Uncertainty analysis for neural network

What I need is uncertainty analysis of neural network. Because bayesian method doesn't seem to work. I instead do direct uncertainty analysis. Let's define probability of weights through error rate

$$p(\boldsymbol{\omega}|\text{data}) = N(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\omega})|0, \boldsymbol{x}_i, \boldsymbol{y}_i, \sigma^2)$$

And choose σ^2 from precomputed optimal solution $\boldsymbol{\omega}_0$ which is also a starting point. The related energy functions are

$$U(\boldsymbol{\omega}) = \frac{1}{2}\sum_i \|\boldsymbol{y}_i - f(\boldsymbol{x}_i, \boldsymbol{w})\|^2/\sigma^2$$

$$\nabla_{\boldsymbol{\omega}} U(\boldsymbol{\omega}) = \sum_i \nabla_{\boldsymbol{\omega}} \frac{1}{2}\|\boldsymbol{y}_i - f(\boldsymbol{x}_i, \boldsymbol{w})\|^2/\sigma^2$$

This method works for any number of dimensions but do not have prior information and overfits to the data.

Implementation is in UHMC.cpp.

Hamiltonian Monte Carlo Bayesian Neural Network implementation notes (**"Bayesian Gradient Descent"**)

This does only seem to work in 1-dimensional output for some reason. Implementation is in HMC.cpp.

Tomas Ukkonen, 2015

Bayesian neural network extends basic feedforward network. This paper also describes general idea of *"bayesian gradient descent"* which arises when solving generic error minimization which maximizes likelihood of parameters given data instead of likelihood of data given parameters. This will then lead to qualitatively different and better solutions.

NOTE: Feedforward networks are hopelessly outdated so after you get this one working somehow just move as quickly as possible to Stacked RBMs (Bayesian GB-RBM 80% done).

The optimization method used is Hamiltionian Monte Carlo (HMC) which we assumes we have a distribution $\frac{1}{Z}p(\boldsymbol{w}) = e^{-U(\boldsymbol{w})}$ from which to sample from. Now according to bayesian inference rule we have

$$p(\boldsymbol{\omega}|\text{data}) = p(\text{data}|\boldsymbol{\omega})\, p(\boldsymbol{\omega})$$

And we want to sample from $p(\boldsymbol{\omega}|\text{data}) = \frac{1}{Z(\text{data})}\, e^{-U(\boldsymbol{\omega})}$ using HMC. We can rewrite data likelihood to be $p(\boldsymbol{y}, \boldsymbol{x}|\boldsymbol{\omega}) = p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\omega})\, p(\boldsymbol{x}|\boldsymbol{\omega})$. We now define the first term to be squared error function with normally distributed noise $\boldsymbol{y} \sim N(f(\boldsymbol{x}, \boldsymbol{w}), \mathbf{I}\,\sigma^2)$ with fixed variance σ^2. For $p(\boldsymbol{x}|\boldsymbol{\omega})$ we must use crude approximation and simply regularize/simplify the problem and model our data to have same distribution as the data by choosing $\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})$ and select similarly prior for weights to be gaussian ball with unit variance $\frac{1}{2}\|\boldsymbol{w}\|^2$. The known covariance matrix $\boldsymbol{\Sigma}$ can be obtained before sampling $\boldsymbol{\omega}$ by using some pre-optimization method and calculating the related covariance or by using gibbs sampling to sample from $p(\sigma^2|\boldsymbol{\omega}, \text{data})$ before sampling from $p(\boldsymbol{\omega}|\boldsymbol{\Sigma}, \text{data})$. When taking logarithms from the both sides of the the inference rule this will lead to equation:

$$U(\boldsymbol{w}) = \frac{1}{2}\sum_i \|\boldsymbol{y}_i - f(\boldsymbol{x}_i, \boldsymbol{w})\|^2/\sigma^2 + \log\left(Z(\boldsymbol{\omega})\right) + \frac{1}{2}\|\boldsymbol{w}\|^2 - \log\left(p_{\text{data}}(\boldsymbol{x})\right) + C(\text{data})$$

The most troublesome term in the equation is the $Z(\boldsymbol{\omega})$ partition function that integrates *over data*, is function of $\boldsymbol{\omega}$ and cannot be typically ignored (but is ignored in some contexts/books):

$$Z(\boldsymbol{\omega}) = \int\, e^{-\frac{1}{2}\sum_i \|\boldsymbol{y}_i - f(\boldsymbol{x}_i, \boldsymbol{\omega})\|^2/\sigma^2} p_{\text{data}}(\boldsymbol{x}_1 ... \boldsymbol{x}_N)\, d\,\boldsymbol{x}_1 ... d\boldsymbol{x}_N\, d\boldsymbol{y}_1 ... d\boldsymbol{y}_N$$

Luckily we don't have to calculate $Z(\boldsymbol{\omega})$ directly because in HMC we don't need value of $U(\boldsymbol{\omega})$ but its gradient and difference (or in general MCMC we just need the difference):

$$\nabla_{\boldsymbol{\omega}} U(\boldsymbol{\omega}) = \nabla_{\boldsymbol{\omega}} e(\boldsymbol{\omega}) + \frac{1}{Z(\boldsymbol{\omega})} \nabla_{\boldsymbol{w}} Z(\boldsymbol{\omega}) + \boldsymbol{w}$$

$$U(\boldsymbol{w}_{n+1}) - U(\boldsymbol{\omega}_n) = e(\boldsymbol{\omega}_{n+1}) - e(\boldsymbol{\omega}_n) + \log\left(\frac{Z(\boldsymbol{\omega}_{n+1})}{Z(\boldsymbol{\omega}_n)}\right) + \frac{1}{2}(\|\boldsymbol{\omega}_{n+1}\|^2 - \|\boldsymbol{\omega}_n\|^2)$$

$$e(\boldsymbol{\omega}) = \frac{1}{2}\sum_i \|\boldsymbol{y}_i - f(\boldsymbol{x}_i, \boldsymbol{w})\|^2 / \sigma^2$$

It is rather straightforward to simplify the gradient of the partition function further:

$$\frac{1}{Z(\boldsymbol{\omega})} \nabla_{\boldsymbol{w}} Z(\boldsymbol{\omega}) = -\int \nabla_{\boldsymbol{\omega}} e(\boldsymbol{\omega})\, p(\text{data}|\boldsymbol{w})\, d\,\text{data}$$

This last equation is interesting one because it defines sum of expected errors when ($\boldsymbol{x}$, $\boldsymbol{y}$) is distributed according to our neural network model $\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{w})$. If we make assumption that (in our model) each pair ($\boldsymbol{x}$, $\boldsymbol{y}$) is independently distributed and have the same distribution we can write

$$\int \nabla_{\boldsymbol{\omega}} e(\boldsymbol{\omega})\, p(\text{data}|\boldsymbol{w})\, d\,\text{data} = N \int \frac{1}{2} \nabla_{\boldsymbol{\omega}} \|\boldsymbol{y} - f(\boldsymbol{x}, \boldsymbol{w})\|^2 / \sigma^2\, p_{\text{model}}(\boldsymbol{x}, \boldsymbol{y}|\boldsymbol{w}) d\,\boldsymbol{x}\, d\boldsymbol{y}$$

By inspecting the calculation methods we can see that gradient descent matches distribution of model $p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\omega})$ as close as possible to distribution of data $p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})$.

Learning variance parameter

In practice, variance term must be learnt in order to match model into data as well as possible. We can extend our model by writing

$$p(\boldsymbol{\omega}, \sigma^2|\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{\omega}|\sigma^2, \boldsymbol{x}, \boldsymbol{y})\, p(\sigma^2|\boldsymbol{x}, \boldsymbol{y})$$

Weight parameters $\boldsymbol{\omega}$ can be sampled from the first distribution and we can rewrite the equation another way around

$$p(\sigma^2, \boldsymbol{\omega}|\boldsymbol{x}, \boldsymbol{y}) = p(\sigma^2|\boldsymbol{\omega}, \boldsymbol{x}, \boldsymbol{y})\, p(\boldsymbol{\omega}|\boldsymbol{x}, \boldsymbol{y})$$

And use again the first probability term to sample/estimate σ^2. We can then alternate these sampling steps to generate samples from joint probability. In practice we need to estimate

$p(\sigma^2|\boldsymbol{\omega}, \boldsymbol{x}, \boldsymbol{y})$ which can be done simply by calculating average squared error without sampling (kind of maximum likelihood estimate).

Analysis of $U(\boldsymbol{\omega})$ and effect of $Z(\boldsymbol{\omega})$

At optimum points we have $\nabla_{\boldsymbol{\omega}} U(\boldsymbol{\omega}) = 0$. Therefore we have an equation:

$$\int \nabla_{\boldsymbol{\omega}}\, e(\boldsymbol{\omega})\, [p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x}) - p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\omega})]\, p_{\text{data}}(\boldsymbol{x})\, d\boldsymbol{x}\, d\boldsymbol{y} \approx 0$$

So in optimum points distributions of data and model are matched, or if that is not possible, $U(\boldsymbol{\omega})$ increases or decreases endlessly when generating sampling points using sampler. In practice $U(\boldsymbol{\omega})$ tends to increase to lower probability meaning that $p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x}) < p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\omega})$. Additionally, we can see from the equations that $Z(\boldsymbol{\omega})$ terms cause model to maximize generic model error while data terms minimize error of fitting function to data. Integrating gradient equation we can get formula for $U(\boldsymbol{\omega})$ without Z function.

$$U(\boldsymbol{\omega}) = N\, (E_{\text{data}}[e(\boldsymbol{\omega})] - E_{\text{model}}[e(\boldsymbol{\omega})]) + \frac{1}{2}\|\boldsymbol{\omega}\|^2$$

This equation seem to have a problem that in practice the $E_{\text{model}}[e(\boldsymbol{\omega})]$ tends to dominate in our approximated models meaning that $N E_{\text{data}}[e(\boldsymbol{\omega})] + \frac{1}{2}\|\boldsymbol{\omega}\|^2$ can increase too much. In a simplified analysis there is too much variance in our model but in bayesian probabilistic analysis data and model has equal amount of variance (it might be possible to get beyond bayesian probability analysis somehow by altering $E_{\text{model}}[e(\boldsymbol{\omega})]$ this might include negative probabilities, complex number valued probabilities or other exotic calculation methods - altering E_{model} / Z means that we don't anymore force our probability mass to have fixed value).

We diverge when data has smaller variance (error) than modelling variance. Therefore we must match modelling variance to data variance but our modelling errors tends to increase variance/error of model. When our model too simple variance can grow too much.

Assuming we have initial solution $\boldsymbol{\omega}_0$ which is known to be a good function/model parameters for the data, we can write altered form (our model now integrates over $\boldsymbol{\omega}$ to give a single good model $\boldsymbol{\omega}_0$):

$$U^*(\boldsymbol{\omega}) = N\,(E_{\text{data}}[e(\boldsymbol{\omega})] - E_{\text{model},\boldsymbol{\omega}_0}[e(\boldsymbol{\omega})])$$

And the negative term becomes constant because we force our model to has $\boldsymbol{\omega}_0$ parameters while data term is really sum of squared error values (only approximating E_{data}) and not given parameter $\boldsymbol{\omega}_0$. This mean we will ignore $Z(\boldsymbol{\omega})$ term which maximizes function complexity (variance/error). After this we will only look for solutions $\boldsymbol{\omega}$ which allows small probabilistic variations in errors. This ignorance of $Z(\boldsymbol{\omega})$ will also allow for probability mass of the function to be infinite?.

Solving inverse values of function (not needed anymore)

If input space $\boldsymbol{x}$ is small we can do parallel search (fast) by picking $\boldsymbol{x}$ points randomly (or more preferrably from training data) and then use gradient descent to minimize error function $e(\boldsymbol{x}) = \frac{1}{2}(\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x}))^2$ and keep the solutions which are "close enough zero" (or at least N smallest error cases). This requires calculating $\frac{d\boldsymbol{f}}{d\,\boldsymbol{x}}$ which we can compute using chain rule. For example, assuming two layer neural network layers we have:

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}_2(\boldsymbol{A}_2\,\boldsymbol{f}_1(\boldsymbol{A}_1\,\boldsymbol{x} + \boldsymbol{a}_!) + \boldsymbol{a}_2)$$

$$\frac{d\boldsymbol{f}}{d\boldsymbol{x}} = \boldsymbol{f}_2'(\boldsymbol{A}_2\,\boldsymbol{f}_1(\boldsymbol{A}_1\,\boldsymbol{x} + \boldsymbol{a}_!) + \boldsymbol{a}_2)\,\boldsymbol{A}_2\,\boldsymbol{f}_1'(\boldsymbol{A}_1\boldsymbol{x} + \boldsymbol{a})\,\boldsymbol{A}_1$$

And it is straightforward to extend this to multiple layers. However, the theoretical problem is that there are probably very large or infinite many possible input values $\boldsymbol{x}$ for many cases of $\boldsymbol{y}$ leading to very difficult to estimate $p(\boldsymbol{x}|\boldsymbol{y})$. We can limit the number of solutions if we restrict ourselves to the convex space (+ little extra around corners to do extrapolation..) and set up by training points and calculate our inverse function and probability function only in that space.

In practice, the inverse solutions can be extremely large (outside the convex space setup by data). If we assume our data has zero mean and unit variance, then we can limit the solutions space by regularizing/using prior for $\boldsymbol{x}$ by forcing it to be taken from gaussian ball and define error function to be:

$$E(\boldsymbol{x}) = \tfrac{1}{2}(\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x}))^2 + \tfrac{1}{2}\|\boldsymbol{x}\|^2/\alpha^2$$

$$\nabla E(\boldsymbol{x}) = (\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x}))^T \nabla\,\boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{x}/\alpha^2$$

Where alpha is accuracy/inverse variance parameter which we can set $\alpha = 3$.

After solving inverse values we can then assign probability to each $\boldsymbol{x}$ according to probability density function, normalize to unity and select one according to probability distribution.

Solving difference equation

After we have computed gradient of the network we must solve ratio $\log\left(\frac{Z(\boldsymbol{\omega}_{n+1})}{Z(\boldsymbol{\omega}_n)}\right)$ in order to make accept/reject decisions. We can estimate partition function ratios by using (reference):

$$\frac{Z(\boldsymbol{\omega}_{n+1})}{Z(\boldsymbol{\omega}_n)} = \int\ \frac{1}{Z(\boldsymbol{\omega}_n)}\,p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_{n+1})\,d\boldsymbol{x}\,d\boldsymbol{y} = \int\ \frac{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_{n+1})}{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_n)}\,p(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_n)\,d\boldsymbol{x}\,d\boldsymbol{y}$$

So we must calculate estimate $E_{\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_n}\left[\frac{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_{n+1})}{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_n)}\right]$ by calculating ratio of biassed probability distributions of data. In pratice we have multiple data points in our data likelihood. This leads to equation:

$$E_{\{\boldsymbol{x}_i,\boldsymbol{y}_i\}|\boldsymbol{\omega}_n}\left[\prod_i \frac{p^*(\boldsymbol{x}_i,\boldsymbol{y}_i|\boldsymbol{\omega}_{n+1})}{p^*(\boldsymbol{x}_i,\boldsymbol{y}_i|\boldsymbol{\omega}_n)}\right]$$

Now our terms are identically distributed and independent from each other $E[A\,B] = E[A\,]\,E[B]$, this means

$$E_{\{\boldsymbol{x}_i,\boldsymbol{y}_i\}|\boldsymbol{\omega}_n}\Big[\prod_i \frac{p^*(\boldsymbol{x}_i,\boldsymbol{y}_i|\boldsymbol{\omega}_{n+1})}{p^*(\boldsymbol{x}_i,\boldsymbol{y}_i|\boldsymbol{\omega}_n)}\Big] = \prod_i E_{\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_n}\Big[\frac{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_{n+1})}{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_n)}\Big] = E_{\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_n}\Big[\frac{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_{n+1})}{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_n)}\Big]^N$$

$$\frac{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_{n+1})}{p^*(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{\omega}_n)} = e^{\frac{1}{2}\|\boldsymbol{y}-f(\boldsymbol{x},\boldsymbol{w}_n)\|^2/\sigma^2 - \frac{1}{2}\|\boldsymbol{y}-f(\boldsymbol{x},\boldsymbol{w}_{n+1})\|^2/\sigma^2}$$

And we sample from $p_{\text{model}}(\boldsymbol{y},\boldsymbol{x}|\boldsymbol{w}_n) = p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{w}_n)\; p_{\text{data}}(\boldsymbol{x})$ in feedforward neural network because we are only interested in conditional distribution $p(\boldsymbol{y}|\boldsymbol{x})$ which is in our model is normal distribution.

Additional implementation notes

In practice, we need to estimate negative phase gradient and zratio by sampling from distribution $p_{\text{model}}(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{w})$, which we can only crudely approximate. We are estimating mean value of z-ratio and mean value of gradients (vector) and we much estimate error levels in both of them in order to gather enough samples.

TODO

Divergence of HMC sampler

In practice HMC sampler diverge to larger and larger error levels after successfully visiting small levels of error. The reason for this is currently unclear but only way for sampler to keep accepting the new samples is that Z-ratio becomes very small to always dominate the results which then also means that gradient and is based on Z-value and other terms are ignored.

More detailed analysis seems to imply that the problem is variance (or covariance) term in bayesian neural network. If variance is set to unity, it sets the accuracy (level of detail) of our predicted outcome. Too high variance when compared to data variance means that model "underfits" to the data. Moreover, gradient descent rule now keeps increasing $U(\boldsymbol{w})$ leading to flat posterior distribution $p(\boldsymbol{w})$ [because any parameters can fit to data with too much variance in model] and high model error levels. This is because tails of the model distribution (extreme values) are more likely in $p_{\text{model}}(\boldsymbol{x},\boldsymbol{y}|\boldsymbol{w})$ which means that following the negative gradient causes $U(\boldsymbol{w})$ to increase

$$\nabla_{\boldsymbol{\omega}} U(\boldsymbol{\omega}) = \int\; \nabla_{\boldsymbol{\omega}} \frac{1}{2}\|\boldsymbol{y}-\boldsymbol{f}(\boldsymbol{y}|\boldsymbol{\omega})\|^2 [p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})\, p_{\text{data}}(\boldsymbol{x}) - p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{w})\, p_{\text{data}}(\boldsymbol{x})] + \boldsymbol{w}$$

Additionally, we can also see that gradient descent rule will converge when model distribution has converged to data distribution. And because in practice we are only interested in $p(\boldsymbol{y}|\boldsymbol{x})$ distribution we don't have to calculate inverse of $\boldsymbol{f}(\boldsymbol{x})$ in order to try to (hopelessly) match the whole distribution but sample from $p_{\text{data}}(\boldsymbol{x})$ and then from $p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\omega})$ in order to generate negative particles. So in order for unit variance method to work, our data must have **larger** variance than unit variance and "the level of detail" in our data must have unit st.dev.

In order to fix this, we must update variance parameter σ^2 at every sampling step using Gibbs sampling. First we have $\boldsymbol{w}$ and we sample from $p(\sigma^2|\boldsymbol{\omega},\text{data}) \propto p(\text{data}|\boldsymbol{w},\sigma^2)\; p(\sigma^2)$. After we have sampled from σ^2 we use our previous results to sample using fixed σ^2 from $\boldsymbol{p}(\boldsymbol{w}|\text{data},\sigma^2)$ and store our sample $(\boldsymbol{\omega}_i, \boldsymbol{\sigma}_i^2)$.

We can directly sample from distribution

$p(\sigma^2|\boldsymbol{\omega},\text{data}) \sim X^2(n\,d-1, s^2)$, $\text{s}^2 = \sum_i \|\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i,\boldsymbol{w})\|^2/(n\,d)$

We can sample from this by generating $n\;d-1$, normally distributed zero mean unit variance variables and calculating their squared sum. $Z = \sum_i X_i^2$, after this we scale by s^2.

Sampling covariance matrix $\boldsymbol{\Sigma}$

Similarly, we can do calculate posterior distribution $p(\boldsymbol{\Sigma}|\text{data},\;\boldsymbol{\omega})$ for covariance matrix (InvWishart distribution) and sample from it by using standard multivariate normal distribution inference.

The prior for the normally distributed zero mean data $N(\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{w})|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is zero mean normal distribution $\boldsymbol{\mu} \sim N(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}/\kappa_0)$ and $\boldsymbol{\Sigma}^{-1} \sim \text{Wishart}_{\boldsymbol{v}_0}(\boldsymbol{\Lambda}_0^{-1})$. For mean we decide $\boldsymbol{\mu}_0 = 0$ and $\kappa_0 = \infty$ and we get a posterior distribution for $\boldsymbol{\Sigma}|\text{data}, \boldsymbol{\omega} \sim \text{InvWishart}_{v_n}(\boldsymbol{\Lambda}_n^{-1})$.

$v_n = v_0 + n$

$\boldsymbol{\Lambda}_n = \boldsymbol{\Lambda}_0^{-1} + \sum_i \boldsymbol{z}_i \boldsymbol{z}_i^T$, $\boldsymbol{z}_i = \boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\omega})$

And we can get non-informative prior by taking parameter values $\boldsymbol{\Lambda}_0 = 0$ and $v_0 = -1$. Now we need to sample from InvWishart distribution. This means $\boldsymbol{\Sigma}^{-1} \sim \text{Wishart}_{\boldsymbol{v}_n}(\boldsymbol{\Lambda}_n^{-1})$ and we sample from Wishart distribution. This can be done by sampling $\boldsymbol{\alpha}_i \sim N(\boldsymbol{0}, \boldsymbol{\Lambda}_n^{-1})$ and calculating $\sum_i^{\boldsymbol{v}_n} \boldsymbol{\alpha}_i \boldsymbol{\alpha}_i^T$. (*Bayesian Data Analysis. Andrew Gelman. Appendix A.*)

When number of observations is less than dim $(\boldsymbol{z})$: $n \leqslant \dim(\boldsymbol{z})$ we set identity matrix prior $v_0 = \dim(\boldsymbol{z}) - n$, $\boldsymbol{\Lambda}_0 = \boldsymbol{I}$ and otherwise we use non-informative prior $\boldsymbol{\Lambda}_0 = 0$ and $v_0 = -1$.

In practice we choose $\sigma^2 = \min(\text{eig}(\boldsymbol{\Sigma}))$.

Calculating backpropagration gradient of error term

Calculating gradient of error term $\nabla_{\boldsymbol{\omega}}\, e(\boldsymbol{\omega})$ is complicated because the basic backprogragation algorithm calculates gradient minimizing non whitened error term. The error function:

$$e(\boldsymbol{\omega}) = \tfrac{1}{2}\textstyle\sum_i (\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{w}))^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{y}_i - \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{w}))$$

can be rewritten using $\boldsymbol{W} = \boldsymbol{D}^{1/2} \boldsymbol{X}^T$ where $\boldsymbol{\Sigma}^{-1} = \boldsymbol{X}\, \boldsymbol{D}\, \boldsymbol{X}^T$,

$$e(\boldsymbol{\omega}) = \tfrac{1}{2}\textstyle\sum_i (\boldsymbol{W}\boldsymbol{y}_i - \boldsymbol{W}\boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{w}))^T (\boldsymbol{W}\boldsymbol{y}_i - \boldsymbol{W}\boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{w}))$$

$$\tilde{e}(\boldsymbol{\omega}) = \tfrac{1}{2}\textstyle\sum_i \big(\tilde{\boldsymbol{y}}_i - \tilde{\boldsymbol{f}}(\boldsymbol{x}_i, \boldsymbol{w})\big)^T \big(\tilde{\boldsymbol{y}}_i - \tilde{\boldsymbol{f}}(\boldsymbol{x}_i, \boldsymbol{w})\big)$$

Which have preferred squared error form. However, our neural network now has one extra layer $\mathbf{W}$ which scales and rotates actual output of the function. Therefore we must look into backpropagation algorithm and make slight modifications to calculate gradient when the output layer is changed with additional extra linear layer.

Update rule for local gradient and the update rule for the weights is:

$\boldsymbol{\delta}_{l-1} = \text{diag}(\boldsymbol{\varphi}'(\boldsymbol{v}_{l-1}))\, \boldsymbol{W}_l^T \boldsymbol{\delta}_l$, $\boldsymbol{W}_{ji}^{l-1} = \boldsymbol{\delta}\, \boldsymbol{y}^T$