General Gradient of Neural Network

tomas.ukkonen@iki.fi, 2017

Backpropagation is a commonly known algorithm for computing the gradient of error function which arises when we know target values and the loss, cost or error function is one dimensional. Generalizing this to general gradient calculation when we seek to find the maximum or minimum value of a neural network (thought often ill-fated because of local optimas produced by an over-fitted neural network) is then important. This is needed, for example, when implementing certain reinforcement learning methods.

Consider a two-layer neural network

$$y(\boldsymbol{x}) = f(\boldsymbol{W}^{(2)}\, g(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + \boldsymbol{b}^{(2)})$$

The gradients of the final layer are (non-zero terms are at the j:th row):

$$\frac{\partial\, y(\boldsymbol{x})}{\partial\, w_{ji}^{(2)}} = \mathrm{diag}\Big(\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}\Big) \begin{pmatrix} 0 \\ g_i \\ 0 \end{pmatrix}$$

$$\frac{\partial\, y(\boldsymbol{x})}{\partial\, b_{j}^{(2)}} = \mathrm{diag}\Big(\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}\Big) \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

The derivation chain-rule can be used to calculate the second (and more deep layers' gradients):

$$\frac{\partial\, y(\boldsymbol{x})}{\partial\, w_{ji}^{(1)}} = \mathrm{diag}\Big(\frac{\partial f(\boldsymbol{x})}{\partial (\boldsymbol{W}^{(2)}\boldsymbol{g} + \boldsymbol{b}^{(2)})}\Big) \frac{\partial (\boldsymbol{W}^{(2)}\,\boldsymbol{g} + \boldsymbol{b}^{(2)})}{\partial \boldsymbol{g}} \frac{\partial \boldsymbol{g}(\boldsymbol{x})}{\partial (\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)})} \frac{\partial (\boldsymbol{W}^{(2)}\boldsymbol{x} + \boldsymbol{b}^{(2)})}{\partial\, w_{ji}^{2}}$$

$$\frac{\partial\, y(\boldsymbol{x})}{\partial\, w_{ji}^{(1)}} = \mathrm{diag}\Big(\frac{\partial f(\boldsymbol{x})}{\partial (\boldsymbol{W}^{(2)}\boldsymbol{g} + \boldsymbol{b}^{(2)})}\Big) \boldsymbol{W}^{(2)} \mathrm{diag}\Big(\frac{\partial \boldsymbol{g}(\boldsymbol{x})}{\partial (\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)})}\Big) \begin{pmatrix} 0 \\ x_i \\ 0 \end{pmatrix}$$

$$\frac{\partial\, y(\boldsymbol{x})}{\partial\, b_{j}^{(1)}} = \mathrm{diag}\Big(\frac{\partial f(\boldsymbol{x})}{\partial (\boldsymbol{W}^{(2)}\boldsymbol{g} + \boldsymbol{b}^{(2)})}\Big) \boldsymbol{W}^{(2)} \mathrm{diag}\Big(\frac{\partial \boldsymbol{g}(\boldsymbol{x})}{\partial (\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)})}\Big) \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

By analysing the chain rule we can derive generic backpropagation formula for the full gradient. Let $\boldsymbol{v}^{(k)}$ be a k:th layers local field, $\boldsymbol{v}^{(k)} = \boldsymbol{W}^{(k)} f(\boldsymbol{v}^{(k-1)}) + \boldsymbol{b}^{(k)}$. Then local gradient matrices $\boldsymbol{\delta}^{(k)}$ are

$$\boldsymbol{\delta}^{(L)} = \mathrm{diag}\Big(\frac{\partial \boldsymbol{f}(\boldsymbol{v}^{(L)})}{\partial \boldsymbol{v}^{(L)}}\Big)$$

$$\boldsymbol{\delta}^{(k-1)} = \boldsymbol{\delta}^{(k)}\, \boldsymbol{W}^{(k)}\, \mathrm{diag}\Big(\frac{\partial \boldsymbol{f}(\boldsymbol{v}^{(k-1)})}{\partial \boldsymbol{v}^{(k-1)}}\Big)$$

And network's parameter gradient matrices for each layer are (only j:th element of each row is non-zero):

$$\frac{\partial\, y(\boldsymbol{x})}{\partial\, w_{ji}^{(k)}} = \boldsymbol{\delta}^{(k)} \begin{pmatrix} 0 \\ f(v_i^{(k-1)}) \\ 0 \end{pmatrix}$$

$$\frac{\partial\, y(\boldsymbol{x})}{\partial\, b_{j}^{(k)}} = \boldsymbol{\delta}^{(k)} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

To test that gradient matrix is correctly computed it can be compared with normal squared error calculations (normal backpropagation).

$$\varepsilon(\boldsymbol{x}|\boldsymbol{w}) = \frac{1}{2}\|y_i - y(\boldsymbol{x}|\boldsymbol{w})\|^2$$

$$\frac{\partial \varepsilon(\boldsymbol{x}|\boldsymbol{w})}{\partial \boldsymbol{w}} = (y(\boldsymbol{x}|\boldsymbol{w}) - y_i)^T \frac{\partial y(\boldsymbol{x}|\boldsymbol{w})}{\partial\, \boldsymbol{w}}$$

Sometimes also needs gradient with respect to $\boldsymbol{x}$ and not weights parameters $\boldsymbol{w}$. This can be calculated using the chain rule again. For simplicity, let's consider two-layer case initially.

$$\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{W}^{(2)}\, \boldsymbol{h}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + \boldsymbol{b}^{(2)})$$

The gradient is:

$$\frac{\partial \boldsymbol{g}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{f}(\boldsymbol{v}^{(2)})}{\partial(\boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)})} \frac{\partial(\boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)})}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}(\boldsymbol{v}^{(1)})}{\partial(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)})} \frac{\partial(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)})}{\partial \boldsymbol{x}}$$

$$\frac{\partial \boldsymbol{g}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{f}(\boldsymbol{v}^{(2)})}{\partial \boldsymbol{v}^{(2)}} \boldsymbol{W}^{(2)} \frac{\partial \boldsymbol{h}(\boldsymbol{v}^{(1)})}{\partial \boldsymbol{v}^{(1)}} \boldsymbol{W}^{(1)}$$

This results into following formula (diag() entries are square matrices which diagonal is nonzero):

$$\nabla_{\boldsymbol{x}} \boldsymbol{g}(\boldsymbol{x}|\boldsymbol{w}) = \text{diag}(\nabla_{\boldsymbol{v}^{(L)}} \boldsymbol{f}(\boldsymbol{v}^{(L)}))\boldsymbol{W}^{(L)} ... \text{diag}(\nabla_{\boldsymbol{v}^{(2)}} \boldsymbol{f}(\boldsymbol{v}^{(2)}))\boldsymbol{W}^{(2)} \text{diag}(\nabla_{\boldsymbol{v}^{(1)}} \boldsymbol{f}(\boldsymbol{v}^{(1)}))\boldsymbol{W}^{(1)}$$

In continuous reinforcement learning, we need to maximize the given policy's $\boldsymbol{\mu}$ average **Q**-value $\boldsymbol{Q}(\boldsymbol{x},\ \boldsymbol{\mu}(\boldsymbol{x}))$ which gradient can be computed by using the chain-rule but there is additionally linear pre- and postprocessings $\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ in $\boldsymbol{\mu}$ and $\boldsymbol{Q}$ which makes the calculation of gradient more complicated.

$\nabla_{(\boldsymbol{W}_{\boldsymbol{\mu}}\boldsymbol{x} + \boldsymbol{b}_{\boldsymbol{\mu}})} \boldsymbol{W}'_Q \boldsymbol{Q}(\boldsymbol{W}_Q \boldsymbol{z} + \boldsymbol{b}_Q) + \boldsymbol{b}'_Q, \boldsymbol{z} = [\boldsymbol{x}, \boldsymbol{W}'_{\boldsymbol{\mu}} \boldsymbol{\mu}(\boldsymbol{W}_{\boldsymbol{\mu}}\boldsymbol{x} + \boldsymbol{b}_{\boldsymbol{\mu}}) + \boldsymbol{b}'_{\boldsymbol{\mu}}] =$

$\boldsymbol{W}'_Q \nabla \boldsymbol{Q}(\boldsymbol{W}_Q \boldsymbol{z} + \boldsymbol{b}_Q) \boldsymbol{W}_Q^{(\boldsymbol{\mu})} \boldsymbol{W}'_{\boldsymbol{\mu}} \nabla \boldsymbol{\mu}$

But in practice we don't have post-processing for $\boldsymbol{\mu}$ so the gradient becomes

$$\boldsymbol{W}'_Q \nabla \boldsymbol{Q}(\boldsymbol{W}_Q \boldsymbol{z} + \boldsymbol{b}_Q) \boldsymbol{W}_Q^{(\boldsymbol{\mu})} \nabla \boldsymbol{\mu}$$

Recurrent Neural Networks and Backpropagation (Similar to RTRL)

The basic learning algorithm for recurrent neural networks (RNN) is BPTT but I use modified RTRL instead. (RTRL - real time recurrent learning). This is done by unfolding neural net in time an computing the gradients. The recurrent neural network is

$$\boldsymbol{u}(n+1) = \begin{pmatrix} \boldsymbol{y}(n+1) \\ \boldsymbol{r}(n+1) \end{pmatrix} = \boldsymbol{f}(\boldsymbol{x}(n), \boldsymbol{r}(n))$$

The error function to minimize is:

$$E(N) = \frac{1}{2}\sum_{n=1}^{N} \|\boldsymbol{d}(n+1) - \boldsymbol{\Gamma}_{\boldsymbol{y}} \boldsymbol{f}(\boldsymbol{x}(n), \boldsymbol{\Gamma}_{\boldsymbol{r}} \boldsymbol{u}(n-1)|\boldsymbol{w})\|^2$$

In which $\boldsymbol{\Gamma}$ matrices are used to select $\boldsymbol{y}(n)$ and $\boldsymbol{r}(n)$ vectors from generic output vector and the initial input to feedforward neural network is zero $\boldsymbol{u}(0) = \boldsymbol{0}$.

It is possible to calculate gradient of $\boldsymbol{f}$ using the chain rule

$$\frac{\partial E(N)}{\partial \boldsymbol{w}} = \sum_{n=0}^{N} (\boldsymbol{\Gamma}_{\boldsymbol{y}} \boldsymbol{f}(\boldsymbol{x}(n), \boldsymbol{\Gamma}_{\boldsymbol{r}} \boldsymbol{u}(n-1)|\boldsymbol{w}) - \boldsymbol{d}(n+1))^T \boldsymbol{\Gamma}_{\boldsymbol{y}} \nabla_{\boldsymbol{w}} \boldsymbol{f}(\boldsymbol{x}(n), \boldsymbol{\Gamma}_{\boldsymbol{r}} \boldsymbol{u}(n-1)|\boldsymbol{w})$$

To calculate the gradient $\nabla_{\boldsymbol{w}} \boldsymbol{f}$ one must remember that $\boldsymbol{u}(n)$ now also depends on $\boldsymbol{w}$ resulting into eq:

$$\nabla_{\boldsymbol{w}} \boldsymbol{f}(\boldsymbol{x}(n), \boldsymbol{\Gamma}_{\boldsymbol{r}} \boldsymbol{u}(n-1)) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{w}} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{r}} \boldsymbol{\Gamma}_{\boldsymbol{r}} \frac{\partial \boldsymbol{u}(n-1)}{\partial \boldsymbol{w}}$$

To further compute gradients we get a generic update rule

$$\frac{\partial \boldsymbol{u}(n)}{\partial \boldsymbol{w}} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{w}} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{r}} \boldsymbol{\Gamma}_{\boldsymbol{r}} \frac{\partial \boldsymbol{u}(n-1)}{\partial \boldsymbol{w}}$$

The computation of gradients can be therefore bootstrapped by setting $\frac{\partial \boldsymbol{u}(0)}{\partial \boldsymbol{w}} = \frac{\partial \boldsymbol{f}(\boldsymbol{0}, \boldsymbol{0})}{\partial \boldsymbol{w}}$ and iteratively updating $\boldsymbol{u}$ gradient while computing the current error for the timestep.

RNN-RBM

RNN-RBM was described on the web to create learn creating "music" by using BPTT but I use extended RTRL approach instead.

(`http://danshiebler.com/2016-08-17-musical-tensorflow-part-two-the-rnn-rbm/`)

In RNN-RBM we have a standard RBM model but RBM's biases $\boldsymbol{a}(n+1)$ and $\boldsymbol{b}(n+1)$ are generated by a recurrent neural network $\begin{pmatrix} \boldsymbol{y}(n+1) \\ \boldsymbol{r}(n+1) \end{pmatrix} = \boldsymbol{f}(\boldsymbol{x}(n), \boldsymbol{r}(n)|\boldsymbol{w})$, $\boldsymbol{y}(n+1) = \begin{pmatrix} \boldsymbol{a}(n+1) \\ \boldsymbol{b}(n+1) \end{pmatrix}$ and visible units (MIDI notes) are fed to be inputs of recurrent neural network $\boldsymbol{x}(n) = \boldsymbol{v}(n)$.

One can then compute RBM's log-likelihood gradient with respect to recurrent neural networks weights $\boldsymbol{w}$ maximizing probability of "semi" independent MIDI notes observations

$$-\log[p(\boldsymbol{v}(1), \boldsymbol{v}(2)...\boldsymbol{v}(N))] \approx \sum_n -\log(p(\boldsymbol{v}_n))$$

We want to calculate gradient with respect to $\boldsymbol{w}$ where only elements $\boldsymbol{a}$ and $\boldsymbol{b}$ depend on $\boldsymbol{w}$.

$$\frac{\partial -\log(p(\boldsymbol{v}_i))}{\partial \boldsymbol{w}} = \frac{\partial -\log(p(\boldsymbol{v}_i))}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{w}} + \frac{\partial -\log(p(\boldsymbol{v}_i))}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{w}}$$

This can be rewritten using free-energy $p(\boldsymbol{v}) = \frac{1}{Z} e^{-F(\boldsymbol{v})} = \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})}$ and the gradient formula for $\frac{\partial -\log(p(\boldsymbol{v}))}{\partial \theta} = \frac{\partial F(\boldsymbol{v})}{\partial \theta} - E_{\boldsymbol{v}}\Big[\frac{\partial F(\boldsymbol{v})}{\partial \theta}\Big]$:

$$\frac{\partial -\log(p(\boldsymbol{v}_i))}{\partial \boldsymbol{w}} = \Big(\frac{\partial F(\boldsymbol{v})}{\partial \boldsymbol{a}} - E_{\boldsymbol{v}}\Big[\frac{\partial F(\boldsymbol{v})}{\partial \boldsymbol{a}}\Big]\Big)\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{w}} + \Big(\frac{\partial F(\boldsymbol{v})}{\partial \boldsymbol{b}} - E_{\boldsymbol{v}}\Big[\frac{\partial F(\boldsymbol{v})}{\partial \boldsymbol{b}}\Big]\Big)\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{w}}$$

We assume GB-RBM model (`see RBM_notes.tm`) with the following energy function

$$E_{\text{GB}}(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{2}(\boldsymbol{v} - \boldsymbol{a})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{v} - \boldsymbol{a}) - (\boldsymbol{\Sigma}^{-0.5}\boldsymbol{v})^T \boldsymbol{W}\boldsymbol{h} - \boldsymbol{b}^T\boldsymbol{h}$$

And we extend RNN to also output/predict variance $\boldsymbol{z}(n) = \log(\text{diag}(\boldsymbol{\Sigma}(n)))$ [here we use $\boldsymbol{z}(n)$ for two different things, one for variance parameter of GB-RBM and other for recurrent neural networks output]. Our gradients therefore are (`see RBM_notes.tm`):

$$\frac{\partial \boldsymbol{u}(n)}{\partial \boldsymbol{w}} = \frac{\partial \boldsymbol{f}(\boldsymbol{v}(n))}{\partial \boldsymbol{w}} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{r}} \boldsymbol{\Gamma}_{\boldsymbol{r}} \frac{\partial \boldsymbol{u}(n-1)}{\partial \boldsymbol{w}},\ \boldsymbol{u}(n) = \begin{pmatrix} \boldsymbol{a}(n) \\ \boldsymbol{b}(n) \\ \boldsymbol{z}(n) \\ \boldsymbol{r}(n) \end{pmatrix}$$

$$\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{w}} = \boldsymbol{\Gamma}_{\boldsymbol{a}} \frac{\partial \boldsymbol{u}(n)}{\partial \boldsymbol{w}}$$

$$\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{w}} = \boldsymbol{\Gamma}_{\boldsymbol{b}} \frac{\partial \boldsymbol{u}(n)}{\partial \boldsymbol{w}}$$

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{w}} = \boldsymbol{\Gamma}_{\boldsymbol{z}} \frac{\partial \boldsymbol{u}(n)}{\partial \boldsymbol{w}}$$

$$\frac{\partial F}{\partial \boldsymbol{a}} = -\boldsymbol{\Sigma}^{-1}(\boldsymbol{v} - \boldsymbol{a})$$

$$\frac{\partial F}{\partial \boldsymbol{b}} = -\text{sigmoid}(\boldsymbol{W}^T \boldsymbol{\Sigma}^{-1/2} \boldsymbol{v} + \boldsymbol{b})$$

$$\frac{\partial F}{\partial \boldsymbol{z}_i} = -\frac{1}{2}(v_i - a_i)^2 e^{-z_i} + \frac{1}{2} e^{-z_i/2} v_i \sum_j w_{ij}\, \text{sigmoid}(\boldsymbol{W}^T \boldsymbol{\Sigma}^{-1/2} \boldsymbol{v} + \boldsymbol{b})_j$$

$$\frac{\partial F}{\partial \boldsymbol{W}} = -\text{sigmoid}(\boldsymbol{W}^T \boldsymbol{\Sigma}^{-1/2} \boldsymbol{v} + \boldsymbol{b})\, (\boldsymbol{\Sigma}^{-1/2}\boldsymbol{v})^T$$

When using these gradients it is important to remember that, in general, one must update variance terms $\boldsymbol{z}$ independently from other parameters or the GB-RBM doesn't converge. Initially, however, I will make an attempt to learn both variance $\boldsymbol{z}$, $\boldsymbol{a}$ and $\boldsymbol{b}$ because they now all depend on common weight vector parameter $\boldsymbol{w}$. Pseudocode for RNN-RBM optimization:

1. randomly initialize $\boldsymbol{W}$ and $\boldsymbol{w}$ using small values.

2. Calculate parameters $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{z}$ using current RNN (initially with zero recurrent input parameters including previous step's visible MIDI notes $\boldsymbol{v}(n-1)$)

3. Use RBM and CD-k to calculate $(\boldsymbol{v}, \boldsymbol{h})$ parameters for input sample(s) and calculate contrastive divergence samples in order to calculate gradient of free energy $\nabla \boldsymbol{F}$.

4. Calculate the gradient of the recurrent neural network $\nabla_{\boldsymbol{w}} \boldsymbol{u}(n)$ and use $\nabla \boldsymbol{F}$ to calculate gradient of the probability $p(\boldsymbol{v})$ with respect to $\boldsymbol{W}$ and $\boldsymbol{w}$.

5. Repeat steps 2-4 for each song i (time series) of visible notes $\{\boldsymbol{v}_i(n)\}$ and use the sum of all songs gradients to move parameters $\boldsymbol{W}$ and $\boldsymbol{w}$ towards (hopefully) higher probability of data. (IMPLEMENTATION NOTE: concatenate all songs as a single time-serie and try to learn it).

BB-RBM

Instead of GB-RBM which variance learning is complicated. I initially (and also) implement and test BB-RBM implementation as the RBM part of the RNN-RBM. In this case there is no variance terms $\boldsymbol{z}_i$ to worry about.

$\frac{\partial F}{\partial \boldsymbol{a}} = -\boldsymbol{v}$

$\frac{\partial F}{\partial \boldsymbol{b}} = -\text{sigmoid}(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{b})$

$\frac{\partial F}{\partial \boldsymbol{W}} = -\text{sigmoid}(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{b})\boldsymbol{v}^T$

NOTE: initial use of RNN-RBM outlined here seem to diverge quickly to chaos (many random notes played at once) when applying RNN-RBM to classical MIDI notes data (note range: C-4 .. B-6). It seems problem should be regularized somehow to limit number of on notes played at once.

In practice it seems to be difficult to regularize RNN-RBM because of the special form of the error function (log probability). I suggest the use of "negative gradient". In addition to training samples which probability should be maximized, artificial songs are created where each note has random probability $p > 0.50$ of being in on position and gradient of these additional training samples is calculated normally but the calculated gradient is substracted from the positive gradient so that probability of those "random songs" is greatly reduced.