Hamiltonian Monte Carlo

Bradley Gram-Hansen

July 18, 2017

0.1 Intro to HMC MCMC

Following Neal 2011 Chapter 5 Handbook of MCMCM

0.1.1 Metropolis-hastings algorithm

The Metropolis hastings algorithm is the work horse of Monte Carlo Markov Chain (MCMC) methods, It relies on a simple reject and accept criteria for determining whether or not we should proceed to some next state. If our state is accepted, but does not satisfy the full criteria then a probability is associated with that state, and if that probability exceeds the one sampled from a given cut off, usually determined by sampling from a uniform distribution then it is accepted, else rejected.

Algorithm 1 Metropolis-Hasting algorithm

1: $x^0 \sim P_0(x)$ $\triangleright$ Where $p(x)$ is the proposed initial distribution
2: **for** $s = 0, 1, 2, \ldots$ **do**
3:     $x \leftarrow x^s$
4:     $x^{'} \sim Q(x^{'}|x)$
5:     $a = \frac{\tilde{P}(x')Q(x|x')}{\tilde{P}(x)Q(x'|x)}$ $\triangleright$ Acceptance ratio
6:     $r = min(1, a)$ $\triangleright$ Acceptance condition
7:     $u \sim U(0, 1)$
8:     $x^{s+1} = \begin{cases} x' \text{ if } u < r \\ x^s \text{ if } u \geq r \end{cases}$
9: **end for**

0.1.2 HMC for MCMC

In a top level view Hamilton Monte Carlo for Monte Carlo Markov Chain (HMC MCMC) is a two step process. In step one we define a Hamiltonian function in terms of the probability distribution from which we wish to sample from. We introduce a position variable, q and momentum variable p, where p is an auxiliary variable that typically has a Gaussian distribution. All p's are assumed independent. In step two, the HMC alternates simple updates for the momentum variables with Metropolis updates. This enables us to propose a new state by computing a trajectory according to Hamiltonian dynamics, implemented with the leapfrog method.

Prerequisites

The Hamiltonian of a physical system is defined completely with respect to the position q and p momentum variables, which span the phase space of the system. The Hamiltonian is the Legendre transform of the Lagrangian and gives us the

total energy in the system. It is defined as follows:

$$H(\mathbf{q}, \mathbf{p}) = \sum_{i=1}^{d} \dot{q}_l p_l - L(\mathbf{q}, \dot{\mathbf{q}}(\mathbf{q}, \mathbf{p})) \tag{1}$$

where d is the system dimensions, and so the full state space with has $2d$ dimensions. Thus, for simplicity, if we set $d = 1$ we can derive the Hamiltonian equations as follows:

$$\frac{\partial H}{\partial p} = \dot{q} + p\frac{\partial \dot{q}}{\partial p} - \frac{\partial L}{\partial \dot{q}}\frac{\partial \dot{q}}{\partial p} = \dot{q} \tag{2}$$

and

$$\frac{\partial H}{\partial q} = p\frac{\partial \dot{q}}{\partial q} - \frac{\partial L}{\partial q} - \frac{\partial L}{\partial \dot{q}}\frac{\partial \dot{q}}{\partial q} = -\frac{\partial L}{\partial q} = -\dot{p} \tag{3}$$

and the process is the same for more than one dimension. We can write 1 more succinctly as:

$$H(\mathbf{q}, \mathbf{p}) = K(p) + U(q) \tag{4}$$

where $K(p)$ represents our kinetic energy and $U(q)$ is the potential energy.

Within the HMC MCMC framework the "positions", q, are the variables of interest and for each position variable we have to create a fictitious "momentum", p. For compactness let $z = (q, p)$. The potential energy $U(q)$ will be the minus of the log of the probabilty density for the distribution of the position variables we wish to sample, plus **any** constant that is convenient. The kinetic energy will represents the dynamics of our variables, for which a popular form of $K(p) = \frac{p^T M^{-1} p}{2}$, where M is symmetric, positive definite and typically diagonal. This form of $K(p)$ corresponds to a minus the log probability of the zero mean Gaussian distribution with covariance matrix M. For this choice we can write the Hamiltonian equations, for any dimension d, as follows:

$$\dot{q}_i = \frac{dq_i}{dt} = [M^{-1}p]_i \tag{5}$$

$$\dot{p}_i = \frac{dp_i}{dt} = -\frac{\partial U}{\partial q_i} \tag{6}$$

To view the Hamiltonian in terms of probabilities, we use the concept of the canonical distribution from Statistical mechanics to construct our pdf. Thus, the distribution that we wish to sample from can be related to the potential energy via the canonical distribution as:

$$P(z) = \frac{1}{Z}\exp\left(\frac{-E(z)}{T}\right) \tag{7}$$

As the Hamiltonian is just an energy function we may can insert 4 into our canonical distribution 7 which gives us the joint density:

$$P(q, p) = \frac{1}{Z}\exp(-U(q))\exp(-K(p) \tag{8}$$

where $T = 1$ is fixed. And so we can now very easily get to our target distribution $p(q)$, which is dependent on our choice of potential $U(q)$, as this expression factorizes in to two independent probability distributions. We characterise the posterior distribution for the model parameters using the potential energy function:

$$U(q) = -\log[\pi(q)L(q|D)] \tag{9}$$

where $\pi(q)$ is the prior distribution, and $L(q|D)$ is the likelihood, not the Lagrangian, of the given data D.

0.1.3 The Algorithm

The leapfrog method

The leapfrog method enables reduced error and allows us to dicretize Hamiltons equations, so that we can implement them numerically. We start with a state at $t = 0$ and then evaluate at a subsequent time $t + \epsilon, \ldots, t + n\epsilon$, where ϵ is the step in which we increase and n is the number of time steps.

$$p_i(t + \frac{\epsilon}{2}) = p_i(t) - \left(\frac{\epsilon}{2}\right) \frac{\partial U(q(t))}{\partial q_i} \tag{10}$$

$$q_i(t + \epsilon) = q_i(t) + \epsilon \frac{\partial K(p(t + \frac{\epsilon}{2}))}{dp_i} \tag{11}$$

$$p_i(t + \epsilon) = p_i(t + \frac{\epsilon}{2}) - \left(\frac{\epsilon}{2}\right) \frac{\partial U}{\partial q_i} \tag{12}$$

$$\tag{13}$$

1

For the leapfrog method the local error, error after one step, is of $\mathcal{O}(\epsilon^2)$ and a global error, error after simulating for some fixed time interval s, which requires $\frac{s}{\epsilon}$ is $\mathcal{O}(\epsilon^3)$

Some initial points of notice

Neals implementation of the HMC can only be used to sample from continuous distributions on $\mathbb{R}^d$ for which a density function can be evaluated.
We must be able to compute the partial derivative of the log of the density function. The derivaties must exists at the points at which they are evaluated [**Automatic differention**]
HMC samples from the canonical distribution for q and p. q has the distribution of interest as specified by the potential $U(q)$. The distribution of the p's can be chosen by us and are independent of the q's. The p components are specified to be independent, with component p_i having variance m_i. The kinetic energy $K(p) = \sum_{i=1}^{d} \frac{p_i^2}{2m_i}(q(t + \epsilon))$

---

[1]For the usual choice of kinetic energy, we have $\frac{\partial K(p + \frac{\epsilon}{2})}{dp_i} = \frac{p_i(t + \frac{\epsilon}{2})}{m_i}$

The steps

1. Step 1: Changes only the momentum

2. Step 2: May change both position and momentum

Both steps leave the canonical distribution of (q,p) invariant, hence the distribution remains invariant.

In **Step 1** we first draw the p_i randomly from their Gaussian distribution independently of the current values of the position values. In **Step 2** a Metropolis update is performed, using the Hamiltonian dynamics to propose a new state. Starting with the current state (q, p), Hamiltonian dynamics is simulated for L steps using the leapfrog method, with a stepsize of ϵ. L and ϵ are parameters of the mnodel that need to be tuned.

The momentum vairables at the end of this L-step trajectory are then negated, giving s proposed state $(q^*.p^*)$. This proposed state is accepted as the next state in the Markov Chain with probability:

$$\min[1, \exp(-H(q^*, p^*) + H(p, q)] = \min[1, \exp(-U(q^*) + U(q) - K(p^*) + K(p))] \tag{14}$$

If the proposed state is rejected, then the next state is the same as the current state and is counted again when calculating the expected value of some function. The negation of the momentum variables at the end of the trajectory makes the Metropolis proposal symmetrical, as needed for the acceptance probability above to be valid. This negation need not be done in practice, since $K(p) = K(-p)$, and the momentum will be replaced before it is used again, in the first step of the next iteration.

A function that implements a single iteration of the HMC algorithm is given in algorithm 2. There are three additional functions within this iteration: U, which returns the potential energy given a value for q, ∇U, which returns the vector of partial derivatives of U given q and ∇K, which returns the vector of partial derivatives of K given p. Other arguments are the stepsize, ϵ, for leapfrog steps, the number of leapfrog steps in the trajectory, L, and the current position, $q_{current}$, that the trajectory starts from. Momentum variables are sampled within this function, and discarded at the end, with only the next position being returned.

0.1.4 Tuning the HMC

We need to select suitable parameters ϵ and L, which together determine the length of the trajectory in fictitious time ϵL

0.2 Automatic differentiation AD

Automatic differentiation is quite different from both Numerical and Analytical differentiation. It is, in a very loose sense a hybrid of the two, that is optimal for computational performance. It works by systematically applying the chain

Algorithm 2 Simple Hamiltonian Monte Carlo MCMC

1: **procedure** HMC SINGLE ITERATION(ϵ, L, $q_{current}$)
2:     $q \leftarrow q_{current}$
3:     $p \sim \mathcal{N}(len(q), 0, 1)$
4:     $p_{current} \leftarrow p$
5:     $p \leftarrow p - \frac{\epsilon}{2}\nabla_q U(q)$ ▷ make half step for momentum
6:     **for** i in $1 : L$ **do**
7:         $q \leftarrow q + epsilon * \nabla_p K(p(t + \frac{\epsilon}{2}))$ ▷ make full step for the position
8:         **if** $i! = L$ **then**
9:             $p \leftarrow p - \epsilon\nabla_q U(q)$
10:         **end if**
11:     **end for**
12:     $p - \frac{\epsilon}{2}\nabla_q U(q)$ ▷ Half step for momentum
13:     $p \leftarrow -p$ ▷ Ensuring symmetry of Metropolis proposal
14:     $U_{current} \leftarrow U(q_{current})$
15:     $U_{proposed} \leftarrow U(q)$
16:     $K_{current} \leftarrow \frac{1}{2}p_{current} \cdot p_{current}$
17:     $K_{proposed} \leftarrow \frac{1}{2}p \cdot p$
18:     $u \sim Uniform(0, 1)$
19:     **if** $u < exp(U_{current} - U_{proposed} + K_{current} - K_{proposed})$ **then**
20:         return $q \leftarrow q$ ▷ Accept
21:     **else**
22:         return $q_{current} \leftarrow q_{current}$ ▷ Reject
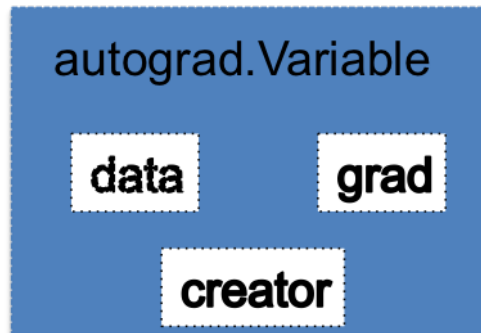23:   **end if**

Figure 1: Variable

rule of differential calculus at the elementary operator level, enabling us to generate numerical derivative evaluations through the accumulation of values. This "interleaving" idea forms the basis of AD and provides an account of its simplest form: apply symbolic differentiation at the elementary operation level and keep intermediate numerical results, in lockstep with the evaluation of the main function. This is AD in the forward accumulation mode

0.2.1 Automatic Differentiation in Pytorch

To use autodiff in pytorch we make use of the *autograd* package. it provides automatic differentiation for all operations on tensors. It is a **define-by-run framework**, which means brackprop is defined by how your code is run and that every single iteration **can be** different.

Variable

autograd.Variable is the central class of the package. Once you finish your call *.backward()* and have all the gradients computed automatically. You can access the raw tensor through the *.data* attribute, while the gradient w.r.t to that variable is accumulated in *.grad*. Another important class is *autograd.Function*, which is interconnected with *Variable*. Together they build an **acyclic** that encodes the complete history of the computation. Each variable has a *.creator* attribute that references a *Function* that has created the *Variable*.

Gradients

To calculate **gradients** from our *Variable* object we apply the method *¡newVariable¿.backward()* (where *newVariable* is created from the original variable) and then extract the attribute *¡Variable¿.grad* from the original variable. We are interested in the relation between the *newVariable* and the *Variable* as composition of functions, and the gradients of those functions evaluated at the a particular point.

0.2.2 Neural Networks in Torch

Using the *torch.nn* package, we can develop a NN. The *torch.nn* package is dependent upon the *autograd* package, to define models and differentiate them. The *nn.Module* contains layers and a method *.forward(input)* returns the *.output*.

Training Procedure

1. Define the NN that has some learnable parameters (or weights)

2. Iterate over a dataset of inputs

3. Process input through the network

4. Compute the loss (how far is the output from being corrected)

5. Propagate gradients back into the network's parameters

6. Update the weights of the network, using a update rule

A simple update rule might be of the form: $weight = weight - learningRate \times gradient$. When using PyTorch, you always have to define the *forward* function, but once that is done, PyTorch, using *autograd*, will automatically define the *backward* function. The learnable parameters of the model can be recalled by using *¡class_name_for_nn¿.parameters()*. See **blitztor.py** for more details.

Important note: *torch.nn* only supports mini-batches, the entire *torch.nn* package only supports inputs that are a mini-batch of samples and not a single sample.

To load data in to pytorch, we load data in via an nparray and convert this array into a *torch.*Tensor*. For loading audio, use scipy and librosa. For loading text either raw Python an Cython based loading, ir NLTK and SpaCy. For vision Pytorch has the *torchvision* package.

Loss function

The loss function takes the (output, target) pairs of inputs, and computers a value that estimates how far away the output is from the target, under some measure. The Pytorch library contains many different loss functions, see documentation. See **blitzor.py** for more details. To give all *Variables* there specified gradients, up to their node in the tree, we simply call *loss.backward()* and that accumulates all *.grad*'s to calculate the gradient for the different variables, w.r.t the loss. This also enables us to backprop the error through the network, although the existing gradients will need to be cleared. Else, you will be double adding. To zero we use the command *¡neural_net_object¿.zero_grad()*.

Backprop

To get any Variables derivative w.r.t the loss, we call, after doing *loss.backward()*,
¡network_name¿.¡Variable_name¿.{other_properties}.grad

Updating the Weights

Using the *torch.optim* package we can use an array of optimizers, including SGD.
See documentation.