Machine Learning II

Prof. Dr. Tim Downie

Lecture 11

Mathematical and practical aspects of fitting neural networks

15th December 2023

Berliner Hochschule für Technik

Contents

There is a huge variety of neural network configurations, and there many different programs that fit them.

Each trained neural network has many settings that are either part of the training process or are specified by the user/default settings in the software.

This week we will consider the some of the most important variants of fitting NN models, and tweaks to make the process more efficient.

Some of these features are implemented internally in the software, others are requested by the user by specifying an option in the user's code. The aim olf this lecture is to give a brief insight to what these variations are.

Many of the ideas here are covered briefly in Hastie, Tibshirani & Freedman.

In this lecture we will only considering fully connected networks with feed-forward nodes. This type of NN is often called a Multilayer Perceptron (MLP). Many of the methods discussed are applicable to most types of NN.

Stochastic gradient descent

So far we have assumed that the grad vector $\nabla R(\boldsymbol{\theta})$ is calculated by calculating the partial derivatives over all the (training data) observations.

If the amount of data is large then this can be unnecessarily slow. We will consider two methods which can be employed to speed this up by processing fewer observations at each iteration.

One method goes by the name *stochastic gradient descent*, which means that the data is partitioned into small batches of observations (also called *batch learning*).

Suppose our training data has 10000 observations. We can divided this into 10 batches each with 1000 observations. The grad vector $\nabla R(\boldsymbol{\theta})$ is first calculated using the only first 1000 observations. This grad vector won't be as good as when calculated using all 10000 observations, but it will usually point in an appropriate direction.

The $\boldsymbol{\theta}$ vector is updated and then $\nabla R(\boldsymbol{\theta})$ is calculated using the second batch of 1000 observations. $\boldsymbol{\theta}$ is updated again and so on.
One sweep through all 10 batches is called a **training epoch**.

In one training epoch we get 10 downhill steps for little extra computational effort than for one step not using batches. These 10 approximate steps often outweighs finding the true steepest downhill direction and taking one step.

Some data scientists propose using a batch size of *one* to calculate the grad vector. In this case one training epoch has the same number steps as the are observations.

A similar approach is to use **online training**. The idea behind online training is that a suitable neural network has already been fitted and a new observation becomes available (*comes online*), this is called *dynamic data*. Weights and biasses for the current model are taken as the starting point and then updated.

The online training approach can be used even if the data set is *static*, ie. all of the data is available form the start. The idea is to follow the same procedure as if the data were dynamic. Train a NN on a smallish sample of the training data add a new observation (or a batch of observations) and re-train etc. until all observations have come online.

Learning rate

The iteration in the gradient descent algorithm uses a parameter, s, called the step size. $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - s\nabla f(\boldsymbol{x}_i)$.

In machine learning applications this is called the learning rate.

We saw in the demos and workshop last week that too low a learning rate leads to slow convergence, but too high a learning rate will jump about and not start to converge until a low gradient is found by chance.

There is no easy way to estimate a good value of the learning rate: trial and error works but is not what we want in a machine learning algorithm. It is possible to adjust the value of s by looking at how the loss function decreases after each iteration e.g. if the loss starts increasing then the step size is too large.

One procedure which certainly helps with finding a good learning rate, is to transform all of the variables so that they are on the same scale.

Example: Without scaling, the weights which are linked to a variable defined using km will be larger that if that variable were measured on a mm scale. The grad function for these weights will be correspondingly larger. A good learning rate for the variables on one scale will be poor for variables on another scale.

By scaling the data, a learning rate that is good for one parameter should be acceptable for all parameters.

NB: The scaling should be done using the training data not the full data.

Adaptive learning rate

Hastie, Tibshirani & Freedman suggest using a constant learning rate when using stochastic gradient descent.

When using online learning, they recommend decreasing s “slowly” as the number of observations fitted increases. “Slowly” is formally defined but they give an example of $s_r = \frac{s}{r}$ where s is the initial step size and r is the number of observations currently being fitted.

A more appealing concept is to use *cyclical learning rates*. The value s_i (with i as iteration number) grows and shrinks in a saw tooth manner:

A helpful explanation on this subject by Hafidz Zulkifli including many variations can be found at this website: https://tinyurl.com/y8cqp76f

“Understanding Learning Rates and How It Improves Performance in Deep Learning”

Starting values

Almost all optimisation routines converge to a local minimum (or even to a saddle point).

Only in the special case that the loss function $R(\boldsymbol{\theta})$ is convex will there be only one minimum.

Least squares estimation for a linear model is one example where $R(\boldsymbol{\theta})$ is convex.

If the loss function is a likelihood function then $R(\boldsymbol{\theta})$ is smooth, it is easy to find a local minimum and this local minimum will usually be a “good” solution. The downside is that the likelihood function either requires strict assumptions on the data (such as normally distributed residuals) or Bayesian methods.

With a NN $R(\boldsymbol{\theta})$ is often very bumpy with many local minima. There is no guarantee that a local minimum will give comparable results compared to the best solution.

There is no nice solution to this problem. As with other machine learning methods, e.g. K-Means, you can start with multiple starting points by setting the random number generator seed to different values. In each case fit the NN quickly (with a small max. number of epochs or a relaxed convergence criterion). Find the starting point that gives the smallest value of $R(\boldsymbol{\theta})$ (on the training data) and use this starting point to train the NN fully.

A positive aspect of NNs, is that when a good local minimum in terms of $R(\boldsymbol{\theta})$ is found, it is not a problem if $\boldsymbol{\theta}$ is far away from the perfect set of weights and biasses. This is because we only use NNs for prediction and almost never try to interpret the values of the weights and biasses.

Starting values ctd.

Another issue is that the form of the starting weights can have a significant influence on the run-time of the optimisation algorithm.

The hidden layer activations have the form

$$\begin{aligned}
a_1^{(1)} &= \sigma(z_1^{(1)}) = \sigma(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}) \\
a_2^{(1)} &= \sigma(z_2^{(1)}) = \sigma(w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}) \\
&\dots
\end{aligned}$$

The derivative of $\sigma(v)$ is very small when $|v|$ is large. If the starting values give a large absolute value of v the algorithm takes a long time to move into the area where the current solution can quickly improve, where $|v|$ is small $\rightarrow$ another reason to scale the data.

Suppose all the starting parameters are the same, then all the activation values will be the same within one layer, i.e. $a_1^{(1)} = a_2^{(1)} = a_3^{(1)} = \ldots$, etc.

As MLPs have a symmetric configuration an increase in $a_1^{(1)}$ by an amount 0.1, say, will have the same effect as an increase in $a_2^{(1)}$ by an amount 0.1 etc.

This property is not good as a starting point and can lead to slower convergence.

It is a practical to use a starting point, where each parameter is:

- close to zero, meaning that the $\sigma'(z)$ are relatively large,
- but not equal to zero,
- and is a little bit different from each other parameter.

Assigning a small random number as the starting value for each element is an easy way to implement this.

Activation functions

So far we have just considered the sigmoid function as the activation function

$$\sigma(v) = \frac{1}{1 + e^{-v}}.$$

Other popular alternatives are:

Hyperbolic tangent $\mathtt{tanh}(v) = \frac{sinh(v)}{cosh(v)} = \frac{e^v - e^{-v}}{e^v + e^{-v}}$

tanh has very similar shape to the sigmoid function but maps onto $[-1, 1]$.

ReLU *Rectified linear unit*: $Relu(v) = \begin{cases} v & v \geqslant 0 \\ 0 & v < 0 \end{cases}$

This is very widely used as the default setting because it is fast to compute and often gives good results.

The derivative of Relu(v) is 0 or 1 and has a discontinuity at 0, which is a disadvantage for the steepest descent algorithm.

Activation functions continued

Leaky ReLU $LRelu(v, \alpha) = \begin{cases} v & v \geqslant 0 \\ \alpha v & v < 0 \end{cases}$

This avoids the flat region for $v < 0$ but introduces another hyperparameter to choose α. A standard value is $\alpha = 0.01$.

ELU *Exponential linear unit*: $Elu(v, \alpha) = \begin{cases} v & v \geqslant 0 \\ \alpha(e^v - 1) & v < 0 \end{cases}$

Similar to Leaky ReLU for v positive or small negative,
but converges to $-\alpha$ as $v \to -\infty$.
Because of the e^v component this is no longer quick to compute.

Output activation for classification

From Lecture 7: The predicted probabilities for for class k in NN classification are found using

$$\pi_k(\boldsymbol{x}) = \frac{a_k^{(L)}}{\sum_{j=1}^{K} a_j^{(L)}} \quad \text{with} \quad a_j^{(L)} = \sigma\left(z_j^L\right) = \frac{1}{1 + e^{z_j^{(L)}}}.$$

Note that using the other activation functions on sslides 13 & 14 is not a good idea, as we do not want $\pi_k(\boldsymbol{x})$ to be negative or zero.

Another common alternative is the **softmax** activation, which defines the output activation to be $a_j^{(L)} = e^{z^{(L)}}$.

$$\pi_k(\boldsymbol{x}) = \frac{e^{z_k^{(L)}}}{\sum_{j=1}^{K} e^{z_j^{(L)}}},$$

The difference between these two methods is minimal.

For $K{=}2$, some programs fit just one output node $\pi_1(\boldsymbol{x})$ setting $\pi_2(\boldsymbol{x}) = 1 - \pi_1(\boldsymbol{x})$. This reduces the total number of parameters in the NN.

Hyperparameters

The NN parameters are the weights and biasses in the parameter vector $\boldsymbol{\theta}$.

In addition, the user has to make many choices about the form of the NN model. These are called the **hyperparameters**.

These choices can be:

Numeric – Number of hidden layers, the number of nodes in each hidden layer, learning rate, size of tuning parameter.

Mathematical – type of activation function, type of loss function.

Algorithmic – Type of optimisation algorithm, whether to use stochastic gradient descent or online optimisation.

How do you choose the optimal hyperparameters? This is called hyperparameter **tuning**.

Some choices will come from experience and the actual problem at hand. Other choices especially the number of nodes and the number of hidden layers can be chosen using cross validation.

Neural Network software is may come with routines to test out many different values for the numeric hyperparameters, and suggest the best ones. Such tuning routines still need the user to specify the range of each hyperparameter to investigate.

You saw and example of this with SVMs in Workshop 6 and the function `tune(svmfit, ...)`.

Overfitting

This can be a big problem with NNs.

The NN model is very flexible. You can easily add more hidden layer nodes, and more hidden layers, improving the fit.

One approach is to stop the optimisation routine early. This is not as naive as it first appears.

The elements of $\boldsymbol{\theta}$ start of small, so those elements that should be large will iteratively *grow* towards the optimal value.

Stopping before the coefficients are fully grown has a very similar effect, to starting with the optimal values of $\boldsymbol{\theta}$ and shrinking them towards zero.

This approach is a non-linear version of ridge regression!

Revision: Ridge regression, ML1 week 7

Start with a linear model

$$y_i = \widehat{\beta}_0 + \widehat{\beta}_1 x_{i1} + \cdots + \widehat{\beta}_p x_{ip} + \epsilon_i$$

The beta parameters are the least squares estimates i.e. the parameters which minimise the squared residuals, RSS.

With *ridge regression*, the parameters are the penalised least squares estimates, i.e. minimise

$$RSS + \lambda \sum_{j=1}^{p} \widehat{\beta}_j^2 \;=\; \sum_{i=1}^{n} \left(y_i - \widehat{\beta}_0 - \sum_{j=1}^{p} \widehat{\beta}_j x_i \right)^2 + \lambda \sum_{j=1}^{p} \widehat{\beta}_j^2 .$$

The second term penalises models with large coefficient values.
$\lambda \geqslant 0$ is called the **tuning parameter** and controls how strong the penalty effect is.
When $\lambda = 0$ then the standard least squares estimates for the maximal model is obtained.

A formal (and better) way of fitting a NN than just stopping early, is to modify the loss function to include a penalty term for the squares of the weights and biasses.

$$R(\boldsymbol{\theta}) = \sum_{i \in Tr} (y_i - f(\boldsymbol{x_i}; \boldsymbol{\theta}))^2 + \lambda \sum_{j=1}^{p} \theta_j^2 \tag{1}$$

In neural network terminology this is called **weight decay**, or **L2 regularisation** but is exactly the same as penalised least squares.

Dropout Regularisation

A newer but popular method of minimising overfitting is to use dropout regularisation.

During training: each node in the hidden layer is removed random with probability $(1 - p)$ for one training epoch or one batch.

There are at least two different methods of implementing this. The method proposed by Srivastava et al. 2014[1] is to multiply the output activation from that node by zero for the duration of the drop out. Others suggest setting the weights and biasses equal to zero.

[1] *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Srivastava, Hinton, Krizhevsky, Sutskever & Salakhutdinov (2014)
https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

A side effect of the drop out process is, after training with dropout, the weights and biasses will on average be larger compared to training without dropout.

At the prediction stage, we want to use all the trained weights and biasses so the dropout is turned off. To get the correct preditions each activation/weight/bias has to be multiplied by p which effectively shrinks them.

If back propagation is used with dropout regularisation, the minimisation of the loss function is equivalent to a particular type of regularisation very similar to Equation **??**.

Another easy approach to avoid overfitting is to split the data into 3 groups.

- **Training data**
- **Validation data** and
- **Test data**

The NN fitting is done on the training data.

At each iteration the loss function is calculated for the validation data, and the algorithm stops when the validation loss starts to increase.

The choice of hyperparameters is also based on the validation loss.

The proper estimation of how good the model is then done on the test data. If you want to estimate the true MSE of a model or make a comparison between machine learning methods, you should not use observations used in either the model fitting or the model choice routine.

Neural Network software often obtains the validation results for each training epoch while the model is being trained, and then presented graphically. This can be useful to find when the NN starts to be overfitted.

Summary of terms

- Stochastic gradient descent: Training in batches
- Training epoch: One sweep through all batches
- Static/Dynamic data: Fixed data set or data increases over time
- Online training: Training by adding data as if using a dynamic data set
- Learning rate: Step size in the optimiser
- Activation functions: Sigmoid, tanh, relu, leaky relu and ELU.
- Softmax: a type of output activation used for classification.
- Tuning: finding a good combination of hyperparameters.
- L2 regularisation or weight decay: Penalised Least squares.
- Dropout regularisation: a variant of L2 regularisation

Next Week: Neural Networks for **time series data**.

Workshop 11

This week you will use software to fit and assess regression MLPs.
Next week you will move onto classification MLPs.

There is a discussion at the start of the worksheet comparing the different NN programs available in R.

You will be using *R-Torch* as it is easy to set up and use. If you already have R-Keras set up on your laptop feel free to use Keras for the workshop instead.

With Exercise 4 (College data), I would like you to try out several different options and write down the results, including the mean absolute error (MAE). I would be keen to get your feedback on which methods seem to give the best results.