Hamiltonian Monte Carlo

Bradley Gram-Hansen

July 20, 2017

1 Intro to HMC MCMC

Following Neal 2011 Chapter 5 Handbook of MCMCM

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: $x^0 \sim P_0(x)$ ▷ 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: $a = \frac{\tilde{P}(x')Q(x|x')}{\tilde{P}(x)Q(x'|x)}$ ▷ Acceptance ratio
6: $r = min(1, a)$ ▷ 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**

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.

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 differentiation**]

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.

2 Discrete 'Discontinuous' Hamiltonian Monte Carlo (DHMC)

Heavily Based on the paper by Nishimura et al. 2017.
Note: $q \to \theta$ The DHMC can handle discontinuous posterior distributions.
Key Ideas:

- Embedding of discrete parameters into a continous space, inducing parameters with piecewise constant densities.

Algorithm 2 MISTAKES - needs edit Simple Hamiltonian Monte Carlo MCMC

procedure HMC SINGLE ITERATION(ϵ, L, $q_{current}$)
  $q \leftarrow q_{current}$
  $p \sim \mathcal{N}(len(q), 0, 1)$
  $p_{current} \leftarrow p$
  $p \leftarrow p - \frac{\epsilon}{2}\nabla_q U(q)$ ▷ make half step for momentum
  for i in $1 : L$ **do**
    $q \leftarrow q + epsilon * \nabla_p K(p(t + \frac{\epsilon}{2}))$ ▷ make full step for the position
    if $i! = L$ **then**
      $p \leftarrow p - \epsilon\nabla_q U(q)$
    end if
  end for
  $p - \frac{\epsilon}{2}\nabla_q U(q)$ ▷ Half step for momentum
  $p \leftarrow -p$ ▷ Ensuring symmetry of Metropolis proposal
  $U_{current} \leftarrow U(q_{current})$
  $U_{proposed} \leftarrow U(q)$
  $K_{current} \leftarrow \frac{1}{2}p_{current} \cdot p_{current}$
  $K_{proposed} \leftarrow \frac{1}{2}p \cdot p$
  $u \sim Uniform(0, 1)$
  if $u < exp(U_{current} - U_{proposed} + K_{current} - K_{proposed})$ **then**
    return $q \leftarrow q$ ▷ Accept
  else
    return $q_{current} \leftarrow q_{current}$ ▷ Reject
  end if

- Simulation of Hamiltonian dynamics on a piecewise smooth density function

Given any target distribution, for each iteration, DHMC only requires evaluations of the density and the following:

- The full conditional densities of the discrete parameters up to a normalising constant.
- Which is done via evaluations of a directed acryclic graph framework, taking advantage of conditional independence structure (Lunn et al 2009)
- Either the gradient of the log density w.r.t. continuous parameters or their individual full conditional densities.

2.1 Embedding Discrete Parameters Into a Continuous Space

Let N denote a discrete parameter with prior distribution $\pi_n(\cdot)$, assume with out loss of generality that $N \in \mathbb{N}$. To embed N into a continuous space, we introduce a latent parameter $\tilde{N}$, whose relationship with N is defined as follows: $N = N \iff \tilde{N} \in (a_n, a_{n+1}]$ for a sequence $0 = a_1 \leq a_2 \leq \ldots$, where $a_n \in \mathbb{R}$.
To match the priordistribution $\pi(N)$, the corresponding, piecewise constant,

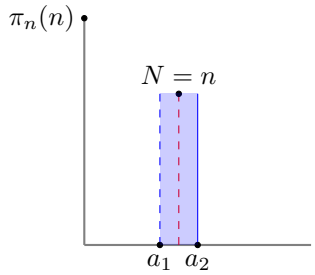

Figure 1: A view of our continuous embedding of a discrete parameter. The shaded region represents the piecewise constant prior density $\pi_{\tilde{N}}(\tilde{n})$ over $\tilde{N}$

prior over the latent parameter $\tilde{N}$ is given by:

$$\pi_{\tilde{N}}(\tilde{n}) = \sum_{n \geq 1} \frac{\pi_N(n)}{a_{n+1} - a_n} \mathbb{I}(a_n < \tilde{n} \leq a_{n+1}) \tag{15}$$

where the Jacobian-like factor $(a_{n+1} - a_n)^{-1}$ adjusts for embedding into non-uniform intervals. Rather than pick uniform embedding such as $a_n = n$ for all n, it is better to use a non-uniform one. For example $a_n = log(a_n)$.

Why does the HMC fail for discrete parameters

When the target distribution $\pi_\theta(\cdot)$ has discrete parameters, we have discontinuities. Thus, when we try and perform our integrator step incorporating Hamiltonian dynamics, which will depend on derivatives that require continuity, we of course are going to fail.

2.2 Resolution

To resolve this at points of discontinuity we need a correct limiting procedure to take into account the steps. A note on integrability, provided the discontinuities at the boundaries can be detected, Hamiltonians equations can be integrated to take into account the instantaneous change in $pi_\theta(\cdot)$. We do this by constructing a discontinuous set from $U(\Theta) = -log_{\pi(\theta)}(\theta)$, the potential energy, to form a piecewise smooth manifold.

Whilst it is true that a discontinuous function does not have a *well defined* derivative in the classical sense, the gradient of the potential can be defined through a distributional derivative, which allows us to interpret Hamiltonians equations 2 3 as a measure-valued differential equation. There is however a cravat, that being the solution to a measure-value differential equation is not unique. To circumvent this we use the **selection principle** (Ambrosio 2008) and construct a solution with desirable properties. In general, the process is as follows: Define a sequence of smooth approximations $U_\delta(\theta)$ of $U(\theta)$, for $\delta > 0$. We construct the smooth approximations through a convolution $U_\delta = U * \phi_\delta$, with $\phi_\delta(\theta) = \delta^{-d}\phi(\delta^{-1}\theta)$. Where ϕ is a compactly smooth function, i.e a function that has all the required properties to be a probability distribution $\phi \geq 0$ s.t. $\int \phi = 1$ and d is the dimension of θ.

2.3 Defining the trajectory

Let $\theta_\delta(t), p_\delta(t)$ be a solution to Hamiltons equations, with potential energy U_δ. Taking the limit as $\delta \to 0$ leads to the true trajectory $(\theta(t), p(t))$. Although the limit exists for most trajectories, it does exist for all (proof?). Despite this we define the dynamics of $U(\theta)$ in terms of this pointwise limit. Suppose the trajectory $(\theta(t), p(t))$ hits the discontinuity at an event time t_e. Let t_{e_-} and t_{e_+} denote the infinitesimal moment before and after that point, i.e our limiting values.
Since the discontinuity set of $U(\theta)$ was assumed to be piecewise smooth, at a discontinuity point θ we have:

$$\lim_{\delta \to 0} \frac{\nabla_\theta U_\delta(\theta)}{||\nabla_\theta U_\delta(\theta)||} = \nu(\theta) \tag{16}$$

where $\nu(\theta)$ denotes a unit vector orthonormal to the discontinuity, pointing in the direction of the higher potential. The relation 16 and $\frac{dp_\sigma}{dt} = -\nabla_\theta U_\delta$ imply

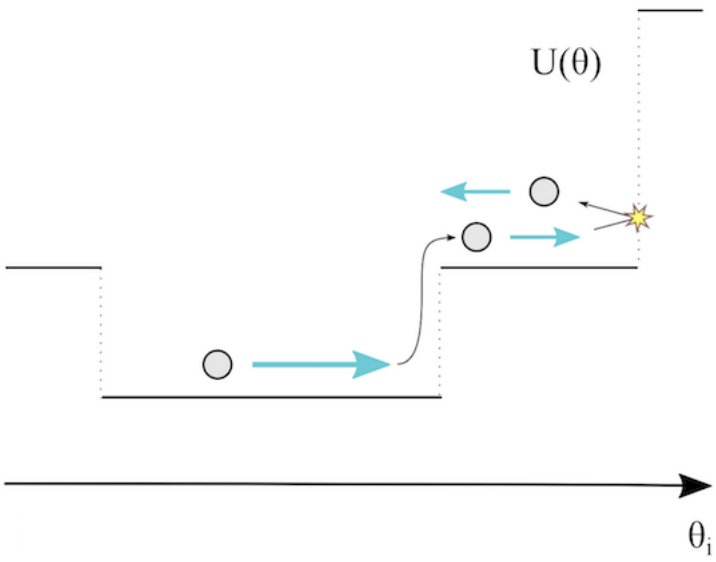

Figure 2: If there is not enough momentum energy, then the particle cannot progress to the step

that the only change in $p(t)$ upon encountering the discontinuity occurs in the direction of $\nu_e = \nu(\theta(t_e))$, that being:

$$p(t_{e_+}) = p(t_{e_-}) - \gamma\nu_e \tag{17}$$

for some $\gamma > 0$. There are two types possible types of change in p depending on the potential energy difference ∇U_e at the discontinuity, which are defined formally as:

$$\nabla U_e = \lim_{\epsilon \to 0^+} U(\theta(t_e)) + \epsilon p(t_{e_-}) - U(\theta(t_e)) \tag{18}$$

when the momentum does not provide enough kinetic energy to overcome the potential energy increase ∇U_e, the trajectory bounces back against the plane orthogonal to ν_e, see figure 2.3 . Else, the trajectory, particle, moves through the potential by transferring kinetic energy to potential energy. The magnitude of the instantaneous change γ is determined by looking at the conversation of energy:

$$K(p(t_{e_+})) - K(p(t_{e_-})) = U(\theta(t_{e_-})) - U(\theta(t_{e_+})) \tag{19}$$

2.4 The Need For A New Momentum

In the discrete setting Gaussian momentum defined in the Kinetic energy, $K(p) = \frac{p \cdot p}{m}$ does not work efficiently, as we now need to perform additional computations to detect the discontinuities. This leads [Nishimura et al. 2017] to use independent Laplace momentums, drawn from a Laplace distribution $\pi_p(p) \propto \Pi_i \exp(-\frac{|p_i|}{m_i})$. This allows us to construct a reversible integrator that can jump through multiple discontinuities in a small number of target density evaluations, while preserving the total energy. Under the Laplace momentum Hamiltonians equations become:

$$\dot{\theta} = m^{-1} \odot sign(p) \tag{20}$$

$$\dot{p} = -\nabla_\theta U(\theta) \tag{21}$$

and so we only depend on the sign of p and not the magnitudes. If we know that $p_i(t)$'s do not change their signs on the time interval $t \in [\tau, \tau + \epsilon]$, then we also know that the full step for the position will be:

$$\theta(t + \epsilon) = \theta(\tau) + \epsilon m^{-1} \odot sign(p(\tau)) \tag{22}$$

irrespective of the intermediate values $U(\theta(t))$ along the trajectory $(\theta(t), p(t))$ for $t \in [\tau, \tau+\epsilon]$. The ability of the integrator to jump through multiple discontinuities of $U(\theta)$ in a single target density evaluation depends critically on $\theta(\tau+\frac{\epsilon}{2})$. Where as the momentum step $p(\tau + \epsilon)$ is dependent on the intermediate values $U(\theta(t))$. This can actually be solved by splitting the differential operator in 21 into its coordinate-wise component.

2.5 Creating an Integrator for Laplace Momentum via Operator Splitting

To derive a scheme for the integrator under a independent Laplace momentum, we consider the equation for each coordinate (θ_i, p_i), while the other parameters (θ_{-i}, p_{-i}) are fixed, which would mean that the only non-zero derivatives are given by:

$$\dot{\theta_i} = m_i^{-1} sign(p_i), \;\; \dot{p_i} = -\nabla_{\theta_i} U(\theta) \tag{23}$$

In the case that the initial momentum is large enough that $m_i^{-1}|p_i(\tau)| > U(\theta^*(t)) - U(\theta(\tau))$ for all $t \in [\tau, \tau + \epsilon]$, we have:

$$\theta(t + \epsilon) = \theta^*(\tau + \epsilon) + \epsilon m^{-1} \odot sign(p(\tau)) \tag{24}$$

Otherwise, the momentum is p_i flips, and the trajectory $\theta(t)$ reverses its course at the event time t_e, figure 2.3 given by:

$$t_e = \inf\{t \in [\tau.\tau + \epsilon] : U(\theta^*(t)) - U(\theta(\tau)) > K(p(\tau))\} \tag{25}$$

The reversibility of the described integrator is guaranteed by randomly permuting the order of the coordinate updates.

Algorithm 3 Integrator step for purely discrete parameters, with Laplace momentum

procedure CORDINATE-WISE INTEGRATOR(θ, p, i, ϵ)
  $\theta_i^* \leftarrow \theta$
  $\theta_i^* \leftarrow \theta_i^* + \epsilon m_i^{-1} sign(p_i)$
  $\nabla U \leftarrow U(\theta^*) - U(\theta)$
  if $m_i^{-1}|p_i| > \nabla U$ **then**
    $\theta \leftarrow \theta^*$
    $p_i \leftarrow p_i - m_i \nabla U$
  else
    $p_i \leftarrow -p_i$
    reuturn θ, p

3 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 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

3.1 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 *¡new-Variable¿.backward()* (where *newVariable* is created from the original variable)

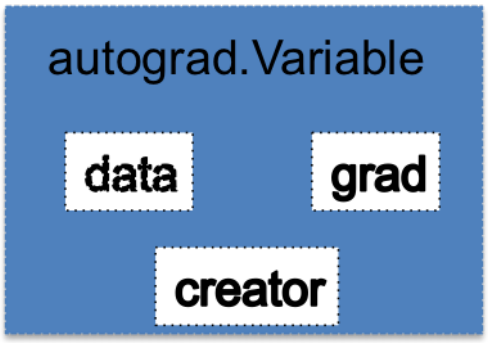

Figure 3: Variable

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.

3.2 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.