Hamiltonian Monte Carlo in Bayesian Neural Networks

Juan Maroñas
jmaronasm@gmail.com

PRHLT Research Center,
Universitat Politècnica de València

April 2019

1 Introduction

In this document I show how can we evaluate the predictive distribution in a classification scenario when using Neural Networks to parameterize the likelihood model in a Bayesian setting. In this case I will be using Hamiltonian Monte Carlo [2, 3] to get samples from the posterior distribution over the model parameters. For an alternative and shorter explanation see [1] chapter 11 section 5. Rather than a technical report this document aims at briefly explain the different parts of the model in order to be able to follow and understand the provided implementation. I assume the reader knows what Markov Chain Monte Carlo (MCMC) algorithms (in particular Hamiltonian Monte Carlo) are, what a Bayesian setting is and how can we benefit from reasoning under it, and why do we need MCMC to draw samples from the posterior in this particular setting.

The key points of Markov Chain Hamiltonian Monte Carlo is take the Hamiltonian function from the mechanical physics and make an analogy with unormalized probability distributions. Once this is done, we "move" in the mechanical system to draw a new sample from the desired distribution. Because we have to deal system described using partial derivatives we need to discretize the system (computer can only do discrete steps). To correct the errors this integration method can produce, we use a Metropolis-Hasting correction on the proposed sample. The discretization is done using the leapfrog numerical method, see https://en.wikipedia.org/wiki/Leapfrog_integration

2 Model

For simplicity, I assume that the prior over the parameters of a neural network w (these includes any parameters: bias, weight, γ and β from Batch Normal-

ization...) follows a factorized standard normal distribution:

$$p(w) = \prod_{i=1}^{|W|} \frac{1}{\sqrt{2 \cdot \pi}} \exp\big(-\frac{1}{2} w_i^2\big) \tag{1}$$

With partial log derivative w.r.t to weight w_k given by:

$$\frac{\partial \log p(w)}{\partial w_k} = -w_k \tag{2}$$

Note that computing this derivative is straightforward just by noting that the terms of the log product for values of $k' \neq k$ does not contribute to the derivative of w_k, and that many terms are constant w.r.t w.

On the other side, in a classification scenario where t denotes the class target of our feature x, the posterior probability of the weights given a set of observations $\mathcal{O} = \{(x_i, t_i)\}_{i=1}^N$ is given by:

$$p(w|\mathcal{O}) = \frac{p(\mathcal{O}|w) \cdot p(w)}{Z} \tag{3}$$

where Z is a normalization factor which is computed by marginalizing the numerator over w. Our set of observations is assumed to be iid, thus we approximate the likelihood term as:

$$p(\mathcal{O}|w) = \prod_{i=1}^{N} p(t_i|x_i, w) \tag{4}$$

where we assume that the input distribution $p(x_i|w) = 1$ is not modelled. Our final posterior distribution takes the form:

$$p(w|\mathcal{O}) = \frac{\prod_{i=1}^{N} p(t_i|x_i, w) \cdot p(w)}{Z} \tag{5}$$

If we assume that the terms of the likelihood $p(t_i|x_i, w)$ follow a Bernoulli (for the binary classification problem) or a categorical (for the multiclass one), then the derivative of the log likelihood of the posterior can be expressed as:

$$\frac{\partial \log p(w|\mathcal{O})}{\partial w_k} = \frac{\partial \sum_{i=1}^{N} -\mathrm{CE}(x_i, t_i, w)}{\partial w_k} + \frac{\partial \log p(w)}{\partial w_k} \tag{6}$$

Where CE is the cross entropy function given by $-t_i \log f_w(x_i)$, $f_w()$ is the model parameterized by w (in this particular case will be a neural network)[1]. It can be clearly seen that the first term corresponds to the gradients used in

[1]Note that this is a particularization for multiclass problem. If the problem is binary and your model just outputs one value, then $\mathrm{CE} = -t \log(f_w(x)) - (1-t)\log(1 - f_w(x))$ is the binary cross entropy

the update rule of stochastic gradient guided methods, and that can be easily computed using automatic differentiation packages like autograd from PyTorch. The second term has already been computed and $-\frac{\partial \log Z}{\partial w_k} = 0$ as it has no dependence on the parameter.

3 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) is a Markov Chain Monte Carlo used to sample from a desired distribution, in this case $p(w|\mathcal{O})$. HMC uses an alternative random variable, which is call the momentum random variable denoted by m with density $p(m)$. The good point is that the joint probability defined in HMC factorizes, and thus this auxiliary variable does not influence the samples from the desired posterior. In this case, my model is fully specified by the next hamiltonian function:

$$H(m, w) = U(w) + K(m) \tag{7}$$

where $U(w) = -\log \prod_{i=1}^{N} p(t_i|x_i, w) \cdot p(w)$ and $K(m) = -\frac{1}{2} m^T m$. The joint probability distribution defined by using the Hamiltonian system is given by:

$$p(w, m) \propto \exp(-H(m, w)) = \exp(-U(w) - K(m)) \tag{8}$$

Under this definition it can be easily check that $p(w, m) = p(w) \cdot p(m)$, that $p(w|\mathcal{O}) = p(w) \propto \exp(-U(w))$ and that $p(m) = \mathcal{N}(0, I)$. The final algorithm (applied concurrently to each weight in the network) states as:[2]

[2]Note that the exponential of the kinetic function is the same as the standard normal, we only require the factor $(\sqrt{2\pi})^{-1}$ make the kinetic function a probability distribution. For $U(w)$ the normalization factor is given by Z. MCMC algorithms can sample from unormalized distributions [1] thus we do not need to compute these values

Input: ϵ,T,L # Leapfrog step size, number of MCMC and Leapfrog steps
Output: $w^* \sim p(w|\mathcal{O})$
Initialize: $w_0 \sim N(0, I)$ # initial value of your markov chain
for $t = 1$ *until* T **do**
  $m_0 \sim \mathcal{N}(0, I)$#*resample momentum variable*
  $m_L = m_0$
  $w_L = w_0$
  for $l = 1$ *until* L **do**
    $m' = m_L - \frac{\epsilon}{2} \frac{\partial p(w_L|\mathcal{O})}{\partial w_L}$
    $w_1 = w_0 + \epsilon \frac{\partial p(m)}{\partial m'}$
    $m_1 = m' - \frac{\epsilon}{2} \cdot \frac{\partial p(w_1|\mathcal{O})}{\partial w_1}$
    $m_L = m_1$
    $w_L = w_1$
  end
  #*Metropolis Hasting Correction*
  $u \sim \text{Uniform}[0, 1]$
  $\alpha = min(1, \exp(H(w_L, m_L) - H(w_0, m_0))$
  if $u < \alpha$, **then** $w_0 = w_L$;
  if $t = T$, **then return** $w^* = w_0$;
end

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[2] Radford M. Neal. Bayesian learning for neural networks. 1996.

[3] Radford M. Neal. Mcmc using hamiltonian dynamics. 2012.