}(\bar{\theta}^{[s]}, \Sigma_\theta)$$

A Gaussian distribution with a standard deviation of Σ_θ, and mean ($\bar{\theta}^{[s]}$) calculated using a gradient based update of the parameter values from the previous step ($\theta^{[s]}$).

$$\bar{\theta}^{[s]} = \theta^{[s]} + r \times \nabla E(\theta^{[s]})$$

with learning rate r and gradient update ($\nabla E(\theta^{[s]})$) according to the model residuals.

$$E(\theta^{[s]}) = \sum_{t \in \mathcal{T}} (y_t - F(\mathbf{x}_i, \theta^{[s]}))^2$$

$$\nabla E(\theta^{[s]}) = \left(\frac{\partial E}{\partial \theta_1}, \ldots, \frac{\partial E}{\partial \theta_L} \right)$$

Hence, the Langevin proposal distribution (also referred as Langevin-gradient) consists of 2 parts:

1. Gradient descent-based weight update
2. Addition of Gaussian noise from $\mathcal{N}(0, \Sigma_\theta)$

We need to ensure that the detailed balance is maintained while sampling, since the Langevin proposals are not symmetric. We note that MCMC implementations with relaxed detailed balance conditions for some applications also exist. Therefore, we use a combined update in the Metropolis-Hastings step, which accepts the proposal θ^p for a position s with the probability α, as shown below:

$$\alpha = \min\left\{1, \frac{p(\theta^p|\mathbf{y})q(\theta^{[s]}|\theta^p)}{p(\theta^{[s]}|\mathbf{y})q(\theta^p|\theta^{[s]})}\right\}$$

where $p(\theta^p|\mathbf{y})$ and $p(\theta^{[s]}|\mathbf{y})$ can be computed using the likelihood and prior. We give the ratio of the proposed and the current $q(\theta^p|\theta^{[s]})$ below:

$$q(\theta^{[s]}|\theta^p) \sim N(\bar{\theta}^{[s]}, \Sigma_\theta)$$

which is based on a one-step (epoch) gradient $\nabla E_{\mathbf{y}}\left[\theta^{[s]}\right.$ and learning rate r, as given in Equation \ref{eqn:gradient}

$$\bar{\theta}^{[s]} = \theta^{[s]} + r \times \nabla E_{\mathbf{y}}\left[\theta^{[s]}\right]$$

Thus, this ensures that the detailed balance condition holds, and the sequence $\theta^{[s]}$ converges to draw from the posterior $p(\theta|\mathbf{y})$. Since our implementation is in the log-scale, we give the log-posterior below:

$$\log\left(p(\theta|\mathbf{y})\right) = log\left(p(\theta)\right) + \log\left(p(\mathbf{y}|\theta)\right) + \log(q(\theta|\theta^*))$$

Lesson is based on: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647

Note the code below does not run. We will run them all together in the next session with data.

```
# https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/code/BN

    def likelihood_func(self, neuralnet, data, w, tausq):
        y = data[:, self.topology[0]]
                fx = neuralnet.evaluate_proposal(data, w)
                rmse = self.rmse(fx, y)
                loss = -0.5 * np.log(2 * math.pi * tausq) - 0.5 * np.square(y - fx) / tausq
                return [np.sum(loss), fx, rmse]

        def prior_likelihood(self, sigma_squared, nu_1, nu_2, w, tausq):
```

```
                h = self.topology[1]  # number hidden neurons
                d = self.topology[0]  # number input neurons
                part1 = -1 * ((d * h + h + 2) / 2) * np.log(sigma_squared)
                part2 = 1 / (2 * sigma_squared) * (sum(np.square(w)))
                log_loss = part1 - part2  - (1 + nu_1) * np.log(tausq) - (nu_2 / tausq)
                return log_loss
```

Below code implements the functions of the neural network. The important function here is evaluate_proposal( ) which encodes the weights in neural networks. In order to implement Langevin gradients, we need to compute gradients with the backward-pass.

```
# https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/code/BN
    def decode(self, w):
        w_layer1size = self.Top[0] * self.Top[1]
                w_layer2size = self.Top[1] * self.Top[2]

                w_layer1 = w[0:w_layer1size]
                self.W1 = np.reshape(w_layer1, (self.Top[0], self.Top[1]))

                w_layer2 = w[w_layer1size:w_layer1size + w_layer2size]
                self.W2 = np.reshape(w_layer2, (self.Top[1], self.Top[2]))
                self.B1 = w[w_layer1size + w_layer2size:w_layer1size + w_layer2size + self.Top[1]]
                self.B2 = w[w_layer1size + w_layer2size + self.Top[1]:w_layer1size + w_layer2size +

        def langevin_gradient(self, data, w, depth):  # BP with SGD (Stocastic BP)

                self.decode(w)  # method to decode w into W1, W2, B1, B2.
                size = data.shape[0]

                Input = np.zeros((1, self.Top[0]))  # temp hold input
                Desired = np.zeros((1, self.Top[2]))
                fx = np.zeros(size)

                for i in xrange(0, depth):
                        for i in xrange(0, size):
                                pat = i
                                Input = data[pat, 0:self.Top[0]]
                                Desired = data[pat, self.Top[0]:]
                                self.ForwardPass(Input)
                                self.BackwardPass(Input, Desired)

                w_updated = self.encode()

                return  w_updated

        def evaluate_proposal(self, data, w ):  # BP with SGD (Stocastic BP)

                self.decode(w)  # method to decode w into W1, W2, B1, B2.
                size = data.shape[0]
```

```
        Input = np.zeros((1, self.Top[0]))  # temp hold input
        Desired = np.zeros((1, self.Top[2]))
        fx = np.zeros(size)

        for i in xrange(0, size):  # to see what fx is produced by your current weight upda
                Input = data[i, 0:self.Top[0]]
                self.ForwardPass(Input)
                fx[i] = self.out

        return fx
```

Next, we look at the main algorithm that shows how the samples are accepted taking Langevin gradeints into account.

Alg. 1 Langevin Dynamics for neural networks

Data: Univariate time series $\mathbf{y}$
Result: Posterior of weights and biases $p(\boldsymbol{\theta}|\mathbf{y})$
Step 1: State-space reconstruction $\mathbf{y}_{\mathcal{A}_{D,T}}$ by Equation 2
Step 2: Define feedforward network as given in Equation 3
Step 3: Define $\boldsymbol{\theta}$ as the set of all weights and biases
Step 4: Set parameters σ^2 ,ν_1, ν_2 for prior given in Equation 6

for *each k until max-samples* **do**
  1.Compute gradient $\Delta\boldsymbol{\theta}^{[k]}$given by Equation 8
  2. Draw $\boldsymbol{\eta}$ from $\mathcal{N}(0, \Sigma_\eta)$
  3. Propose $\boldsymbol{\theta}^* = \theta^{[k]} + \Delta\boldsymbol{\theta}^{[k]} + \boldsymbol{\eta}$
  4. Draw from uniform distribution $u \sim \mathcal{U}[0, 1]$
  5. Obtain acceptance probability α given by Equation 9
  if $u < \alpha$ **then**
    $\boldsymbol{\theta}^{[k+1]} = \boldsymbol{\theta}^*$
  end
  else
    $\boldsymbol{\theta}^{[k+1]} = \boldsymbol{\theta}^{[k]}$
  end
end

Chandra, R., Azizi, L., & Cripps, S. (2017). Bayesian neural learning via langevin dynamics for chaotic time series prediction. In *Neural Information Processing: 24th International Conference, ICONIP 2017, Guangzhou, China, November 14–18, 2017, Proceedings, Part V 24* (pp. 564-573).

```
#Source: https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/
for i in range(samples - 1):

    lx = np.random.uniform(0,1,1)

    if (self.use_langevin_gradients is True) and (lx< self.l_prob):
        w_gd = neuralnet.langevin_gradient(self.traindata, w.copy(), self.sgd_depth) # Eq 8
        w_proposal = np.random.normal(w_gd, step_w, w_size) # Eq 7
        w_prop_gd = neuralnet.langevin_gradient(self.traindata, w_proposal.copy(), self.sgd_depth)
        #first = np.log(multivariate_normal.pdf(w , w_prop_gd , sigma_diagmat)) #how likely is that
        #second = np.log(multivariate_normal.pdf(w_proposal , w_gd , sigma_diagmat)) # this gives n

        wc_delta = (w- w_prop_gd)
        wp_delta = (w_proposal - w_gd )
        sigma_sq = step_w

        first = -0.5 * np.sum(wc_delta  *  wc_delta  ) / sigma_sq  # this is wc_delta.T  *  wc_delt
        second = -0.5 * np.sum(wp_delta * wp_delta ) / sigma_sq

        #in random-walk idealy first - second would be 0 or close to 0.
        diff_prop =  first - second
        langevin_count = langevin_count + 1
    else:
        diff_prop = 0
        w_proposal = np.random.normal(w, step_w, w_size)

    eta_pro = eta + np.random.normal(0, step_eta, 1)
    tau_pro = math.exp(eta_pro)

    [likelihood_proposal, pred_train, rmsetrain] = self.likelihood_func(neuralnet, self.traindata,
    [likelihood_ignore, pred_test, rmsetest] = self.likelihood_func(neuralnet, self.testdata, w_pro
                                                                    tau_pro)

    prior_prop = self.prior_likelihood(sigma_squared, nu_1, nu_2, w_proposal,
                                       tau_pro)  # takes care of the gradients


    diff_prior = prior_prop - prior_current

    diff_likelihood = likelihood_proposal - likelihood

    #mh_prob = min(1, math.exp(diff_likelihood + diff_prior + diff_prop))

    try:
        mh_prob = min(1, math.exp(diff_likelihood+diff_prior+ diff_prop))

    except OverflowError as e:
        mh_prob = 1
```

The results summary of benchmark problems is shown below.

Table 1: Results for the respective methods

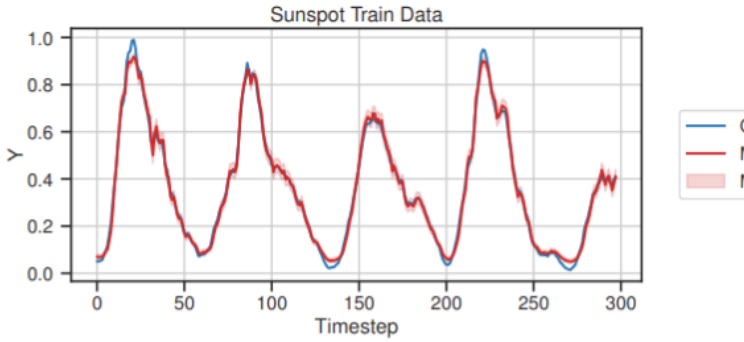
Problem	Method	Train (mean)	Train (std)	Test (mean)	Test(std)	% Accepted
Lazer	GD*	0.02191	0.00129	0.02675	0.00085	-
	GD	0.01035	0.00162	0.01732	0.00142	-
	MCMC	0.02549	0.00837	0.02643	0.00718	9.70
	LD-MCMC	0.01658	0.00211	0.02280	0.00351	57.86
Sunspot	GD*	0.02117	0.00160	0.02359	0.00209	-
	GD	0.00775	0.00026	0.00975	0.00031	-
	MCMC	0.01466	0.00174	0.01402	0.00178	4.55
	LD-MCMC	0.01155	0.00167	0.01090	0.00146	45.89
Mackey	GD*	0.00590	0.00026	0.00669	0.00029	-
	GD	0.00286	0.00025	0.0033	0.00026	-
	MCMC	0.00511	0.00058	0.00520	0.00058	1.74
	LD-MCMC	0.00615	0.00091	0.00627	0.00091	30.35
Lorenz	GD*	0.01570	0.00056	0.01608	0.00052	-
	GD	0.00400	0.00024	0.00460	0.00026	-
	MCMC	0.00813	0.00174	0.00713	0.00150	2.74
	LD-MCMC	0.00890	0.00211	0.00821	0.00207	26.95
Rossler	GD*	0.01570	0.00056	0.01608	0.00052	-
	GD	0.00281	0.00029	0.00462	0.00045	-
	MCMC	0.01371	0.00291	0.01355	0.00297	3.69
	LD-MCMC	0.00722	0.00121	0.00692	0.00120	35.53
Henon	GD*	0.01366	0.00033	0.01778	0.00025	-
	GD	0.00555	0.00029	0.00604	0.00025	-
	MCMC	0.03256	0.03920	0.03127	0.03850	8.71
	LD-MCMC	0.00948	0.00112	0.00912	0.00114	27.17

Chandra, R., Azizi, L., & Cripps, S. (2017). Bayesian neural learning via langevin dynamics for chaotic time series prediction. In *Neural Information Processing: 24th International Conference, ICONIP 2017, Guangzhou, China, November 14–18, 2017, Proceedings, Part V 24* (pp. 564-573).

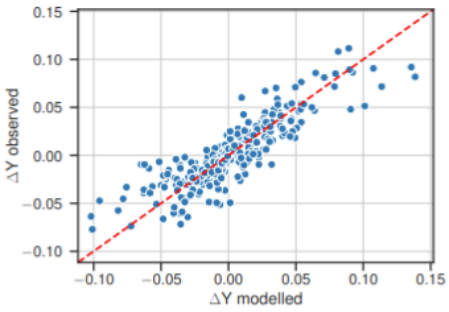
Results comparing Bayes Linear model with Bayes Neural Networks: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647

Method	Problem	Train RMSE	Test RMSE	Accept. rate
Bayesian linear model	Sunspot	0.025 (0.013)	0.022 (0.012)	13.5%
Bayesian neural network	Sunspot	0.027 (0.007)	0.026 (0.007)	7.4%
Bayesian linear model	Abalone	0.085 (0.005)	0.086 (0.005)	5.8%
Bayesian neural network	Abalone	0.080 (0.002)	0.080 (0.002)	3.8%

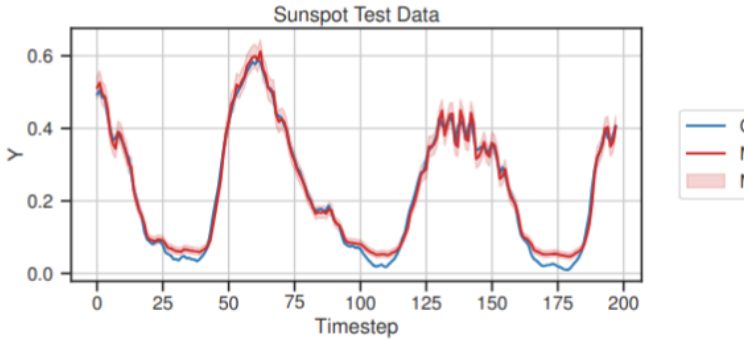
Bayesian linear regression for Sunspot prediction: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647



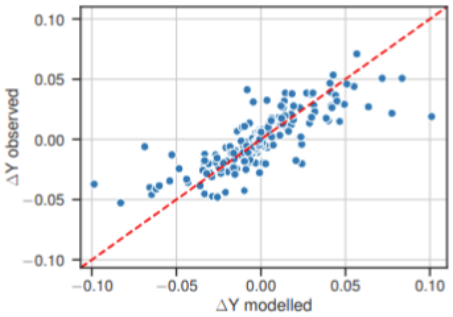
(a) Predictions and actual train dataset for Sunspot



(b) Predictions and actual of ΔY train dataset for Sunspot prediction.

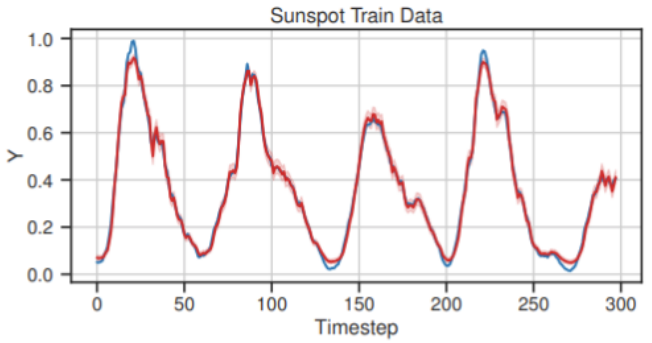


(c) Predictions and actual test dataset for Sunspot

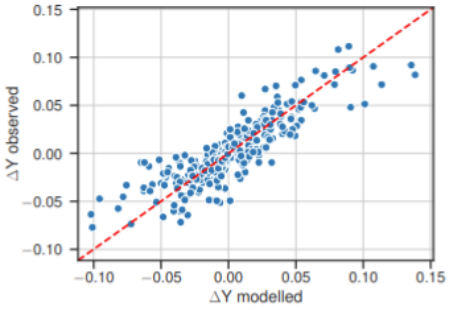


(d) Predictions and actual of ΔY test dataset for Sunspot prediction.

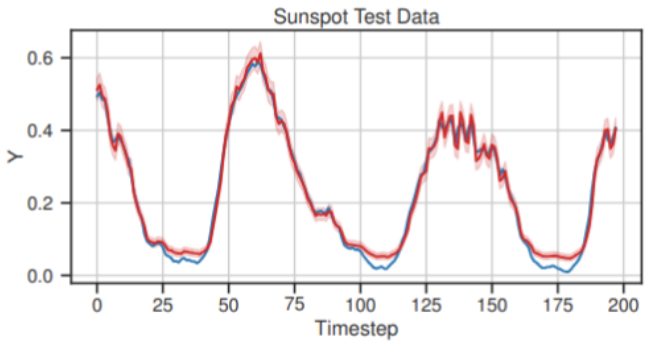
Bayesian neural network for Sunspot prediction: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647



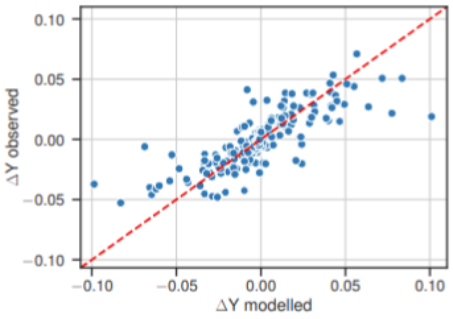
(a) Predictions and actual train dataset for Sunspot



(b) Predictions and actual of ΔY train dataset for Sunspot prediction.



(c) Predictions and actual test dataset for Sunspot



(d) Predictions and actual of ΔY test dataset for Sunspot prediction.

References:

1. Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)* (pp. 681-688). http://people.ee.duke.edu/~lcarin/398_icmlpaper.pdf
2. Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647
3. Chandra R; Jain K; Deo RV; Cripps S, 2019, 'Langevin-gradient parallel tempering for Bayesian neural learning', *Neurocomputing*, vol. 359, pp. 315 - 326, http://dx.doi.org/10.1016/j.neucom.2019.05.082 https://github.com/rohitash-chandra/research/blob/master/2019/Chandra_LangevinNeurocom2019.pdf
4. Chandra, R., Azizi, L., & Cripps, S. (2017). Bayesian neural learning via langevin dynamics for chaotic time series prediction. In *Neural Information Processing: 24th International Conference, ICONIP 2017, Guangzhou, China, November 14–18, 2017, Proceedings, Part V 24* (pp. 564-573).

Detailed balance for proposal distribution

We go back to the acceptance probability and look at the ratio between the posterior $p(.)$ and prior $q(.)$ values as shown below. We note that in the case when the proposal distribution is a random-walk, the prior ratios cancel out, but in the case when the proposal distribution is Langevin-gradients, the priors do not cancel out.

$$\alpha = \min\left\{1, \frac{p(\boldsymbol{\theta}^p|\mathbf{y})q(\boldsymbol{\theta}^{[k]}|\boldsymbol{\theta}^p)}{p(\boldsymbol{\theta}^{[k]}|\mathbf{y})q(\boldsymbol{\theta}^p|\boldsymbol{\theta}^{[k]})}\right\}$$

essentially means

$q(\boldsymbol{\theta}^p|\boldsymbol{\theta}^{[k]})$ is given by $\boldsymbol{\theta}^p \sim \mathcal{N}(\bar{\boldsymbol{\theta}}^{[k]}, \Sigma_\theta)$

and $q(\boldsymbol{\theta}^{[k]}|\boldsymbol{\theta}^p) \sim N(\bar{\boldsymbol{\theta}}^p, \Sigma_{\boldsymbol{\theta}})$

taking into account the gradient ∇E and learning rate r, we utilize $\bar{\boldsymbol{\theta}}^p = \boldsymbol{\theta}^p + r \times \nabla E_{\mathbf{y}}\left[\boldsymbol{\theta}^p\right]$

The above ensures that the detailed balance condition holds and the sequence $\boldsymbol{\theta}^{[k]}$ converges to draws from the posterior $p(\boldsymbol{\theta}|\mathbf{y})$.

In the code, we have the following. The first and second are representing the numerator and denominator.

```
if (self.use_langevin_gradients is True) and (lx< self.l_prob):
       w_gd = neuralnet.langevin_gradient(self.traindata, w.copy(), self.sgd_depth) # Eq 8
       w_proposal = np.random.normal(w_gd, step_w, w_size) # Eq 7
       w_prop_gd = neuralnet.langevin_gradient(self.traindata, w_proposal.copy(), self.sgd_depth)
       #first = np.log(multivariate_normal.pdf(w , w_prop_gd , sigma_diagmat))
       #(how likely is that you end up with w given w_prop_gd)
       #second = np.log(multivariate_normal.pdf(w_proposal , w_gd , sigma_diagmat))
       #(this gives numerical instability - hence we give a simple implementation next that takes

       wc_delta = (w- w_prop_gd)
       wp_delta = (w_proposal - w_gd )
       sigma_sq = step_w
```

```
        first = -0.5 * np.sum(wc_delta  *  wc_delta  ) / sigma_sq
        second = -0.5 * np.sum(wp_delta * wp_delta ) / sigma_sq

        #in random-walk idealy first - second would be 0 or close to 0
        #and hence we assume it cancels out
        diff_prop =  first - second


# get likelihood and prior likelihood values (prior_prop and diff_likelihood)

    diff_prior = prior_prop - prior_current
    diff_likelihood = likelihood_proposal - likelihood
    mh_prob = min(1, math.exp(diff_likelihood + diff_prior + diff_prop))
```

Below is a figure showing random-walk proposal distribution.

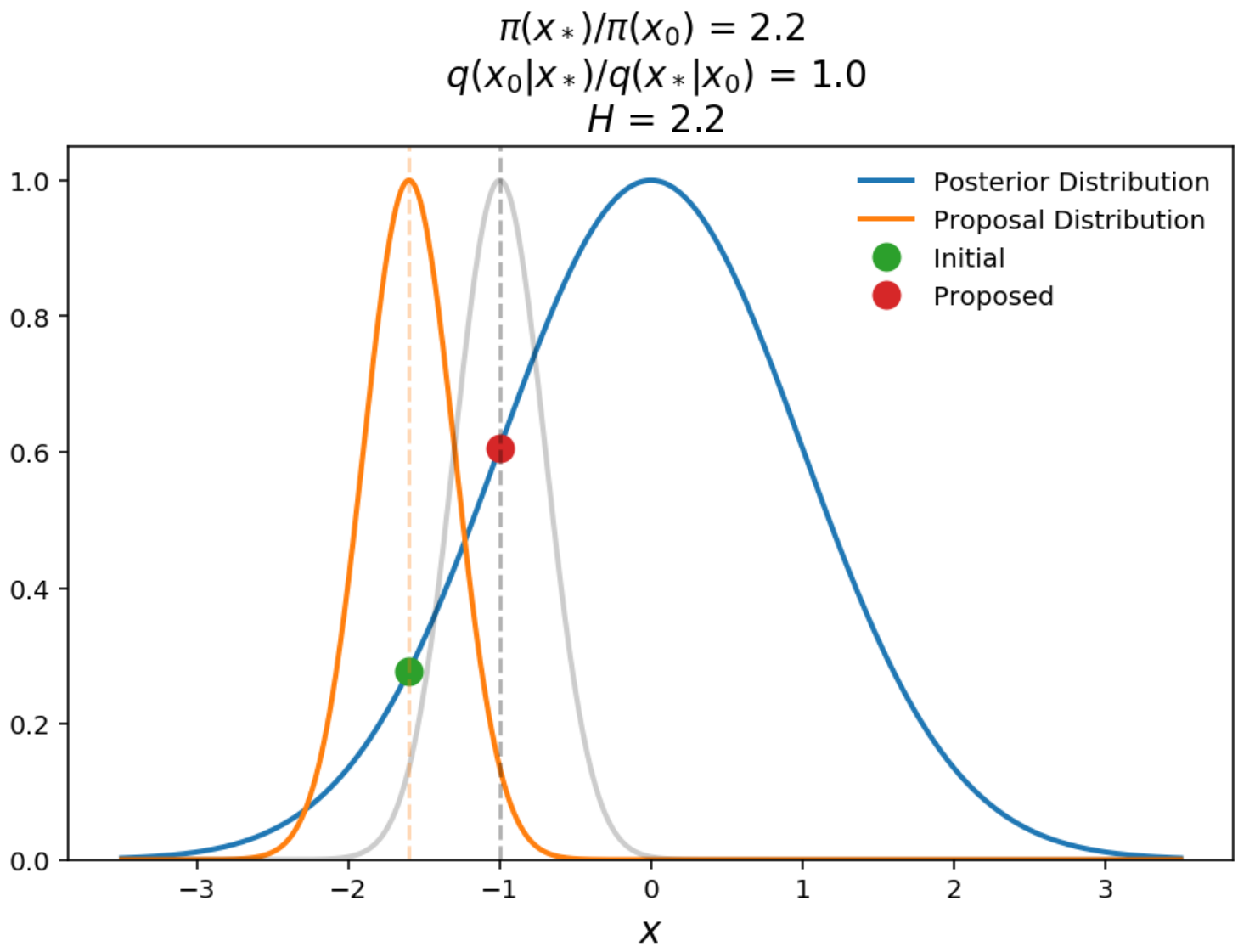


Figure Source. *"The blue shows the 1-D Gaussian posterior distribution, the orange is the Gaussian proposal distribution, q(x∗|x0), (σ=0.5), and the green and red points are the initial and proposed parameters, respectively. For illustration purposes, the gray curve shows q(x0|x∗) to show the symmetry of the proposal distribution. In this case we see that the proposed point returns a Hastings ratio of 2.2 (i.e. transition probability of 1) and therefore the jump will be accepted."* A Practical Guide to MCMC Part 1: MCMC Basics: https://jellis18.github.io/post/2018-01-02-mcmc-part1/

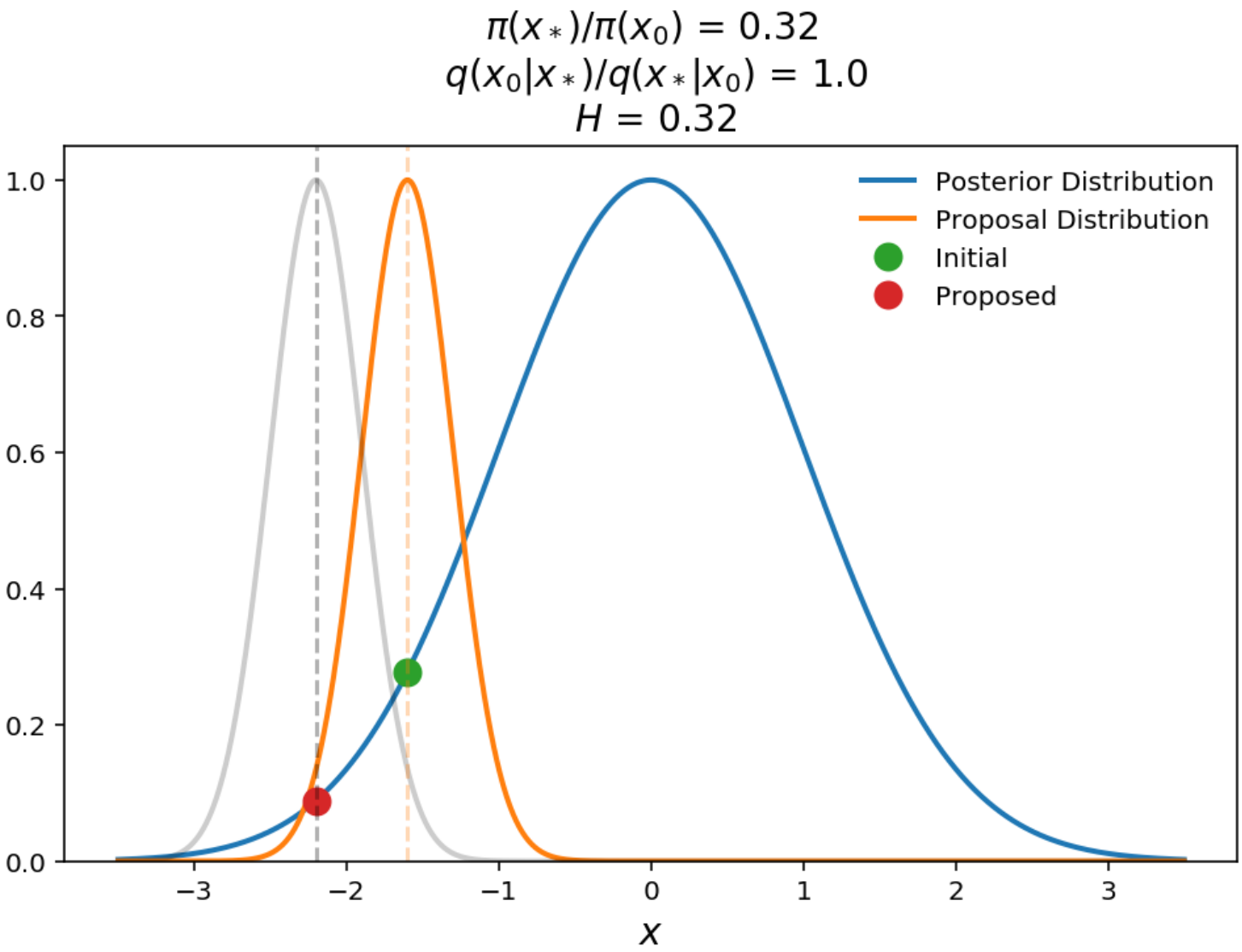


Figure Source. " In this plot, we show what happens when a jump is proposed to a lower probability region. The transition probability is equal to the Hastings ratio here (remember transition probability is min(1,H) which is 0.32, which means that we will move to this new point with 32% probability. This ability to move back down the posterior distribution is what allows MCMC to sample the full probability distribution instead of just finding the global maximum." A Practical Guide to MCMC Part 1: MCMC Basics: https://jellis18.github.io/post/2018-01-02-mcmc-part1/

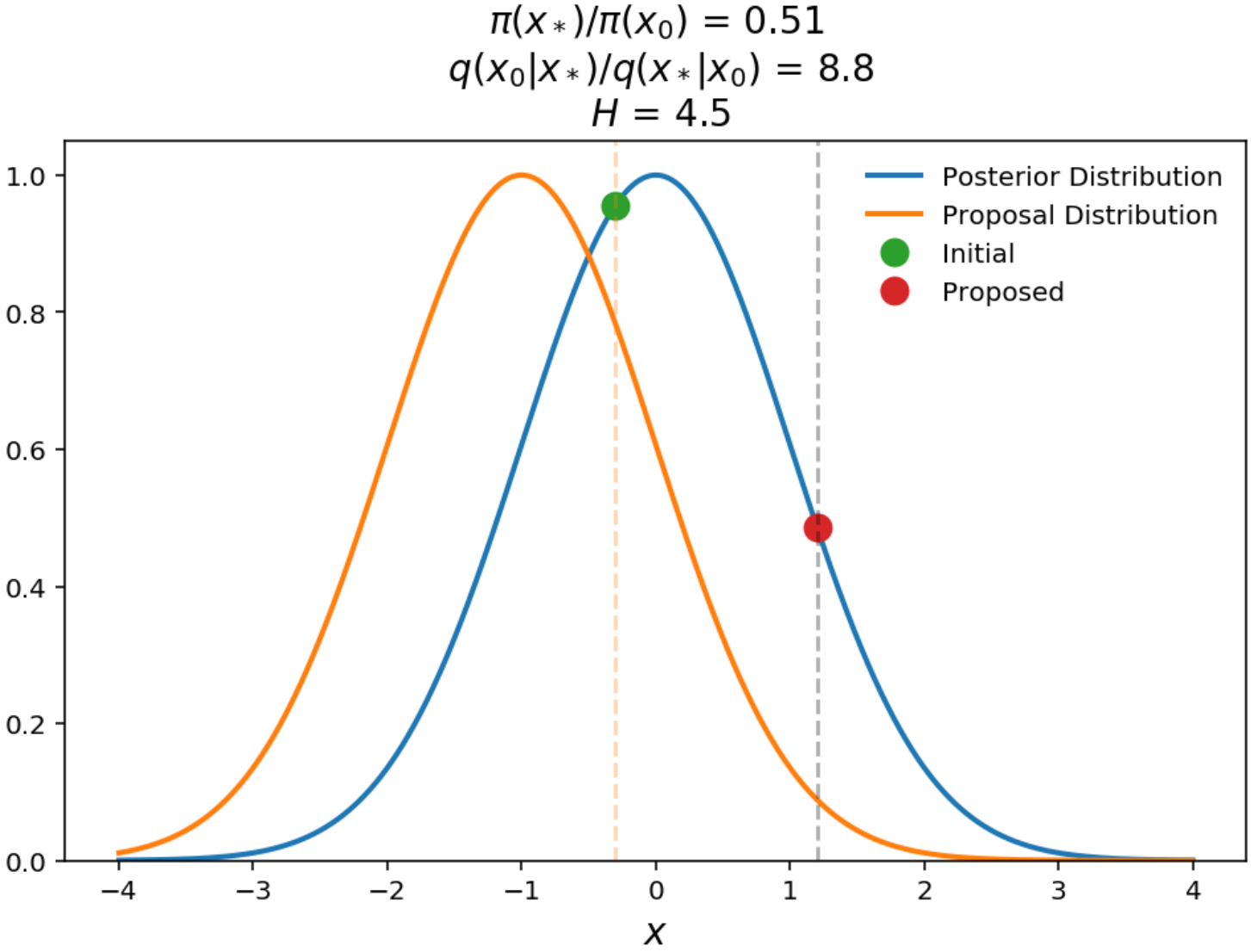


Figure Source. "In the above plot we show that the proposal distribution is a fixed Gaussian q(x∗|x0)∼Normal(-1, 1). Here we show that even though the proposed point is at a *lower* posterior probability than the initial point, the Hastings ratio is still >1 will accept jump with 100% probability). Qualitatively this makes sense because we need to weight that proposed point higher to take into account for the fact that the proposed point is “hard” to get to even though it still has a relatively high posterior probability value." A Practical Guide to MCMC Part 1: MCMC Basics: https://jellis18.github.io/post/2018-01-02-mcmc-part1/

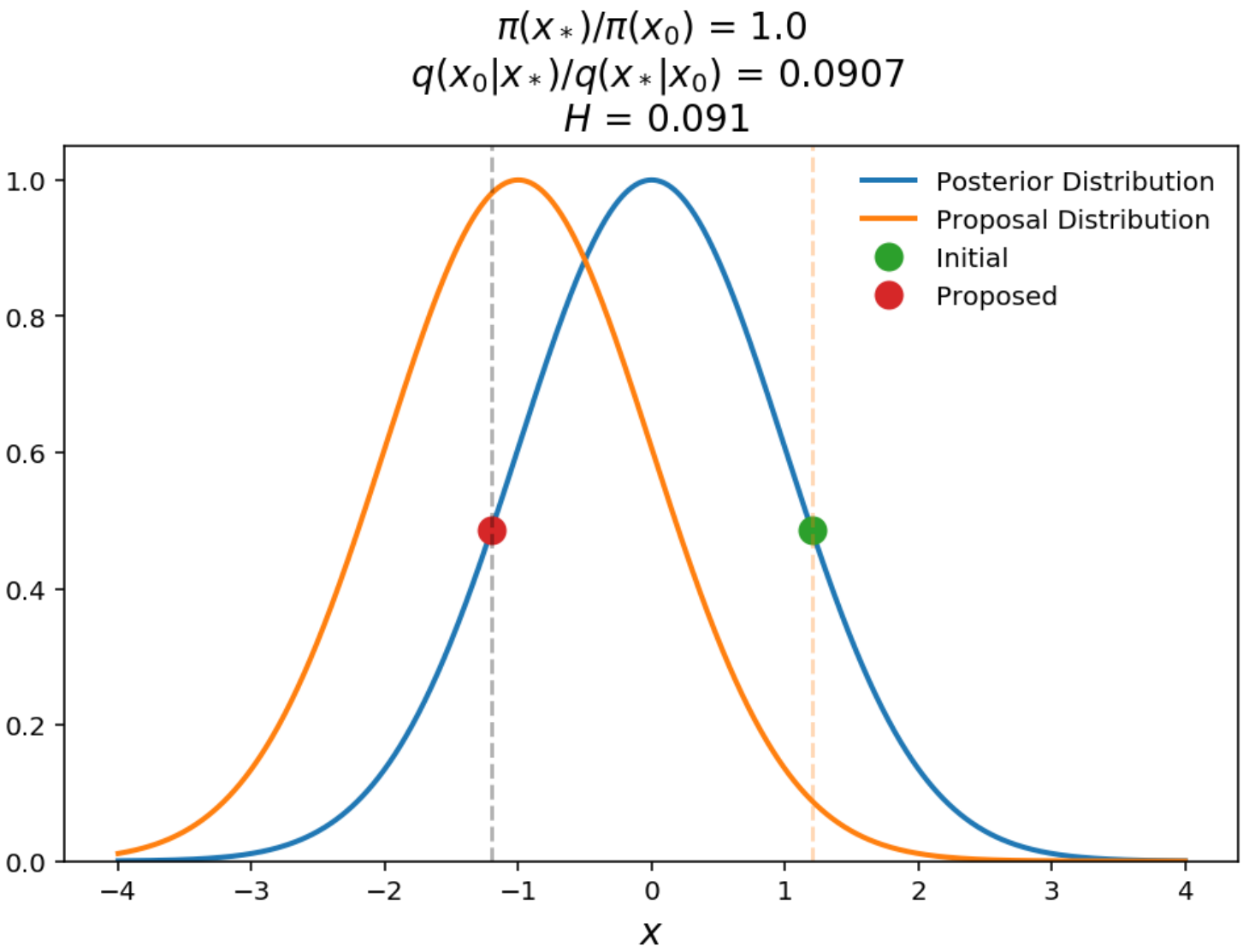


Figure Source. "In this last example, we show the opposite effect. In this case, even though the posterior probabilities are the same for the current and proposed points, the Hastings ratio is only 0.09 (i.e. 9% chance of accepting jump). Again, with some thought this makes sense. The proposed point must be weighted down because it is near the peak of the proposal distribution (i.e. lots of points will be proposed around this position) and therefore is "easy" to get to even though the posterior probability is no different than at the initial point." A Practical Guide to MCMC Part 1: MCMC Basics: https://jellis18.github.io/post/2018-01-02-mcmc-part1/

Further information:

1. https://stats.stackexchange.com/questions/332350/detailed-balance-distribution-reflecting-a-random-walk
2. https://jellis18.github.io/post/2018-01-02-mcmc-part1/
3. https://courses.physics.illinois.edu/phys466/sp2013/lnotes/PPT/RandomWalk.PDF
4. https://people.duke.edu/~kh269/teaching/notes/MetropolisExplanation.pdf
5. https://cims.nyu.edu/~holmes/teaching/asa19/handout_Lecture3_2019.pdf

Langevin-gradient Bayesian neural networks

This code slide does not have a description.

Bayesian Neural Networks - Classification

Bayesian neural networks via the MCMC sampler require an appropriate likelihood function, suitable for discrete outcomes, to capture classification problems. Hence, we use the multinomial likelihood, which is applicable to both binary and multi-class classification problems. We define the multinomial log-likelihood function for the classification problems

$$\log\left(p(\mathbf{y}|\theta)\right) = \sum_{t \in N} \sum_{k=1}^{K} z_{t,k} \log \pi_k$$

for classes $k = 1, \ldots, K$, where π_k is the output of the neural network model after applying the transfer function, and N is the number of instances in the training data. In this case, we utilize the \textit{softmax function} as the transfer function:

$$\pi_k = \frac{\exp(f(x_p))}{\sum_{k=1}^{K} \exp(f(x_k))}$$

for $k = 1, \ldots, K$. $z_{t,k}$ is an indicator variable for the given instance of data t. We define class k in the data by

$$z_{t,k} = \begin{cases} 1, & \text{if } y_t = k \\ 0, & \text{otherwise.} \end{cases}$$

We note that we do not use the noise parameter (i.e., τ^2) as in the case of the inverse gamma distribution for the Gaussian likelihood for the regression case; hence, we do not need a prior distribution for the noise.

We will only use a Gaussian prior for weights and biases of the neural network model. Therefore, in the case of classification, our prior distribution simplifies:

$$p(\theta) \propto \frac{1}{(2\pi\sigma^2)^{M/2}} \times \exp\left\{ -\frac{1}{2\sigma^2} \left(\sum_{i=1}^{M} \theta^2 \right) \right\}$$

and finally, the log-prior can be expressed as:

$$\log p(\theta) \propto -\frac{M}{2} \log 2\pi\sigma^2 \times -\frac{1}{2\sigma^2} \left(\sum_{i=1}^{M} \theta^2 \right)$$

```
#taken from: https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/m

    def loglikelihood(self, neuralnet, data, w, tausq):

                if self.prob == 'regression':
                        [log_lhood, prediction, perf] = self.gaussian_loglikelihood(neuralnet, data
                elif self.prob == 'classification':
                        [log_lhood, prediction, perf] = self.multinomial_loglikelihood(neuralnet, d

                return [log_lhood, prediction, perf]

        def prior(self, sigma_squared, nu_1, nu_2, w, tausq):

                if self.prob == 'regression':
                        logprior = self.prior_regression(sigma_squared, nu_1, nu_2, w, tausq)
                elif self.prob == 'classification':
                        logprior = self.prior_classification(sigma_squared, nu_1, nu_2, w)

                return logprior

        def gaussian_loglikelihood(self, neuralnet, data, w, tausq):

                y = data[:, self.topology[0]]
                fx, prob = neuralnet.evaluate_proposal(data, w) #ignore prob
                rmse = self.rmse(fx, y)

                n = y.shape[0]  # will change for multiple outputs (y.shape[0]*y.shape[1])
                log_lhood = -n/2 * np.log(2 * math.pi * tausq) - (1/(2*tausq)) * np.sum(np.square(y
                return [log_lhood, fx, rmse]

        def prior_regression(self, sigma_squared, nu_1, nu_2, w, tausq): # for weights and biases a
                h = self.topology[1]  # number hidden neurons
                d = self.topology[0]  # number input neurons
                part1 = -1 * ((d * h + h + 2) / 2) * np.log(sigma_squared)
                part2 = 1 / (2 * sigma_squared) * (sum(np.square(w)))
                log_loss = part1 - part2  - (1 + nu_1) * np.log(tausq) - (nu_2 / tausq)
                return log_loss


        def multinomial_loglikelihood(self, neuralnet, data, w):
                y = data[:, self.topology[0]]
                fx, prob = neuralnet.evaluate_proposal(data,w)
                acc= self.accuracy(fx,y)
                z = np.zeros((data.shape[0],self.topology[2]))
                lhood = 0
                for i in range(data.shape[0]):
                        for j in range(self.topology[2]):
                                if j == y[i]:
                                        z[i,j] = 1
                                lhood += z[i,j]*np.log(prob[i,j])
```

```
        return [lhood, fx, acc]

    def prior_classification(self, sigma_squared, nu_1, nu_2, w): # for weights and biases only
        h = self.topology[1]  # number hidden neurons
        d = self.topology[0]  # number input neurons
        part1 = -1 * ((d * h + h + self.topology[2]+h*self.topology[2]) / 2) * np.log(sigma
        part2 = 1 / (2 * sigma_squared) * (sum(np.square(w)))
        log_loss = part1 - part2
        return log_loss
```

Sampler for regression/classification

```
    for i in range(samples - 1):

                lx = np.random.uniform(0,1,1)

                if (self.use_langevin_gradients is True) and (lx< self.l_prob):
                        w_gd = neuralnet.langevin_gradient(self.traindata, w.copy(), self.s
                        w_proposal = np.random.normal(w_gd, step_w, w_size)
                        w_prop_gd = neuralnet.langevin_gradient(self.traindata, w_proposal.
                        #first = np.log(multivariate_normal.pdf(w , w_prop_gd , sigma_diagm
                        #second = np.log(multivariate_normal.pdf(w_proposal , w_gd , sigma_
                        # this gives numerical instability - hence we give a simple impleme

                        wc_delta = (w- w_prop_gd)
                        wp_delta = (w_proposal - w_gd )

                        sigma_sq = step_w

                        first = -0.5 * np.sum(wc_delta  *  wc_delta  ) / sigma_sq  # this i
                        second = -0.5 * np.sum(wp_delta * wp_delta ) / sigma_sq

                        diff_prop =  first - second
                        langevin_count = langevin_count + 1

                else:
                        diff_prop = 0
                        w_proposal = np.random.normal(w, step_w, w_size)


                if self.prob == 'regression':
                        eta_pro = eta + np.random.normal(0, step_eta, 1)
                        tau_pro = math.exp(eta_pro)
                else:# not used in case of classification
                        eta_pro = 0
                        tau_pro = 0


                [likelihood_proposal, pred_train, p_train] = self.loglikelihood(neuralnet,

                [likelihood_ignore, pred_test, p_test] = self.loglikelihood(neuralnet, self
```

```
                prior_prop = self.prior(sigma_squared, nu_1, nu_2, w_proposal,
                                                                        tau_pro)


                diff_prior = prior_prop - prior_current

                diff_likelihood = likelihood_proposal - likelihood

                #mh_prob = min(1, math.exp(diff_likelihood + diff_prior + diff_prop))

                try:
                        mh_prob = min(1, math.exp(diff_likelihood+diff_prior+ diff_prop))

                except OverflowError as e:
                        mh_prob = 1


                u = random.uniform(0, 1)

                if u < mh_prob:
                        # Update position
                        naccept += 1
                        likelihood = likelihood_proposal
                        prior_current = prior_prop
                        w = w_proposal
                        eta = eta_pro #only used for regression
```

Results for classification: Bayesian neural network for Sunspot prediction: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647

Method	Problem	Train Accuracy	Test Accuracy	Accept. rate
Bayesian linear model	Iris	90.392% (2.832)	90.844% (3.039)	83.5%
Bayesian neural network	Iris	97.377% (0.655)	98.116% (1.657)	97.0%
Bayesian linear model	Ionosphere	89.060% (1.335)	85.316% (2.390)	58.8%
Bayesian neural network	Ionosphere	99.632% (0.356)	92.668% (1.890)	94.5%

References:

1. Chandra R; Jain K; Deo RV; Cripps S, 2019, 'Langevin-gradient parallel tempering for Bayesian

neural learning', *Neurocomputing*, vol. 359, pp. 315 - 326, http://dx.doi.org/10.1016/j.neucom.2019.05.082 https://github.com/rohitash-chandra/research/blob/master/2019/Chandra_LangevinNeurocom2019.pdf

2. Chandra R; Kapoor A, 2020, 'Bayesian neural multi-source transfer learning', *Neurocomputing*, vol. 378, pp. 54 - 64, http://dx.doi.org/10.1016/j.neucom.2019.10.042 https://github.com/rohitash-chandra/research/blob/master/2020/Chandra_NC2020.pdf

Bayes Neural Network for Classification and Regression

Based on: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647 https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial

References

https://github.com/sydney-machine-learning/parallel-tempering-neural-net

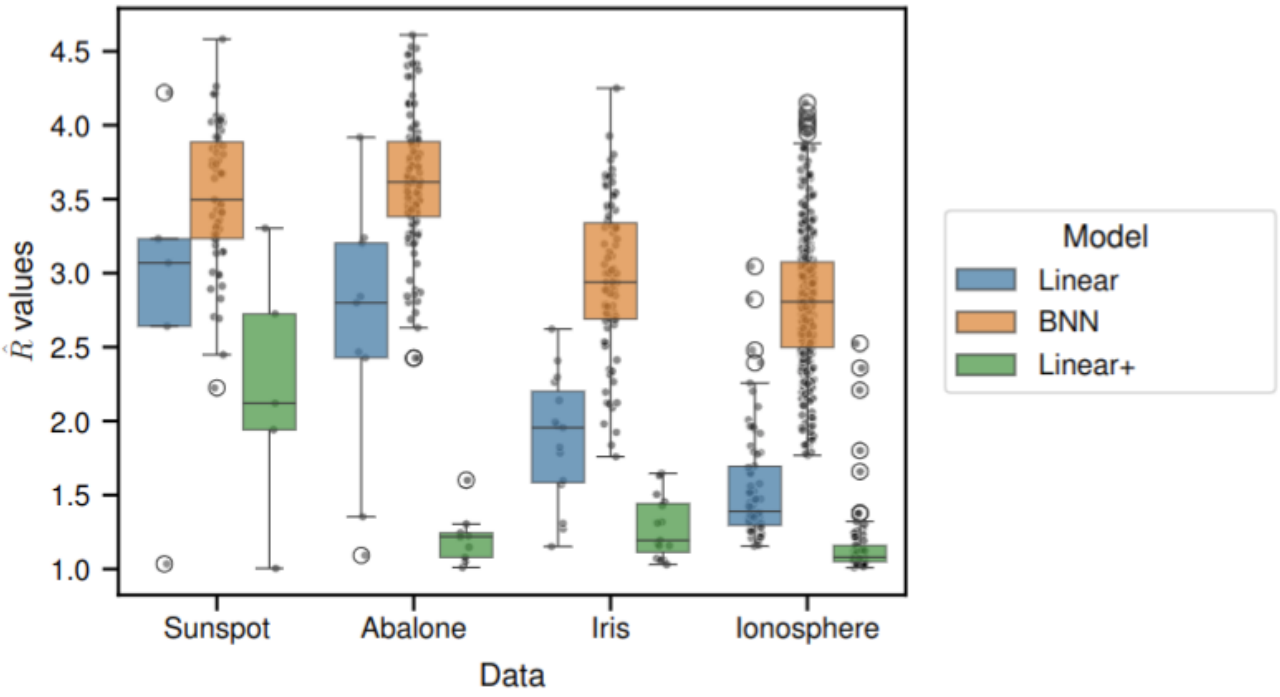
Chandra R; Jain K; Deo RV; Cripps S, 2019, 'Langevin-gradient parallel tempering for Bayesian neural learning', Neurocomputing, vol. 359, pp. 315 - 326, http://dx.doi.org/10.1016/j.neucom.2019.05.082,

Convergence diagnosis

Gelman-Rubin diagnostic

The Gelman-Rubin diagnostic evaluates MCMC convergence by analyzing the behaviour of multiple Markov chains. Given multiple chains from different experimental runs, assessment is done by comparing the estimated between-chains and within-chain variances for each parameter, where large differences between the variances indicate non-convergence.

We calculate the potential scale reduction factor (PSRF) which gives the ratio of the current variance in the posterior variance for each parameter compared to that being sampled. The values for the PSRF near 1 indicates convergence.



Convergence Results for classification and regression: Bayesian neural network for Sunspot prediction: Chandra R, Simmons J. Bayesian neural networks via MCMC: a Python-based tutorial. IEEE Access. 2024: https://ieeexplore.ieee.org/abstract/document/10530647

Autocorrelation time

Autocorrelation refers to the correlation of a time series with a delayed copy of itself over successive time intervals which give the degree of similarity. It measures the relationship between a variable's current value and its past values. A positive 1 autocorrelation represents a perfect positive correlation, while an autocorrelation of negative 1 represents a perfect negative correlation. More information: https://online.stat.psu.edu/stat462/node/188/

Autocorrelation time is used as a convergence diagnosis for MCMC sampling algorithms. Implementation in Emcee library: https://emcee.readthedocs.io/en/stable/tutorials/autocorr/

A tutorial is given here: https://dfm.io/posts/autocorr/

References

1. Brooks, S. P., and A. Gelman. 1998. General methods for monitoring convergence of iterative simulations. *Journal of Computational and Graphical Statistics* 7: 434–455. http://www2.stat.duke.edu/~scs/Courses/Stat376/Papers/ConvergeDiagnostics/BrooksGelman.pdf
2. Gelman, A., and D. B. Rubin. 1992. Inference from iterative simulation using multiple sequences. *Statistical Science* 7: 457–472. https://projecteuclid.org/download/pdf_1/euclid.ss/1177011136

Implementation

1. Stata: https://www.stata.com/new-in-stata/gelman-rubin-convergence-diagnostic/
2. Emcee: http://greg-ashton.physics.monash.edu/the-gelman-rubin-statistic-and-emcee.html
3. Autocorrelation: https://emcee.readthedocs.io/en/stable/tutorials/autocorr/

Bayesian Logistic Regression in R

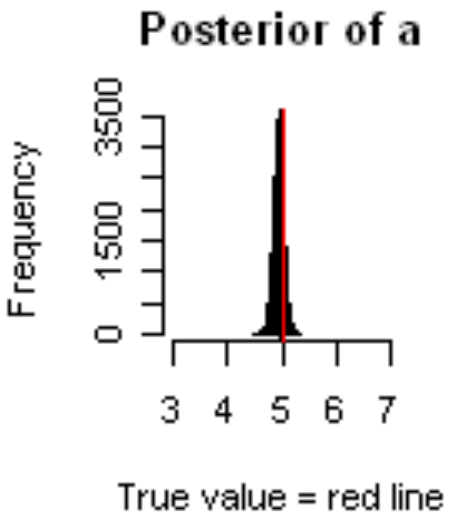
Here is an example of Bayesian Logistic Regression with MCMC in R. Note that this follows the same approach as before, with some minor changes. The way the Metropolis-Hasting acceptance criterion is computed is slightly different in syntax but essentially the same.

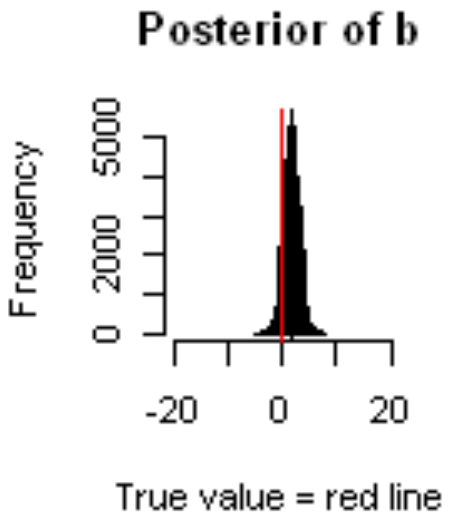
```
#source: https://theoreticalecology.wordpress.com/2010/09/17/metropolis-hastings-mcmc-in-r/

#Creating test data: we create some test data that will be used to fit our model.
#Let's assume a linear relationship between the predictor and the response variable,
#so we take a linear model and add some noise.

trueA <- 5
trueB <- 0
trueSd <- 10
sampleSize <- 31

# create independent x-values
x <- (-(sampleSize-1)/2):((sampleSize-1)/2)
# create dependent values according to ax + b + N(0,sd)
y <-  trueA * x + trueB + rnorm(n=sampleSize,mean=0,sd=trueSd)

plot(x,y, main="Test Data")
#The likelihood is the probability (density) with which we would expect
 #the observed data to occur conditional on the parameters of the model
 #that we look at. So, given that our linear model y = b + a*x + N(0,sd)
 #takes the parameters (a, b, sd) as an input, we have to return the
 #probability of obtaining the test data above under this model
 #(this sounds more complicated as it is, as you see in the code,
 #we simply calculate the difference between predictions
 #y = b + a*x and the observed y, and then we have to look up
 #the probability densities (using dnorm) for such deviations to occur.


likelihood <- function(param){
    a = param[1]
    b = param[2]
    sd = param[3]

    pred = a*x + b
    singlelikelihoods = dnorm(y, mean = pred, sd = sd, log = T)
    sumll = sum(singlelikelihoods)
    return(sumll)
}

# Prior distribution
prior <- function(param){
```

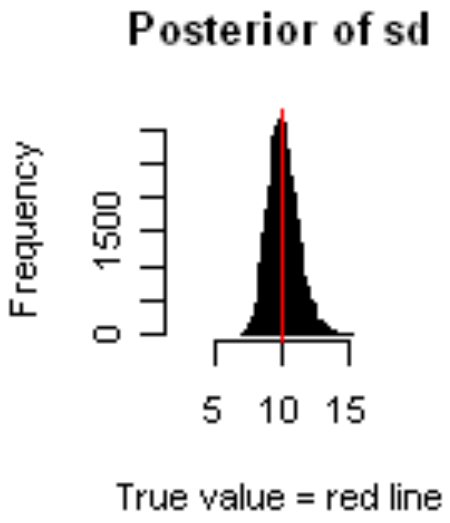
```
    a = param[1]
    b = param[2]
    sd = param[3]
    aprior = dunif(a, min=0, max=10, log = T)
    bprior = dnorm(b, sd = 5, log = T)
    sdprior = dunif(sd, min=0, max=30, log = T)
    return(aprior+bprior+sdprior)
}

#The product of prior and likelihood is the actual quantity the MCMC will
#be working on. This function is called the posterior (or to be exact,
#it's called the posterior after it's normalized, which the MCMC will do
#for us, but let's not be picky for the moment).
#Again, here we work with the sum because we work with logarithms.

posterior <- function(param){
   return (likelihood(param) + prior(param))
}

######## Metropolis algorithm ################

proposalfunction <- function(param){
    return(rnorm(3,mean = param, sd= c(0.1,0.5,0.3)))
}

run_metropolis_MCMC <- function(startvalue, iterations){
    chain = array(dim = c(iterations+1,3))
    chain[1,] = startvalue
    for (i in 1:iterations){
        proposal = proposalfunction(chain[i,])

        probab = exp(posterior(proposal) - posterior(chain[i,]))
        if (runif(1) < probab){
            chain[i+1,] = proposal
        }else{
            chain[i+1,] = chain[i,]
        }
    }
    return(chain)
}

startvalue = c(4,0,10)
chain = run_metropolis_MCMC(startvalue, 10000)

burnIn = 5000
acceptance = 1-mean(duplicated(chain[-(1:burnIn),]))

### Summary: ########################

par(mfrow = c(2,3))
hist(chain[-(1:burnIn),1],nclass=30, , main="Posterior of a", xlab="True value = red line" )
abline(v = mean(chain[-(1:burnIn),1]))
abline(v = trueA, col="red" )
hist(chain[-(1:burnIn),2],nclass=30, main="Posterior of b", xlab="True value = red line")
```

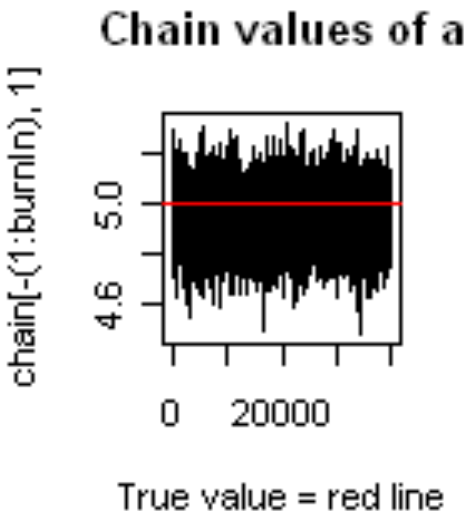
```
abline(v = mean(chain[-(1:burnIn),2]))
abline(v = trueB, col="red" )
hist(chain[-(1:burnIn),3],nclass=30, main="Posterior of sd", xlab="True value = red line")
abline(v = mean(chain[-(1:burnIn),3]) )
abline(v = trueSd, col="red" )
plot(chain[-(1:burnIn),1], type = "l", xlab="True value = red line" , main = "Chain values of a", )
abline(h = trueA, col="red" )
plot(chain[-(1:burnIn),2], type = "l", xlab="True value = red line" , main = "Chain values of b", )
abline(h = trueB, col="red" )
plot(chain[-(1:burnIn),3], type = "l", xlab="True value = red line" , main = "Chain values of sd",
abline(h = trueSd, col="red" )

# for comparison:
summary(lm(y~x))

# Example: plot the likelihood profile of the slope a
#slopevalues <- function(x){return(likelihood(c(x, trueB, trueSd)))}

#slopelikelihoods <- lapply(seq(3, 7, by=.05), slopevalues )
#plot (seq(3, 7, by=.05), slopelikelihoods , type="l", xlab = "values of slope parameter a", ylab =
```

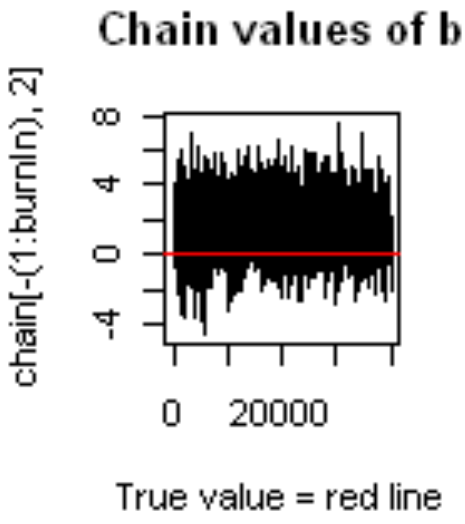
Note the posterior plots below.

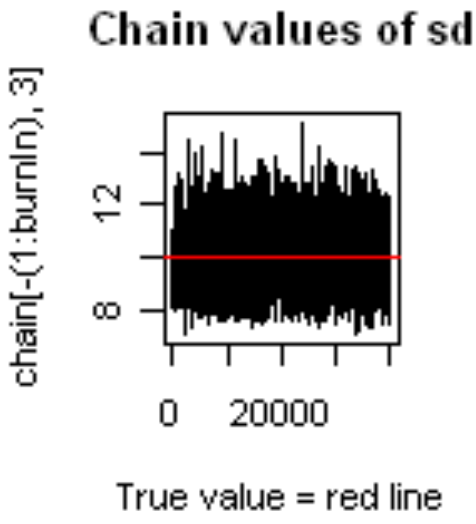












Further notes on easy visualisations with some related libraries:
https://theoreticalecology.wordpress.com/2011/12/09/mcmc-chain-analysis-and-convergence-diagnostics-with-coda-in-r/

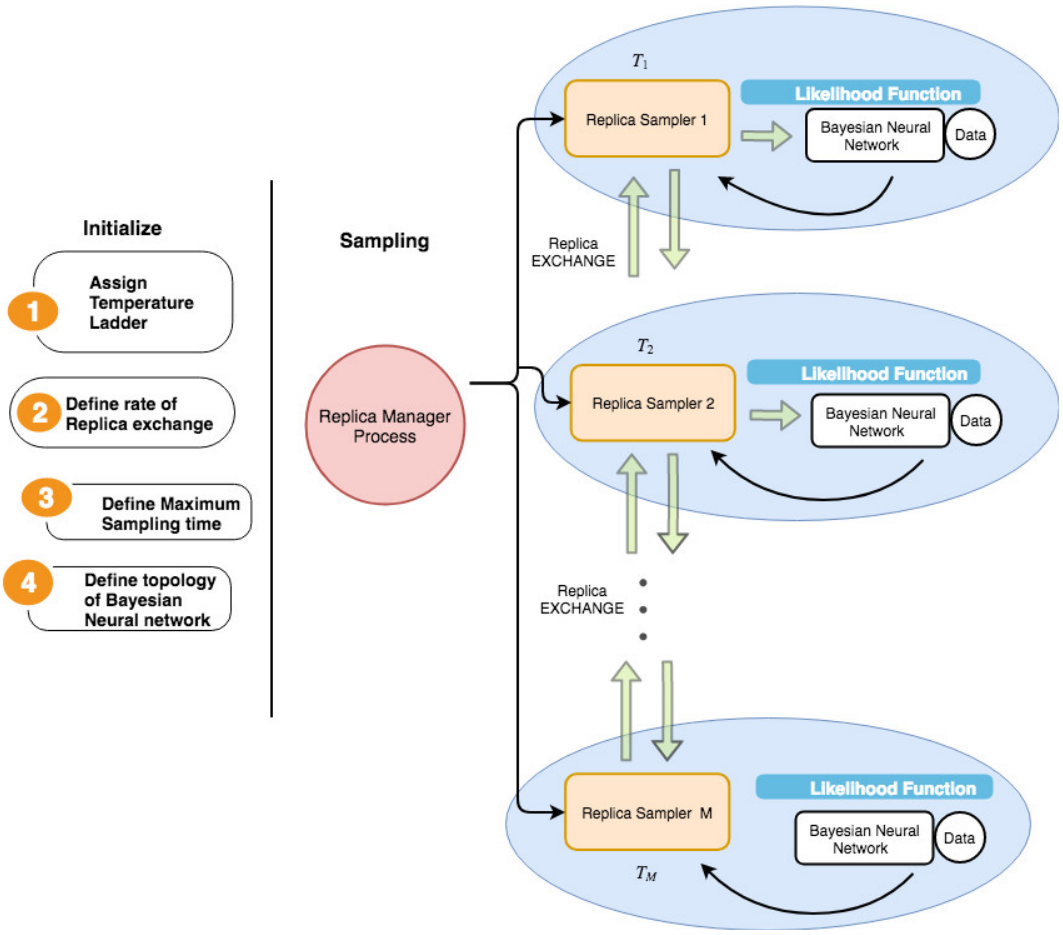
Advances in Bayesian neural networks

There are a number of MCMC variants: Metropolis-Hastings sampling, Gibbs Sampling, ensemble sampling, parallel tempering MCMC, adaptive MCMC, Hamiltonian Monte-Carlo, Langevin MCMC, Reversible Jump MCMC; shower we will only focus on **single-chain MCMC random-walk sampler** and **Langevin-gradient MCMC sampler** in this course. The other variates have special strengths and weakness suitable for different types of models. Note that Hamiltonian and Langevin MCMC require gradients and cannot be used in models where gradients are not present. Further details: https://www.cs.cmu.edu/~epxing/Class/10708-15/notes/10708_scribe_lecture17.pdf

Langevin-gradient parallel tempering for Bayesian neural learning

Abstract: Bayesian inference provides a rigorous approach for neural learning with knowledge representation via the posterior distribution that accounts for uncertainty quantification. Markov Chain Monte Carlo (MCMC) methods typically implement Bayesian inference by sampling from the posterior distribution. This not only provides point estimates of the weights, but the ability to propagate and quantify uncertainty in decision making. However, these techniques face challenges in convergence and scalability, particularly in settings with large datasets and neural network architectures. This paper addresses these challenges in two ways. First, parallel tempering MCMC sampling method is used to explore multiple modes of the posterior distribution and implemented in multi-core computing architecture. Second, we make within-chain sampling scheme more efficient by using Langevin gradient information for creating Metropolis–Hastings proposal distributions. We demonstrate the techniques using time series prediction and pattern classification applications. The results show that the method not only improves the computational time, but provides better decision making capabilities when compared to related methods.

Chandra R; Jain K; Deo RV; Cripps S, 2019, 'Langevin-gradient parallel tempering for Bayesian neural learning', *Neurocomputing*, vol. 359, pp. 315 - 326, http://dx.doi.org/10.1016/j.neucom.2019.05.082
https://github.com/rohitash-chandra/research/blob/master/2019/Chandra_LangevinNeurocom2019.pdf



References for Parallel tempering MCMC

1. Geyer, C. J., & Thompson, E. A. (1995). Annealing Markov chain Monte Carlo with applications to ancestral inference. *Journal of the American Statistical Association*, *90*(431), 909-920. https://www.tandfonline.com/doi/pdf/10.1080/01621459.1995.10476590?casa_token=Zp_XPAPu-O8AAAAA:QNOtizpmyhWnkzyHRHZIbS4xiPRX7HEZMTB09ZFmNFrygy6sCmlMrOz7ogvN8gpOQSwKJ7RNBAGSj84
2. Swendsen, R. H., & Wang, J. S. (1986). Replica Monte Carlo simulation of spin-glasses. *Physical review letters*, *57*(21), 2607. https://stat.duke.edu/~scs/Courses/Stat376/Papers/ClusterSampling/SwendsenWangPhysRevLett1986.pdf
3. Earl, D. J., & Deem, M. W. (2005). Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics*, *7*(23), 3910-3916. http://www.math.pitt.edu/~cbsg/Materials/Earl_ParallelTempering.pdf
4. Sambridge, M. (2014). A parallel tempering algorithm for probabilistic sampling and multimodal optimization. *Geophysical Journal International*, *196*(1), 357-374. http://rses.anu.edu.au/~malcolm/papers/pdf/Sambridge-GJI-2014.pdf (good tutorial)

Bayesian Optimisation: Bayesian optimisation and surrogate-assisted optimisation employs machine learning models to estimate the objective function using a surrogate model or acquisition function during optimisation which handy for expensive problems. The major advantage of Bayesian optimisation has been in reducing computational load by approximating the actual model with an acquisition function that is computationally cheaper. More information:

1. Brochu, E., Cora, V. M., & De Freitas, N. (2010). A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*. https://arxiv.org/pdf/1012.2599.pdf
2. Chandra R, Jain K, Kapoor A, Aman A, 'Surrogate-assisted parallel tempering for Bayesian neural learning'. Eng. Appl. Artif. Intell. 94: 103700 (2020) https://github.com/rohitash-chandra/research/blob/master/2020/Chandra_EngAppAI2020.pdf

Variational inference: Provides an analytical approximation to the posterior probability. Rather than sampling directly from the posterior, variational inference methods approximate it, making them applicable to problems where a large number of variables are present and where MCMC sampling becomes too computationally expensive. More information:

1. https://en.wikipedia.org/wiki/Variational_Bayesian_methods
2. http://www.robots.ox.ac.uk/~sjrob/Pubs/vbTutorialFinal.pdf

Research from our group:

1. Kapoor, A., Negi, A., Marshall, L., & Chandra, R. (2023). Cyclone trajectory and intensity prediction with uncertainty quantification using variational recurrent neural networks. *Environmental Modelling & Software*, *162*, 105654.
2. Chen, E., Andersen, M. S., & Chandra, R. (2024). Deep learning framework with Bayesian data imputation for modelling and forecasting groundwater levels. *Environmental Modelling & Software*, *178*, 106072.
3. https://arxiv.org/html/2410.01847v2

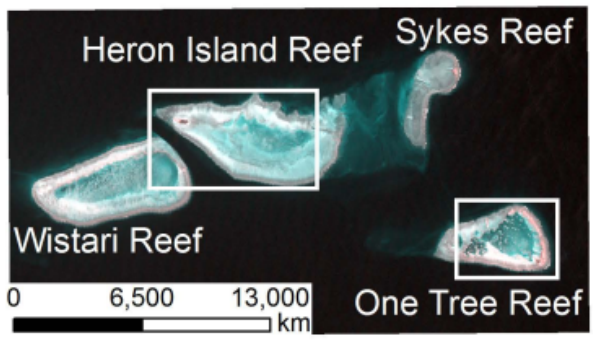
Applications of MCMC

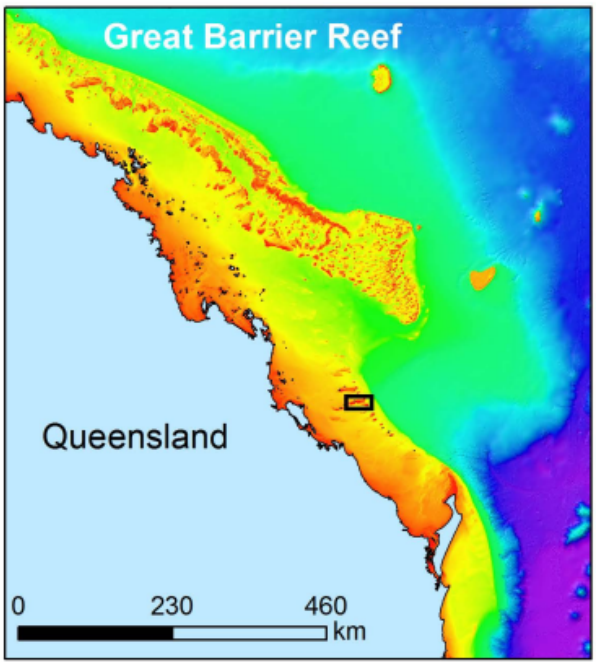
There are models out there where gradients are not present, such as those in Geosciences and environmental science. I spent my postdoc fellowship at the University of Sydney looking at them and these are some publications that look at models where no gradients are available using single-chain MCMC, Adaptive parallel tempering MCMC.

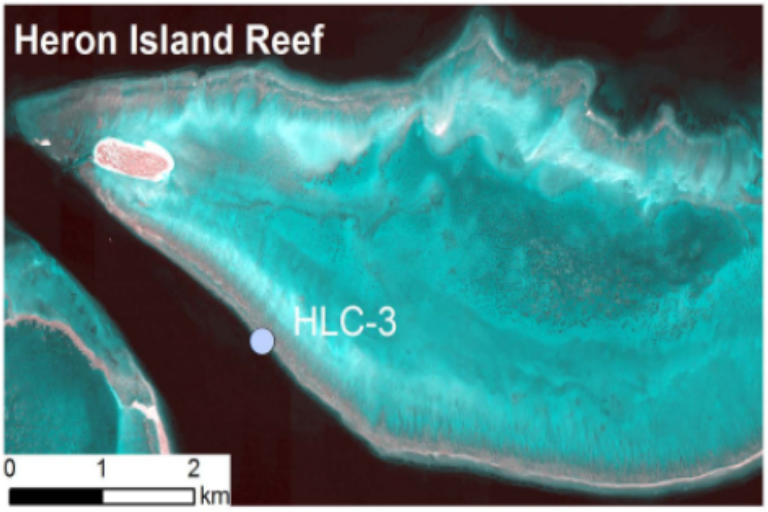
Abstract: Bayesreef: A Bayesian inference framework for modelling reef growth in response to environmental change and biological dynamics

Estimating the impact of environmental processes on vertical reef development in geological time is a very challenging task. *pyReef-Core* is a deterministic carbonate stratigraphic forward model designed to simulate the key biological and environmental processes that determine vertical reef accretion and assemblage changes in fossil reef drill cores. We present a Bayesian framework called *Bayesreef* for the estimation and uncertainty quantification of parameters in pyReef-Core that represent environmental conditions affecting the growth of coral assemblages in geological timescales. Weencounter multimodal posterior distributions and investigate the challenges of sampling using Markov chain Monte-Carlo (MCMC) methods, which includes parallel tempering MCMC. We use a synthetic reef-core to investigate fundamental issues and then apply the methodology to a selected reef-core from the *Great Barrier Reef* in Australia. The results show that Bayesreef accurately estimates and provides uncertainty quantification of the selected parameters that represent environment and ecological conditions in *pyReef-Core*. Bayesreef provides insights into the complex posterior distributions of the parameters in *pyReef-Core*, which provides the groundwork for future research in this area.

Pall J; Chandra R; Azam D; Salles T; Webster JM; Scalzo R; Cripps S, 2020, 'Bayesreef: A Bayesian inference framework for modelling reef growth in response to environmental change and biological dynamics', *Environmental Modelling and Software*, vol. 125, pp. 104610 - 104610, http://dx.doi.org/10.1016/j.envsoft.2019.104610



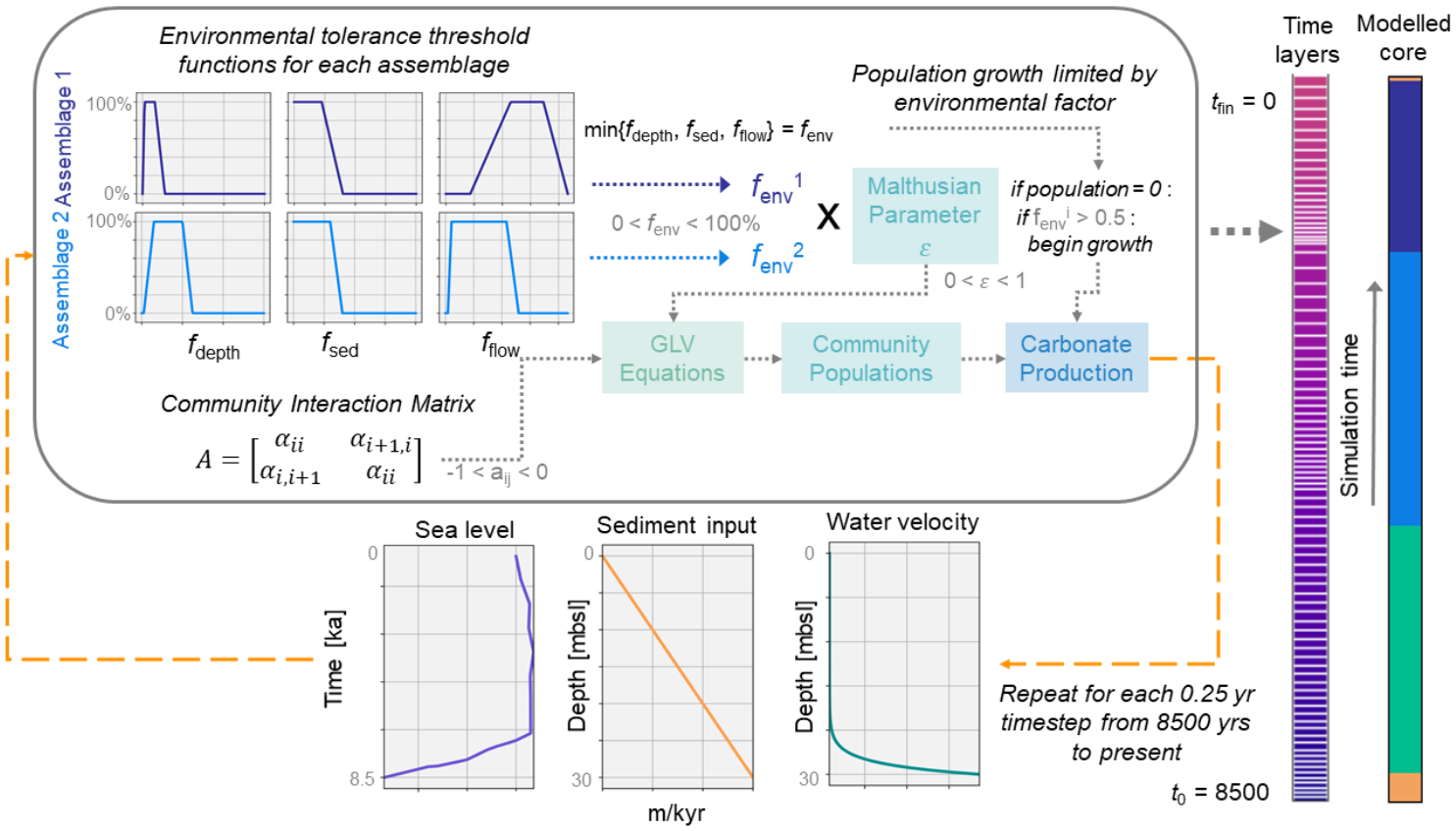




Salas-Saavedra et al.

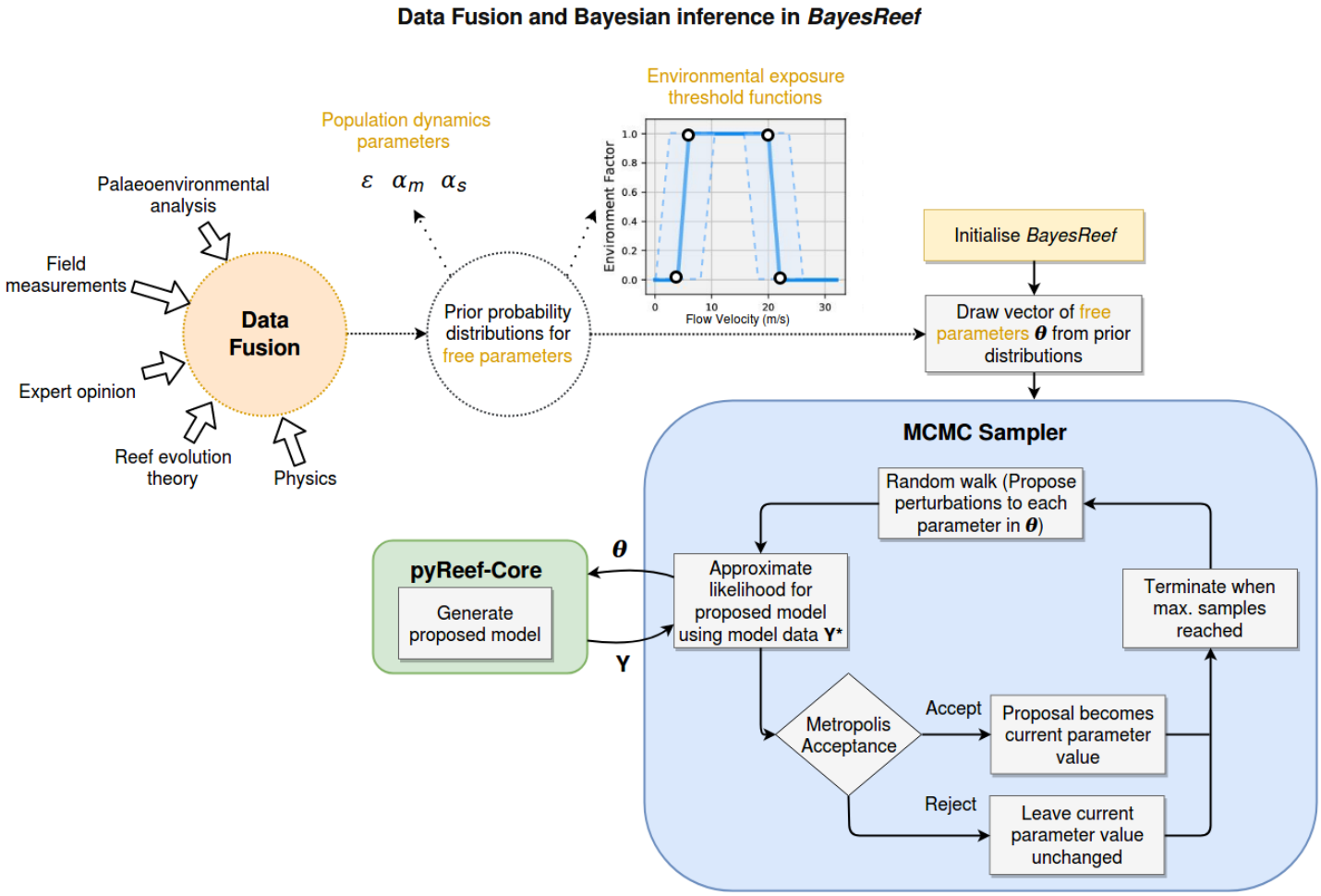
Coordinate System: GDA 1994
Data Sources:
GBRMPA:GBR islands, reefs and cays.
USyd: Geocoastal Research Group, unpublished.
Beaman, R.J. (2010). 3DGBR: A high-resolution depth model for the Great Barrier Reef and Coral Sea. MTSRF Project 2.5i.1a:12.

Above figure gives the local for the study area in the great barrier reef.



Above figure gives the schematic for py-Reef-Core model that simulates vertical reef development over thousands of years.

Below we refer to an MCMC framework for estimating parameters in a geological reef-core model (py-Reef-Core):

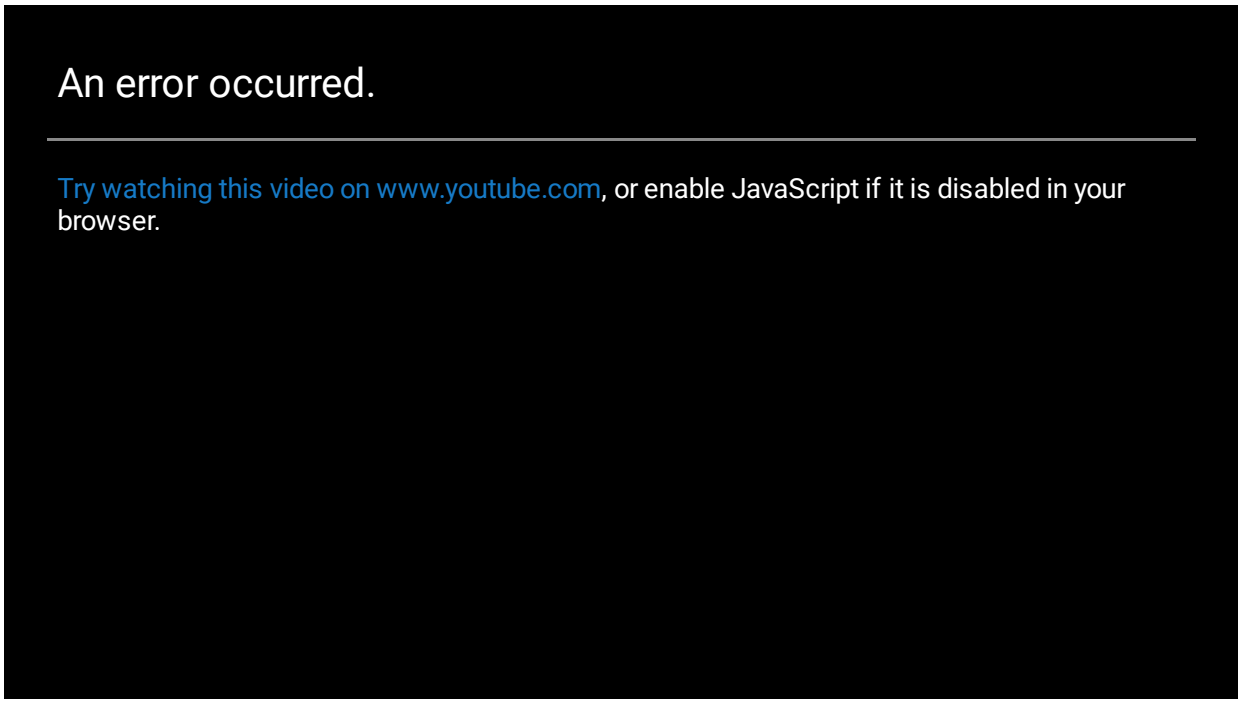


Bayeslands: A Bayesian inference approach for parameter uncertainty quantification in Badlands

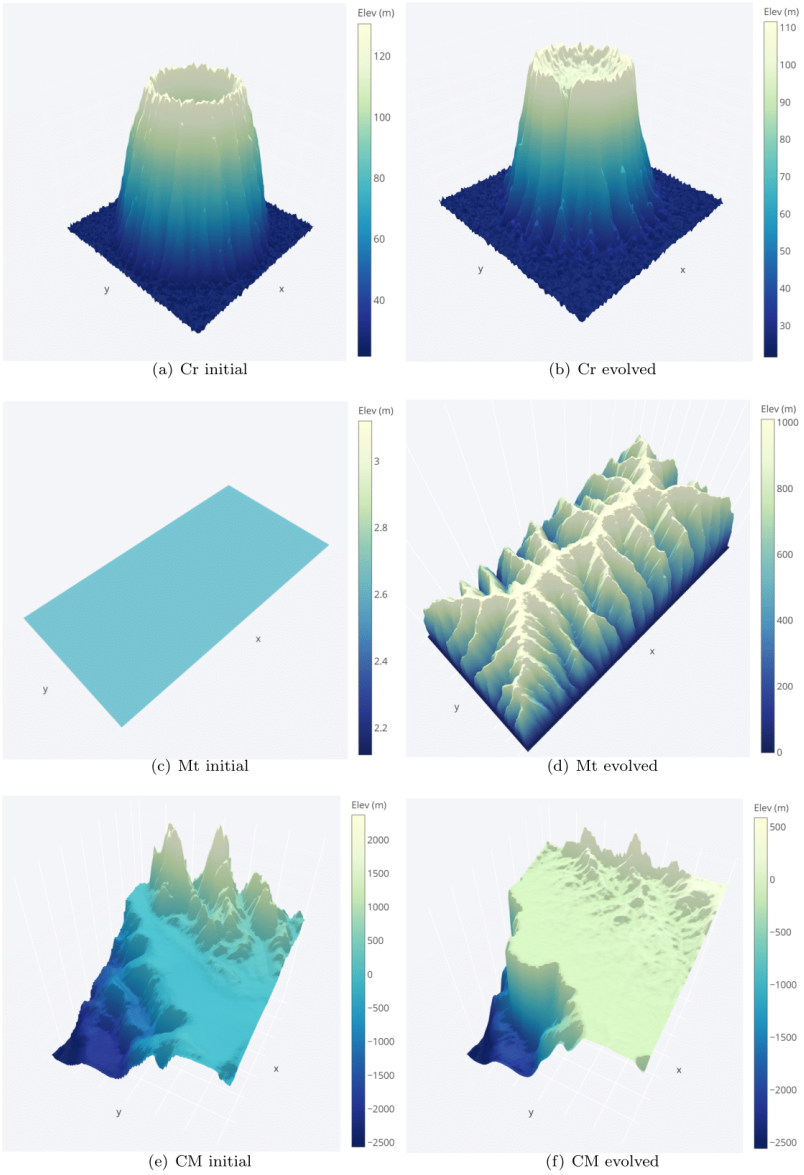
Abstract: Bayesian inference provides a rigorous methodology for estimation and uncertainty quantification of unknown parameters in geophysical forward models. Badlands is a landscape evolution model that simulates topography development at various space and time scales. Badlands consists of a number of geophysical parameters that needs estimation with appropriate uncertainty quantification; given the observed present-day ground truth such as surface topography and the stratigraphy of sediment deposition through time. The inference of the unknown parameters is challenging due to the scarcity of data, sensitivity of the parameter setting, and complexity of the model. In this paper, we take a Bayesian approach to provide inference using Markov chain Monte Carlo sampling (MCMC). We present *Bayeslands*; a Bayesian framework for Badlands that fuses information obtained from complex forward models with observational data and prior knowledge. As a proof-of-concept, we consider a synthetic and real-world topography with two parameters for Bayeslands; namely, precipitation and erodibility. We demonstrate the challenge in sampling irregular and multi-modal posterior distributions using a likelihood surface that has a range of sub-optimal modes. The results of the experiments show that Bayeslands yields a promising distribution of the selected Badlands parameters.

Chandra R; Azam D; Müller RD; Salles T; Cripps S, 2019, 'Bayeslands: A Bayesian inference approach for parameter uncertainty quantification in Badlands', *Computers and Geosciences*, vol. 131, pp. 89 - 101, http://dx.doi.org/10.1016/j.cageo.2019.06.012 https://github.com/rohitash-chandra/research/blob/master/2019/Chandra_Bayeslands_Computers-and-Geoscience.pdf

Here is an example of simulation from a landscape evolution model:



Below are examples of some test problems.



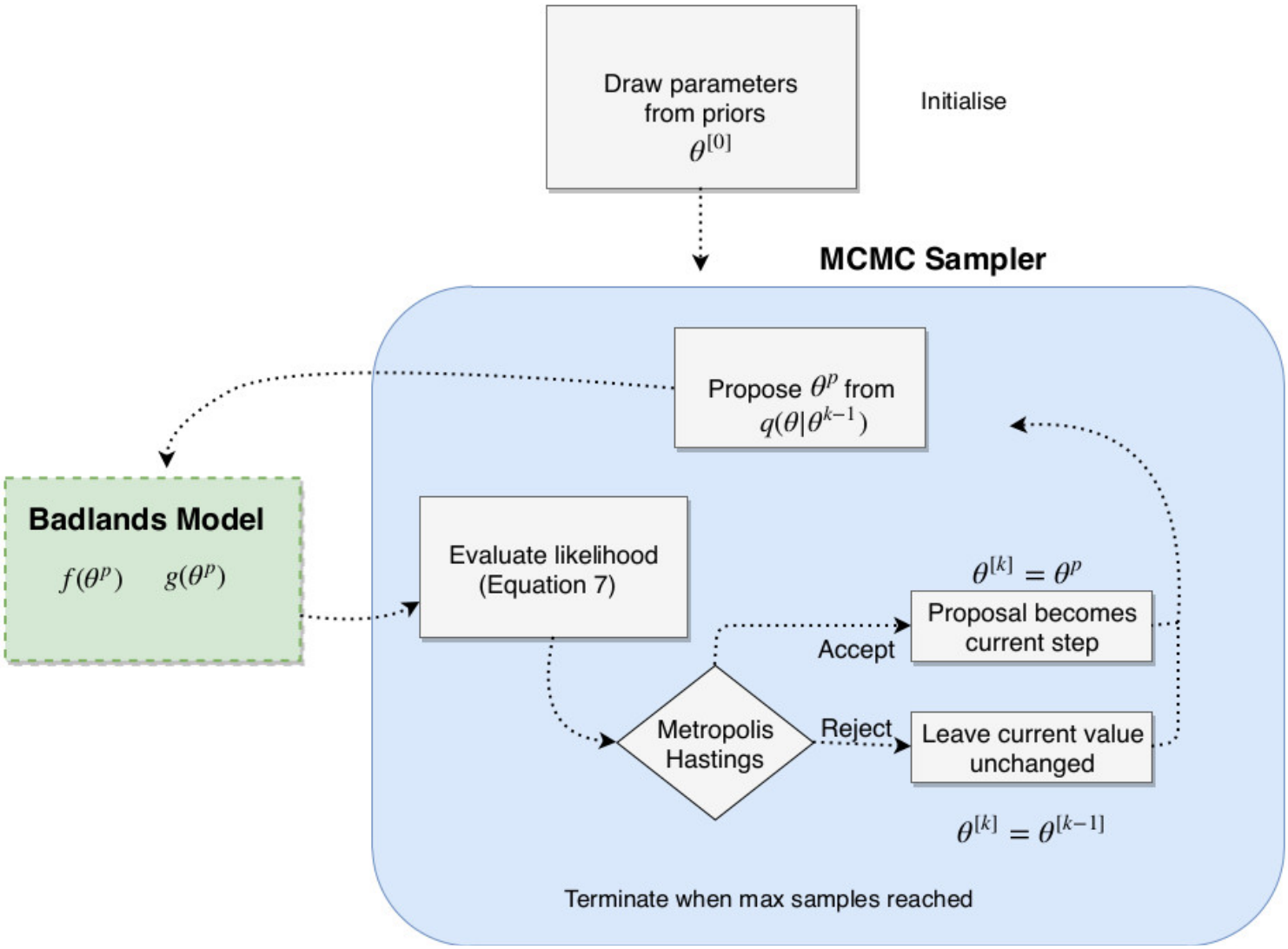
(a) Cr initial (b) Cr evolved

(c) Mt initial (d) Mt evolved

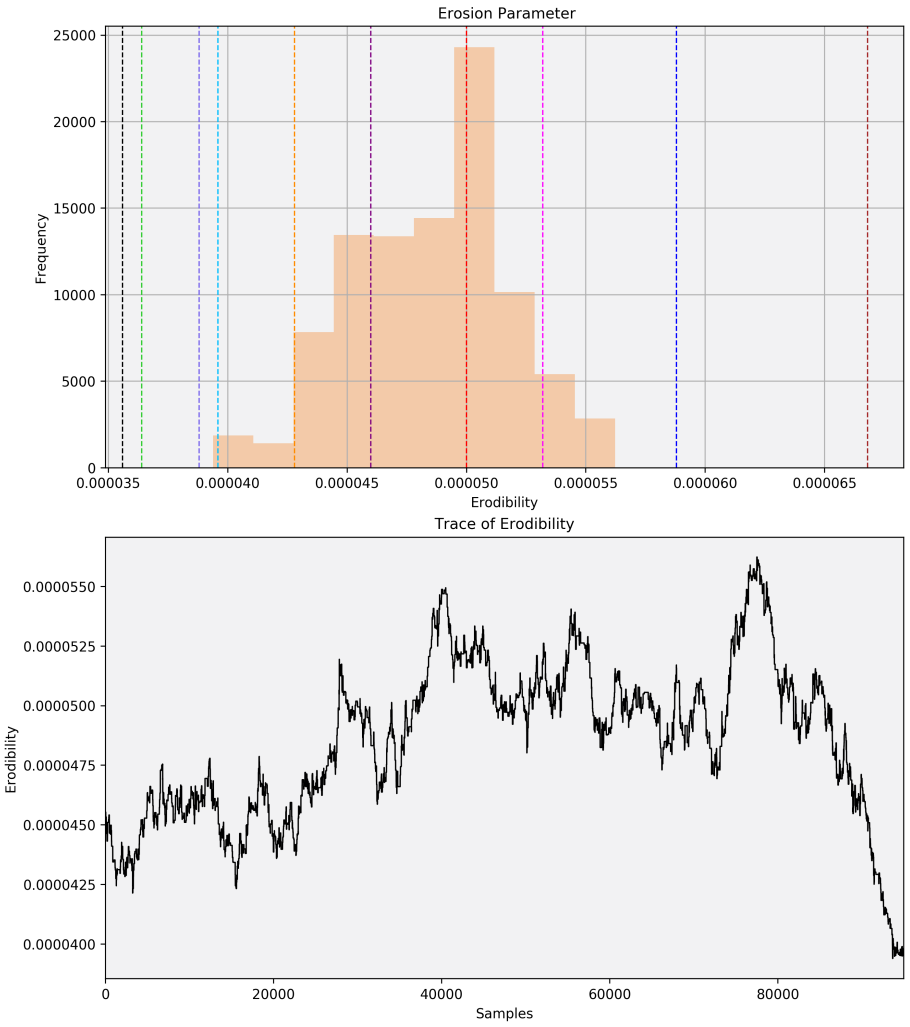
(e) CM initial (f) CM evolved

Figure above shows three different test problems, with initial and final topographies.

Next we see an example of MCMC for landscape evolution model (Badlands)



Source: Chandra R; Azam D; Müller RD; Salles T; Cripps S, 2019, 'Bayeslands: A Bayesian inference approach for parameter uncertainty quantification in Badlands', *Computers and Geosciences*, vol. 131, pp. 89 - 101, http://dx.doi.org/10.1016/j.cageo.2019.06.012



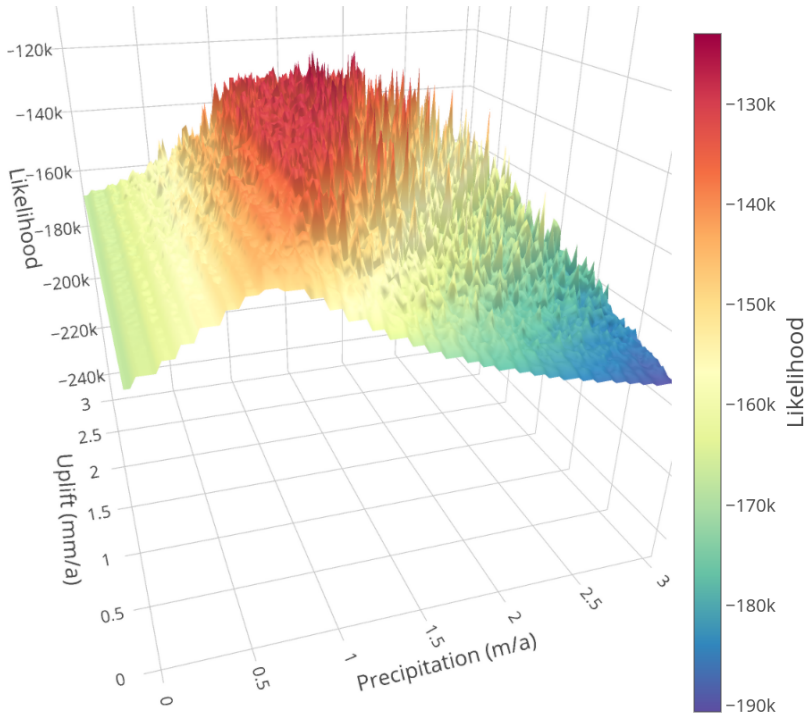
Posterior and trace plot of a parameter called erodibility in the Badlands model by Bayeslands framework.

Multicore Parallel Tempering Bayeslands for Basin and Landscape Evolution

Abstract: The Bayesian paradigm is becoming an increasingly popular framework for estimation and uncertainty quantification of unknown parameters in geophysical inversion problems. Badlands is a *landscape evolution model* for simulating topography evolution at a broad range of spatial and temporal scales. Our previous work presented Bayeslands that used the Bayesian inference for estimating unknown parameters in the Badlands model using Markov chain Monte Carlo sampling. Bayeslands faced challenges in terms of computational issues and convergence due to multimodal posterior distributions. Parallel tempering is an advanced Markov chain Monte Carlo method suited for irregular and multimodal posterior distributions. In this paper, we extend Bayeslands using parallel tempering with high-performance computing to address previous limitations in Bayeslands. Our results show that parallel tempering Bayeslands not only reduces the computation time, but also provides an improvement in sampling multimodal posterior distributions, which motivates future application to continental scale landscape evolution models.

Chandra R; Müller RD; Azam D; Deo R; Butterworth N; Salles T; Cripps S, 2019, 'Multicore Parallel Tempering Bayeslands for Basin and Landscape Evolution', *Geochemistry, Geophysics, Geosystems*, vol. 20, pp. 5082 - 5104, http://dx.doi.org/10.1029/2019GC008465 https://github.com/rohitash-chandra/research/blob/master/2019/Bayeslands_GCubed2019.pdf

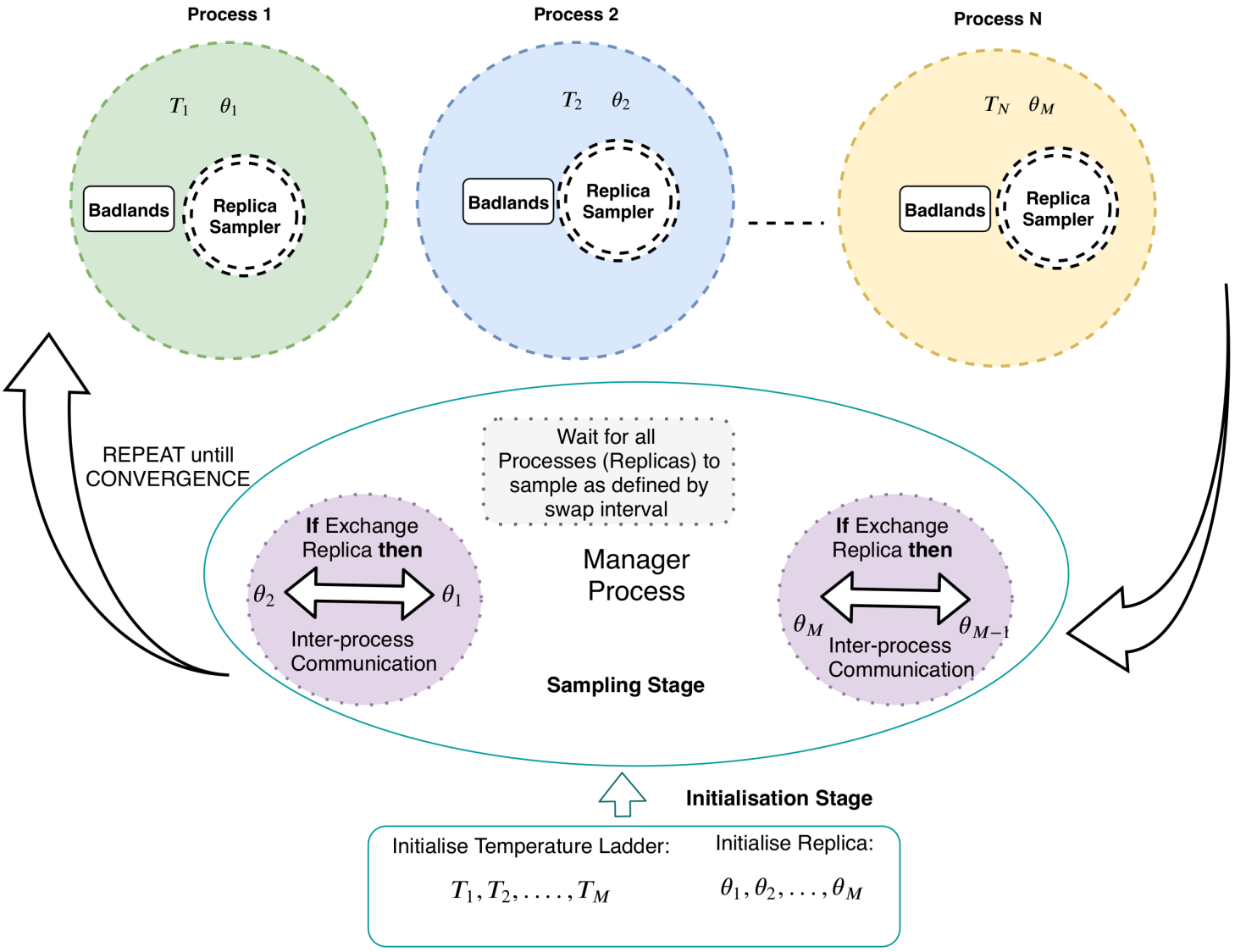
The implementation of Bayesian inference faces challenges as the number of parameters in models became larger. Complex and multimodal posterior as shown below gives further challenges.



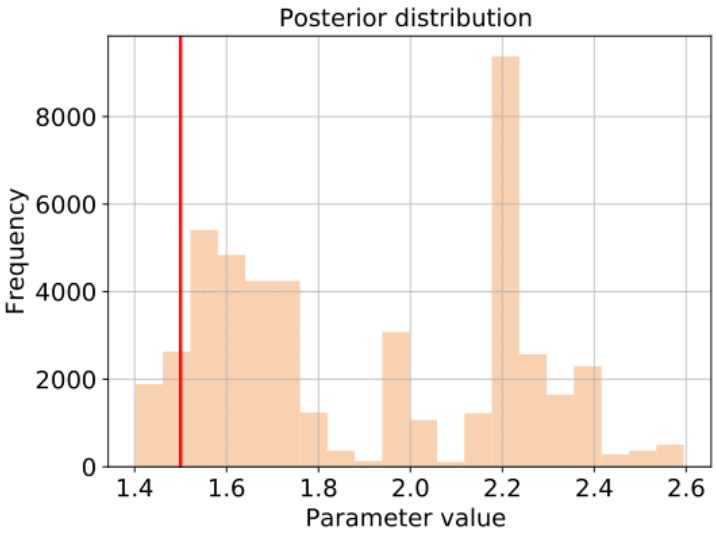
The figure above shows complex posterior (likelihood surface) for two selected parameters in Badlands model.

Furthermore, the problem becomes more challenging when the model evaluations take considerable computational time; in such case, it could take takes weeks or months for drawing thousands of samples on single-core processing units. Hence, it is important to employ sampling methods that can utilize parallel computing.

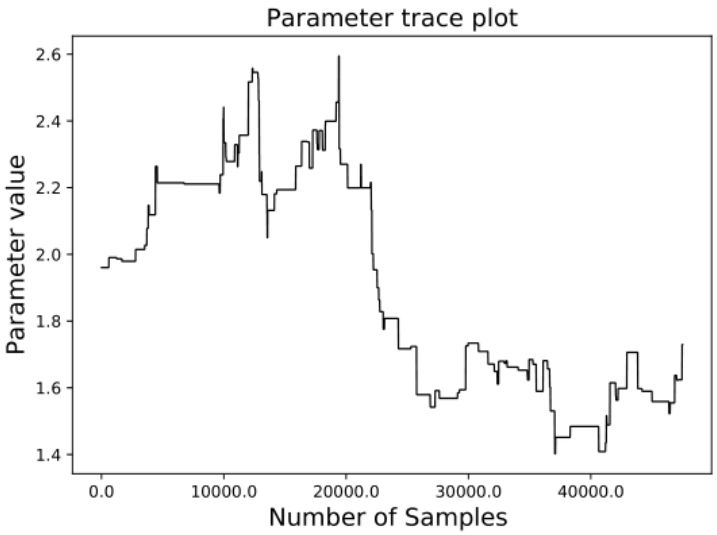
Below is an example where parallel computing is used with parallel tempering MCMC for Bayeslands.



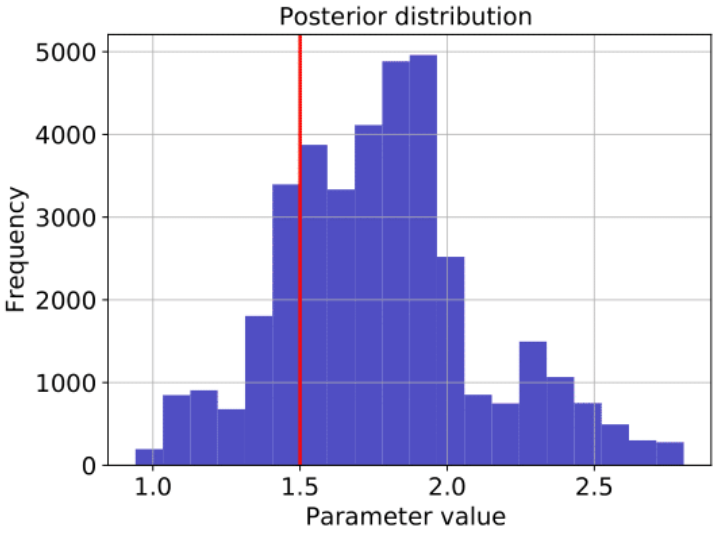
The figure above shows how different replicas are executed in parallel cores.



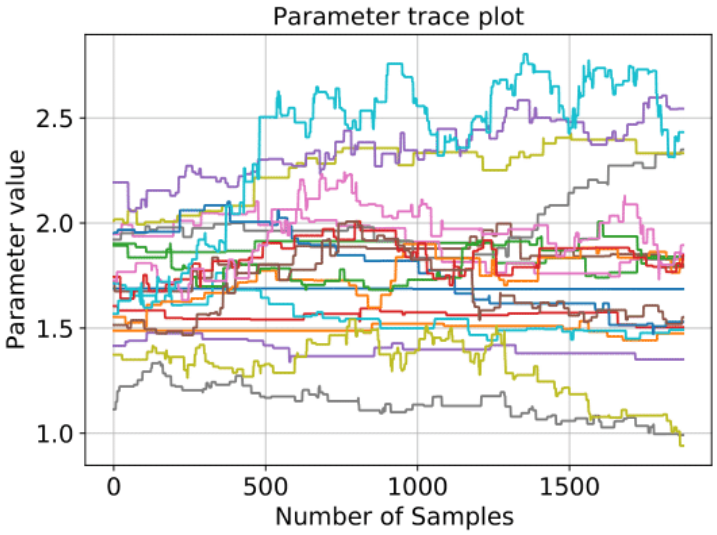
(a) Posterior (SC- Bayeslands)



(b) Trace-plot(SC- Bayeslands)

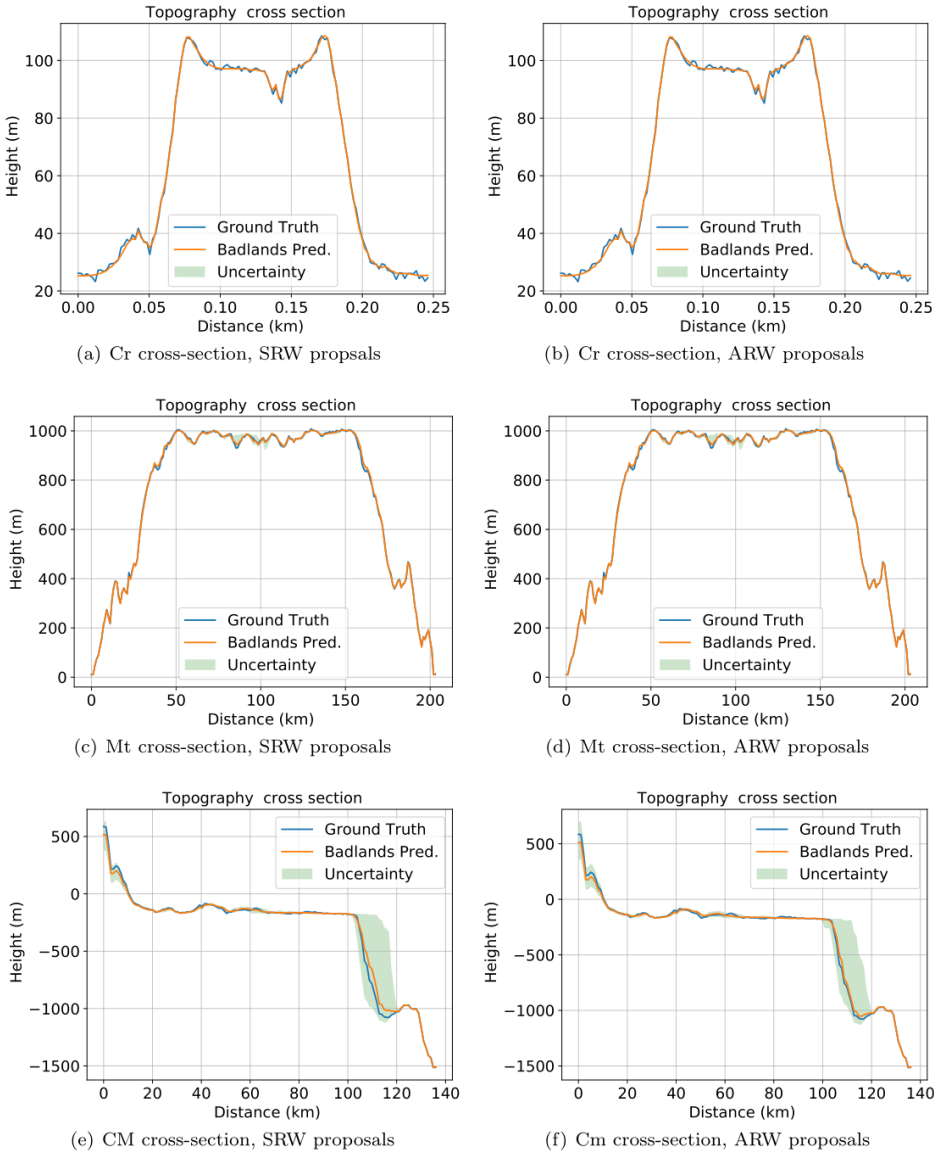


(c) Posterior (PT- Bayeslands)



(d) Trace-plot (PT- Bayeslands)

The figure above shows results that compare single-chain MCMC with parallel tempering MCMC Bayeslands.



(a) Cr cross-section, SRW propsals

(b) Cr cross-section, ARW proposals

(c) Mt cross-section, SRW proposals

(d) Mt cross-section, ARW proposals

(e) CM cross-section, SRW proposals

(f) Cm cross-section, ARW proposals

The figure above shows topography prediction results from Bayeslands.

Other applications

1. Olierook HKH; Scalzo R; Kohn D; Chandra R; Farahbakhsh E; Clark C; Reddy SM; Müller RD, 2020, 'Bayesian geological and geophysical data fusion for the construction and uncertainty quantification of 3D geological models', *Geoscience Frontiers*, http://dx.doi.org/10.1016/j.gsf.2020.04.015
2. Scalzo R; Kohn D; Olierook H; Houseman G; Chandra R; Girolami M; Cripps S, 2019, 'Efficiency and robustness in Monte Carlo sampling for 3-D geophysical inversions with Obsidian v0.1.2: Setting up for success', *Geoscientific Model Development*, vol. 12, pp. 2941 - 2960, http://dx.doi.org/10.5194/gmd-12-2941-2019

MCMC libraries

1. **PyMC3**: A comprehensive python-based statistics and machine learning library featuring MCMC methods, probability distributions, Gaussian process, variational inferences and machine learning libraries via Theano https://docs.pymc.io/ basic tutorial: https://docs.pymc.io/notebooks/api_quickstart.html
2. **Stan:** Computational statistics library available in Python and R: https://cran.r-project.org/web/packages/rstan/vignettes/rstan.html https://mc-stan.org/users/interfaces/rstan
3. **emcee:** emcee is a Python implementation of Goodman & Weare's Affine Invariant Markov chain Monte Carlo (MCMC) Ensemble sampler. It features convergence diagnosis such as autocorrelation. More information: https://emcee.readthedocs.io/en/stable/

References

1. Foreman-Mackey, D., Hogg, D. W., Lang, D., & Goodman, J. (2013). emcee: the MCMC hammer. *Publications of the Astronomical Society of the Pacific*, *125*(925), 306. https://iopscience.iop.org/article/10.1086/670067/pdf
2. Goodman, J., & Weare, J. (2010). Ensemble samplers with affine invariance. *Communications in applied mathematics and computational science*, *5*(1), 65-80. https://msp.org/camcos/2010/5-1/camcos-v5-n1-p04-s.pdf
3. Shi, J., Chen, J., Zhu, J., Sun, S., Luo, Y., Gu, Y., & Zhou, Y. (2017). Zhusuan: A library for bayesian deep learning. *arXiv preprint arXiv:1709.05870*. https://arxiv.org/abs/1709.05870

Resources: Bayesian Logistic Regression with MCMC

- https://docs.pymc.io/notebooks/GLM-logistic.html
- http://barnesanalytics.com/bayesian-logistic-regression-in-python-using-pymc3
- http://people.duke.edu/~ccc14/sta-663-2018/notebooks/S11A_PyMC3.html

```
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from scipy.stats import norm

sns.set_style('white')
sns.set_context('talk')

np.random.seed(123)

data = np.random.randn(200)

#histogram using seaborn (sns)
ax = plt.subplot()
sns.distplot(data, kde=True, ax=ax) # smooth using KDE: https://en.wikipedia.org/wiki/Kernel_densit
_ = ax.set(title='Histogram of observed data', xlabel='x', ylabel='frequency');
plt.tight_layout()
plt.savefig('histo_seaborn.png')
plt.clf()

#plot using matplotlib
# An "interface" to matplotlib.axes.Axes.hist() method
n, bins, patches = plt.hist(x=data, bins='auto', color='#0504aa', alpha=0.7, rwidth=0.85)
plt.grid(axis='y', alpha=0.75)
plt.xlabel('x')
plt.ylabel('frequency')
plt.tight_layout()
plt.savefig('histo_matplotlib.png')
plt.clf()

def calc_posterior_analytical(data, x, mu_0, sigma_0):
    sigma = 1.
    n = len(data)
    mu_post = (mu_0 / sigma_0**2 + data.sum() / sigma**2) / (1. / sigma_0**2 + n / sigma**2)
    sigma_post = (1. / sigma_0**2 + n / sigma**2)**-1
```

```
    return norm(mu_post, np.sqrt(sigma_post)).pdf(x)

ax = plt.subplot()
x = np.linspace(-1, 1, 50)

posterior_analytical = calc_posterior_analytical(data, x, 0., 1.)
print(posterior_analytical)
ax.plot(x, posterior_analytical)
ax.set(xlabel='mu', ylabel='belief', title='Analytical posterior');
sns.despine()
plt.tight_layout()
plt.savefig('histo_analytical.png')
plt.clf()
```

```
#source https://rdrr.io/cran/MCMCpack/man/MCMClogit.html


library(MCMCpack)

   ## Not run:
   ## default improper uniform prior
   data(birthwt)
   posterior <- MCMClogit(low~age+as.factor(race)+smoke, data=birthwt)
   plot(posterior)
   summary(posterior)


   ## multivariate normal prior
   data(birthwt)
   posterior <- MCMClogit(low~age+as.factor(race)+smoke, b0=0, B0=.001,
                          data=birthwt)
   plot(posterior)
   summary(posterior)


   ## user-defined independent Cauchy prior
   logpriorfun <- function(beta){
     sum(dcauchy(beta, log=TRUE))
   }

   posterior <- MCMClogit(low~age+as.factor(race)+smoke,
                          data=birthwt, user.prior.density=logpriorfun,
                          logfun=TRUE)
   plot(posterior)
   summary(posterior)


   ## user-defined independent Cauchy prior with additional args
   logpriorfun <- function(beta, location, scale){
```

```
    sum(dcauchy(beta, location, scale, log=TRUE))
  }

  posterior <- MCMClogit(low~age+as.factor(race)+smoke,
                         data=birthwt, user.prior.density=logpriorfun,
                         logfun=TRUE, location=0, scale=10)
  plot(posterior)
  summary(posterior)



## End(Not run)
```

Exercise 5.2

R Challenge

- Apply Bayesian neural networks for one-step-ahead COVID-19 prediction - prediction for Chine/Australia/USA/India.

Python Challenge

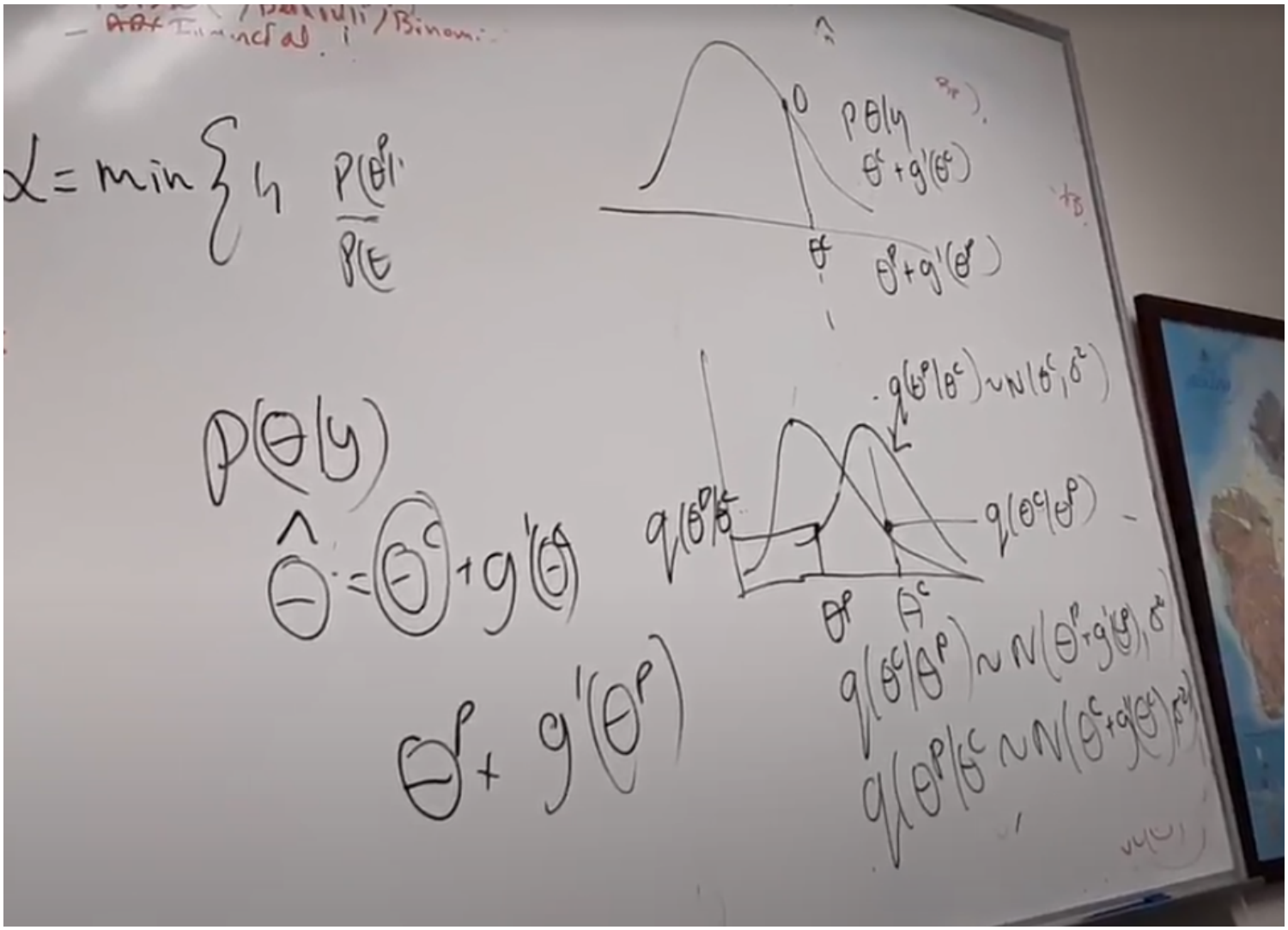
- Apply Bayesian neural networks for one-step-ahead COVID-19 prediction - prediction for Chine/Australia/USA/India.
- Apply Bayesian neural networks to UCI machine learning repository regression problems.
- Apply Bayesian neural networks to UCI machine learning repository classification problems.

Further challenge

Explore MCMC libraries such as Stan (R and Python) and PyMC3 and compare your results with the above.

Detailed balance visual

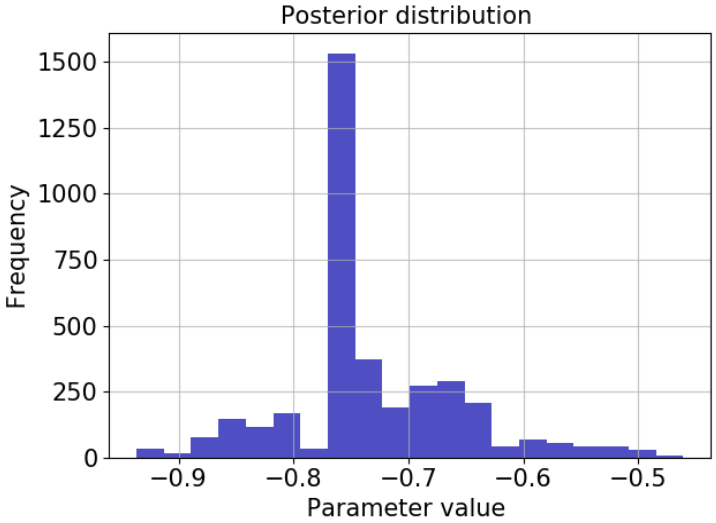
The diagram below will be redrawn to explain it further.

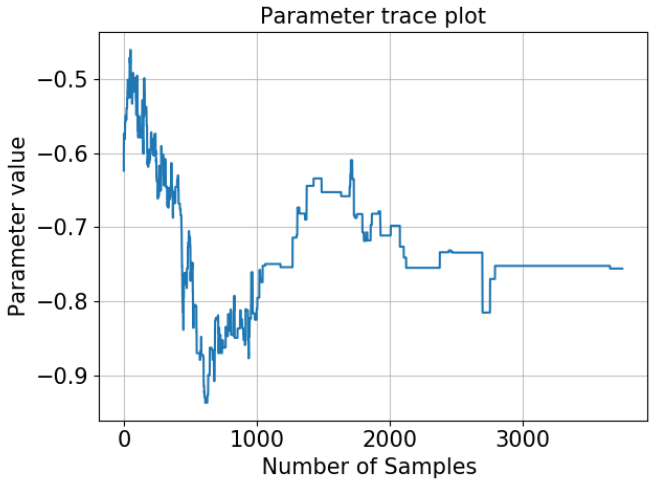


Bayesian linear regression - multiple outputs

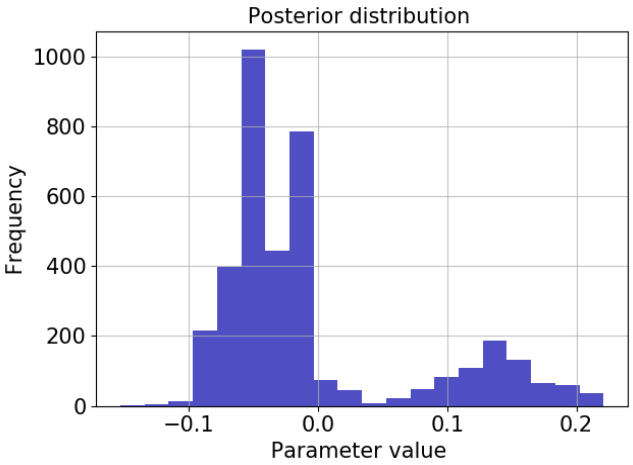
Bayesian linear regression for a single step ahead (Sunspot time series) and multi-step ahead time series prediction (MMM stock market). Below you can see trace-plot and histogram of the posterior distribution. Note that Bayesian multinomial regression can be developed using this code for multi-class classification problems.

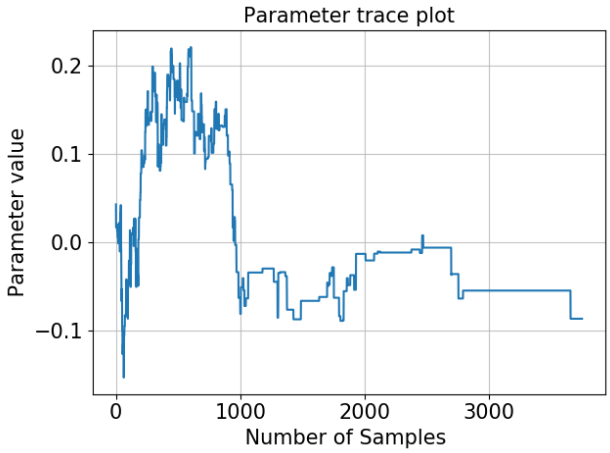
We note that in multi-step time series prediction, 5 step-ahead would mean 5 output neurons. We use sigmoid units in the output layer. Note that gradients are not used and you can compare the results with SGD which is present in the code.





Another selected parameter from the model shown below.





Other posterior visuals: https://github.com/rohitash-chandra/Bayesianlogisticreg_multioutputs/tree/master/posterior

You can execute the code and uncomment some of the print statements to understand.

Weekly coding space

Weekly coding space