Machine Learning II
Week 10 Lecture – 18th December 2019
Mathematical an practical aspects of fitting standard neural networks

Contents

- Stochastic gradient descent and online learning
- Learning rate
- Starting values
- Activation functions
- Hyperparameters
- Overfitting

Many researchers have proposed many different ideas to fitting NN models, and there are many different programs to do this.

Each program has it's own implementation. Some aspects can be changed by the user by specifying options others are prechosen by the software developer. Some software is aimed at visually defining a model (Rapidminer). Other software is aimed at numerical efficiency etc.

We will learn some of the most important variants of fitting NN models.

Many of the ideas here are quickly mentioned in Hastie, Tibshirani & Freedman.

We are (today) only considering fully connected networks with feed-forward nodes. This type of NN is often called a Multilayer Perceptron MLP

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. There are 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 data has 1000 observations, which is divided into 10 batches each with 100 observations. The grad vector $\nabla R(\boldsymbol{\theta})$ is first calculated using the first 100 observations and $\boldsymbol{\theta}$ is updated. The grad vector won't be ideal, but it will usually point in approximately the right direction

Then $\nabla R(\boldsymbol{\theta})$ is calculated using the second batch 100 observations and $\boldsymbol{\theta}$ is updated, etc. One sweep through all the batches is called a **training epoch**.

In one training epoch we get 10 downhill steps for little extra computational effort than not using batches. This often outweighs the disadvantage of not finding the true steepest downhill direction.

Some data scientists propose using a batch size of one to calculate the grad vector and looping through all 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*). 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 there is no new data. Train a NN on a smallish sample of the training data add a new observation and re-train etc. until all observations have come online.

The online training approach can also be used for static data. Start with a small batch and then add observations either individually or in batches.

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: long URL

Other optimisation routines

The $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - s\nabla f(\boldsymbol{x}_i)$ gradient descent approach is widely used but there are plenty other minimisation methods. Most are variants on steepest descent, some use back propagation but many use other methods of estimation the gradient.

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 residual) or Bayesian methods.

With a NN $R(\theta)$ is often very wiggly and has many local minima, that may be far away from the best solution.

There is no nice solution to this problem. As with other machine learning methods, e.g. K-Means, you should 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 iterations or a relaxed convergence criterion). Then take the starting point with the lowest value of $R(\theta)$ (on the training data).

One positive with NNs, is that as long as a *good* local minimum is found, it is not a problem, if it is far away from the best solution. This is because we only use NNs for prediction and not to understand the structure of the data.

Starting values ctd.

Another issue is that the form of the starting weights can have a huge 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.

If all the weights are the same then $a_1^{(1)} = a_2^{(1)} = a_3^{(1)} = \ldots$.
The symmetric property of the NN means that, when all the starting coefficients are the same, the starting point is a saddle point, which has a gradient of zero!

It is a very good idea to use a starting point, where each coefficient is different but small. A random number 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. $\sigma(v)$. Other popular alternatives are:

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

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

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

This is very widely used for NNs because it is fast to compute, but the flat output for $x < 0$ is disadvantageous.

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

Avoids the flat region but introduces another hyperparameter to choose α. Standard $\alpha = 0.01$

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

Similar to Leaky ReLU but converges to $-\alpha$ as $x \to -\infty$.

Output activation for classification

From Lecture 8:
The predicted probabilities for neural network classification for class k are found using

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

Another common alternative is the **softmax** activation.

$$\pi_k(\boldsymbol{x}) = \frac{e^{z_k^{(L)}}}{\sum_{k=1}^{K} e^{z_k^{(L)}}}.$$

The difference between these two methods is minimal.

When $K = 2$, some programs fit just one output node $\pi_1(\boldsymbol{x})$ setting $\pi_2(\boldsymbol{x}) = 1 - \pi_1(\boldsymbol{x})$, reducing 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 to choose the optimal hyperparameters?
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.

Overfitting

This can be a real 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 stupid as it first appears.

The elements of θ 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 θ and shrinking them towards zero.

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

Revision: Ridge regression, ML1, Lecture 6

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.

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

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

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.

Note: Often data is split into two groups training and test data, but the hyperparameters or cross validation is made based on the so-called test data. In this case the proper name to use would be training and validation data.

In January

We will consider: