Machine Learning II
Week 9 Lecture – 11th December 2019
Finding the optimal parameters of an artificial neural network

Contents

Recap of Last week

An artificial neural network consist of the following layers:

- ► The input layer which consists of one node for each predictor variable.
- ► Hidden layers, which combine the values of the nodes at the previous layer.
- ► An output layer combines the values of the nodes at the last hidden layer.

For a regression NN with 1 output variable, the output layer is one node.

For a classifier NN there is one node in the output layer for each class.

The NN is defined by the structure of the network **and** the values of the weights and biasses, which in comparison to other ML methods is a large number of parameters.

The parameters should be chosen to minimise the loss function.

Last week we used a naïve method to minimise the loss function, but the convergence was very slow.

The method of minimisation used in practise is back propagation, which is the subject for today.

Optimisation

Minimisation: find the point x such that $f(x)$ reaches its smallest value.

Maximisation: find the point x such that $f(x)$ reaches its largest value.

As maximising $f(x)$ is exactly the same as minimising $-f(x)$, we usually consider just the minimisation case and call the procedure **Optimisation**

x can be a scalar variable (1-dim) or a vector in $\mathbb{R}^n$ (n-dim)

Most numerical optimisation methods find a local optimum rather than a global optimum.

One dimensional optimisation

Let $y = f(x) = 3x^4 + 5x^3 - 20x^2 + 8x + 10$
The classic approach to find the extreme points of f is to obtain the derivative

$$\frac{dy}{dx} = f'(x) = 12x^3 + 15x^2 - 40x + 8$$

set this equal to zero and find the roots.
An extreme point x_0 of a continuous function has the property $f'(x_0) = 0$.

The numerical equivalent to this method is

- Choose a starting point x_0, and a step size s. Set $i = 0$.
- Calculate $f'(x_i)$. This is the gradient of $f(x)$ at x_i.
- Move downhill an amount $sf'(x_0)$. I.e. $x_{i+1} = x_i - sf'(x_i)$.
- Iterate until x_i converges.

Comments

- $f'(x)$ points uphill. We wish to minimise $f(x) \Rightarrow$ downhill $\Rightarrow -f'(x)$.
- This method finds a local minimum (see R Demo). Finding the global minimum is numerically a much harder problem.
- If $f'(x)$ is small the update will not move very far. This causes a problem when near to the optimal solution and when near an inflection point/saddle point.
- We either need to know $f'(x)$ or be able to numerically estimate it. The former is better if this is possible.
- The step size s does not have to be static, it can be *adaptive*, i.e. change with iteration number, depending on the properties of f.

Demonstration 1 in R

A well known version of this optimisation method is the Newton-Raphson Method which sets the step size to $s_i = \frac{1}{f''(x_i)}$.

This of course means that the second derivative must also be known.

Numerically estimating the second derivative is possible but comes with a significant computational cost. The estimation can sometimes be unstable.

Homework exercise: Derive the Newton-Raphson method

Two dimensional optimisation

Now our function takes two inputs and gives one output $f(x_1, x_2)$.
Notation: Set $\boldsymbol{x} = (x_1, x_2)$, so we seek the minimum of $f(\boldsymbol{x})$.

A local minimum has the property that both **partial derivatives** are zero:

$$\frac{\partial f}{\partial x_1} = 0 \quad \text{and} \quad \frac{\partial f}{\partial x_2} = 0.$$

The vector of partial derivatives of f is $\nabla f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{pmatrix}$ and is called “grad f”.

The minimisation algorithm is similar to the one dimensional case, but now downhill is specified by the vector $-\nabla f(\boldsymbol{x})$

Steepest descent algorithm

Let $f(\boldsymbol{x})$ be the 2-dim function to be minimised.

- Determine the partial derivatives of f.
- Choose a starting point $\boldsymbol{x}_0$, and a step size s. Set $i = 0$.
- Calculate $\nabla f(\boldsymbol{x}_i)$.
- Move downhill an amount $s\nabla f(\boldsymbol{x}_i)$. I.e. $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - s\nabla f(\boldsymbol{x}_i)$.
- Iterate until $\boldsymbol{x}_i$ converges.

Demonstration 2 in R for $f(x_1, x_2) = x_1^4 x_2^2 + (x_1 - 1)^2 (x_2 - 1)^4$

Higher dimensional steepest descent

Many machine learning algorithms involve minimising a function associated with a model that has many parameters $f(\boldsymbol{\theta})$, with $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots \theta_{n_\theta})$.

In some special cases there is a non-iterative procedure which gives the optimal parameters directly. In most other cases the minimisation has to be done using an iterative numerical procedure.

It would be good if we can obtain the partial derivatives for each element of $\boldsymbol{\theta}$, i.e. $\nabla f(\boldsymbol{\theta})$.

Unfortunately this is often not the case and the matrix of second derivatives has to be estimated. In the common situation that f is a sum of squared error loss function then Gauß-Newton optimisation is a good algorithm to use.

For neural networks we are in luck! With a bit of work we can specify $\nabla f(\boldsymbol{\theta})$, using a method called back propagation. The work invested is rewarded by a more efficient algorithm.

The principles behind of back propagation

We'll consider the proper maths later!
This section is not mathematically precise, but the general idea is the important thing.

We will assume a regression NN with a squared error loss function.
Adaptations to classification and other loss functions use the same principles.

The NN loss function depends on all our weights and biasses: we'll call them parameters,and represent the current value of each parameter as a vector $\boldsymbol{\theta}$.

$$R(\boldsymbol{\theta}) = \sum_{i \in Tr} (y_i - f(\boldsymbol{x_i}; \boldsymbol{\theta}))^2$$

There are a *lot* of parameters n_θ can easily be over 1000.

Just as in the 2 dimensional function minimisation we will look for the direction with the steepest descent, that is the vector

$$-\nabla R(\boldsymbol{\theta})$$

as this decreases the loss function most efficiently.

Each element of $-\nabla R(\boldsymbol{\theta})$ tells us how sensitive the loss function is to the corresponding parameter.

If $[-\nabla R(\boldsymbol{\theta})]_1$ is 2.4 and $[-\nabla R(\boldsymbol{\theta})]_2$ is 0.8 then
θ_1 has three times the influence over the loss function as θ_2.
Nudging θ_2 by an amount η gives 3 times the improvement than when nudging θ_2 by an amount η.

We will consider the following simple NN

The input data for the first element is $x_1 = 3$ and $x_2 = -2$ with outcome value $y = 1.2$

The output value is $a_1^{(2)} = 0.809$ (level (2) node 1).

To decrease the loss, we would like to increase $a_1^{(2)}$ because $y = 1.2$.

$a_1^{(2)}$ depends on the activations in layer (1) and the weights and biasses between layer (1) and (2). This is computed using the formula

$$a_1^{(2)} = g_1\left(w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + b_1^{(2)}\right).$$

The numeric values in this example are

$$a_1^{(2)} = 0.6 \times 0.475 + 0.2 \times 0.622 + 0.4 \times 1.$$

$g_1(v) = v$ the identity function and is monotone increasing.

This means that an increase in the value in the brackets leads to a larger $a_1^{(2)}$.

$$a_1^{(2)} = g_1\left(w_{11}^{(2)}a_1^{(1)} + w_{12}^{(2)}a_2^{(1)} + b_1^{(2)}\right).$$
$$= 0.6\times 0.475 + 0.2\times 0.622 + 0.4\times 1.$$

If we nudge $w_{11}^{(2)}$ by an amount η then $a_1^{(2)}$ increases by an amount 0.475η

If we nudge $w_{12}^{(2)}$ by an amount η then $a_1^{(2)}$ increases by an amount 0.622η.

If we nudge $b_1^{(2)}$ by an amount η then $a_1^{(2)}$ increases by an amount 1η.

So $b_1^{(2)}$ is more important than $w_{11}^{(2)}$ is more important than $w_{12}^{(2)}$ in terms of minimising the loss function.

In addition:
If we could nudge $a_1^{(1)}$ by an amount η, then $a_1^{(2)}$ increases by an amount 0.6η
If we could nudge $a_2^{(1)}$ by an amount η, then $a_1^{(2)}$ increases by an amount 0.2η
We cannot change the coefficient of the bias parameter $b_1^{(2)}$ because it is fixed at 1.

So $a_1^{(1)}$ is more important than $a_2^{(1)}$.

We can't change these directly.

If, however, we take one step backwards through the network, then we can express these in terms of other parameters which we can change.

This is the step that gives the name back propagation.

$$a_1^{(1)} = \sigma\left(z_1^{(1)}\right).$$
$$\begin{aligned} z_1^{(1)} &= w_{11}^{(1)} x_1 + w_{12}^{(2)} x_2 + b_1^{(1)} \\ &= 0.2\times 3 + 0.4\times(-2) + 0.1\times 1 = -0.1 \end{aligned}$$

A nudge to $w_{11}^{(1)}$ increases $z_1^{(1)}$ by an amount 3η
A nudge to $w_{12}^{(1)}$ increases $z_1^{(1)}$ by an amount -2η (i.e. a decrease)
A nudge to $b_1^{(1)}$ increases $z_1^{(1)}$ by an amount 1η
Because $\sigma()$ is a monotonic increasing function an increase in $z_1^{(1)}$ gives an increase in $a_1^{(1)}$

Feeding this forward to the output again
A nudge to $w_{11}^{(1)}$ increases $a_1^{(2)}$ by an amount 0.449η
A nudge to $w_{12}^{(1)}$ increases $a_1^{(2)}$ by an amount -0.299η
A nudge to $b_1^{(1)}$ increases $a_1^{(2)}$ by an amount 0.150η
We see that $w_{11}^{(1)}$ is more important than $w_{12}^{(1)}$ is more important than $b_1^{(1)}$ in terms of minimising the loss function.

The second node in the hidden layer is similar.

A nudge to $w_{21}^{(1)}$ increases $z_1^{(2)}$ by an amount 3η
A nudge to $w_{22}^{(1)}$ increases $z_1^{(2)}$ by an amount -2η
A nudge to $b_2^{(1)}$ increases $z_1^{(2)}$ by an amount 1η

Feeding this forward
A nudge to $w_{21}^{(1)}$ increases $a_1^{(2)}$ by an amount 0.141η
A nudge to $w_{22}^{(1)}$ increases $a_1^{(2)}$ by an amount -0.094η
A nudge to $b_2^{(1)}$ increases $a_1^{(2)}$ by an amount 0.045η

Finally the importance of the first element is determined by the residual $e_1 = y - a_1^{(2)} = 0.311$. How effective each nudge is has to be multiplied by $e_1 = 0.311$

The contribution of the first element to $-\nabla R(\boldsymbol{\theta})$ is the vector
$0.311\times(0.449, -0.299, 0.150, 0.141, -0.094, 0.045, 0.455, 0.622, 1)^T$

Consider now the second observation with $x_1 = -1$ and $x_2 = 1$ with outcome value $y = 0.7$.

The output activation is $a_1^{(2)} = 0.88$ so the second residual is $e_2 = -0.18$. Our wish is to decrease $a_1^{(2)}$ for the second element.

How effective is a change in each parameter to decrease the cost? The process is exactly the same, the values for the output node are:

$$\begin{aligned} a_1^{(2)} &= g_1\left(w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + b_1^{(2)}\right). \\ &= 0.6 \times 0.574 + 0.2 \times 0.689 + 0.4 \times 1. \end{aligned}$$

giving:
Nudging $w_{11}^{(2)}$ increases $a_1^{(2)}$ by an amount 0.455η.
Nudging $w_{12}^{(2)}$ increases by an amount η then $a_1^{(2)}$ increases by an amount 0.622η.
Nudging $b_1^{(2)}$ increases by an amount η then $a_1^{(2)}$ increases by an amount 1η.

We do this for every element in the data set and take the sum of all the resulting nine vector elements giving us the value of $-\nabla R(\boldsymbol{\theta})$.

This gives us the steepest downward direction in nine dimensions, and we take a step of length η and update our current position of $\boldsymbol{\theta}$.

Differentiation: Chain rule

Before we look at the mathematics of the back propagation algorithm we should do a little revision of the chain rule in differentiation.

There are two notation systems for differentiation, both are useful in different ways.

One notation is to express a function as $f(x)$ and the derivative as $f'(x)$ (Lagrange's notation)

The "Leibniz notation" expresses the value of the function as y (e.g. $y = 2^2 + 4$) and the derivative of y with respect to x is $\frac{dy}{dx}$

For the chain rule the Leibniz notation is much easier to understand, especially for a chain with many links.

Chain rule with two links

If we can express y as a function in v, and v as a function in x Then the derivative of y w.r.t. x is[1]

$$\frac{dy}{dx} = \frac{dy}{dv}\frac{dv}{dx}$$

Usually the two derivatives on the right will be easier to determine than the one on the left directly.

In the Lagrange notation we write $f(x) = u(v(x))$, and the chain rule is

$$f'(x) = u'(v(x))\, v'(x).$$

[1] w.r.t. means "with respect to"

Example of the chain rule

Let $R(\theta) = (y - \theta)^2$, find $\dfrac{dR}{d\theta}$.

Put $u = (y - \theta)$

$$R = u^2 \qquad \frac{dR}{du} = 2u \qquad \frac{du}{d\theta} = -1$$

$$\frac{dR}{d\theta} = \frac{dR}{du}\frac{du}{d\theta} = -2u$$

$$\frac{dR}{d\theta} = -2(y - \theta) \tag{1}$$

Chain rule with three links

Express y as a function of u and u as a function of v and v as a function in x

Then the derivative of y w.r.t. x is

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dv}\frac{dv}{dx}$$

In the Lagrange notation we write $f(x) = w(u(v(x)))$, and the chain rule is

$$\Rightarrow \qquad f'(x) = w'(u(v(x)))u'(v(x))\,v'(x).$$

The back propagation formulae for a NN

The Loss function for one arbitrary observation is

$$R = (y - a_1^{(2)})^2$$

To use the steepest descent algorithm we need to find $\nabla R(\boldsymbol{\theta})$, so we need to find the partial derivative of R w.r.t. each of the parameters in the NN.

The output is

$$a_1^{(2)} = g_1\left(z_1^{(2)}\right) = z_1^{(2)} = w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + b_1^{(2)}$$

As g_1 is the identity function we can remove it from the equation.

By how much does R change, when we change the parameter $w_{11}^{(2)}$ by a small amount η?

The answer is $\eta \frac{\partial R}{\partial w_{11}^{(2)}}$

$$\frac{\partial R}{\partial w_{11}^{(2)}} = \frac{\partial R}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial w_{11}^{(2)}} \quad \text{Chain rule}$$

$$\frac{\partial R}{\partial a_1^{(2)}} = -2(y - a_1^{(2)}) \quad \text{eqn 1}$$

$$\frac{\partial a_1^{(2)}}{\partial w_{11}^{(2)}} = a_1^{(1)}$$

$$\frac{\partial R}{\partial w_{11}^{(2)}} = -2\left(y - a_1^{(2)}\right) a_1^{(1)}$$

All the terms on the right hand side are known variables in the current state of the NN.

If we nudge $w_{11}^{(2)}$ by an amount η then the loss for this observation will change by an amount

$$-\eta 2\left(y - a_1^{(2)}\right) a_1^{(1)} = -\eta \frac{\partial R}{\partial w_{11}^{(2)}}$$

giving the $w_{11}^{(2)}$ component of $\nabla R(\boldsymbol{\theta})$ for the steepest descent algorithm.

Homework: Find

$$\frac{\partial R}{\partial w_{12}^{(2)}} \quad \text{and} \quad \frac{\partial R}{\partial b_1^{(2)}}$$

While we are still at layer 2, we will determine $\frac{\partial a_1^{(2)}}{\partial a_1^{(1)}}$ and $\frac{\partial a_1^{(2)}}{\partial a_2^{(1)}}$ because we will need them in a moment.

$$\frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} = w_{11}^{(2)} \qquad \text{and} \qquad \frac{\partial a_1^{(2)}}{\partial a_2^{(1)}} = w_{12}^{(2)}$$

We have found a way to compute $\nabla R(\theta)$ for all of the parameters in the second layer.

We now propagate back one layer.
Our aim is to find $\frac{\partial R}{\partial w_{11}^{(1)}}$ and the similar partial derivatives for all the first layer parameters.

$$a_1^{(1)} = \sigma(z_1^{(1)}) = \sigma(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)})$$

$$\begin{aligned}
\frac{\partial R}{\partial w_{11}^{(1)}} &= \frac{\partial R}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} && \text{chain rule} \\
&= \frac{\partial R}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} && \text{chain rule again} \\
&= \frac{\partial R}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} && \text{chain rule yet again}
\end{aligned}$$

Each component in the last equation is

$$\frac{\partial R}{\partial a_1^{(2)}} = -2(y - a_1^{(2)}) \qquad \text{eqn 1}$$

$$\frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} = w_{11}^{(2)} \qquad \text{eqn chain rule}$$

$$\frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} = \text{see below}$$

$$\frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} = x_1$$

$a_1^{(1)} = \sigma(z_1^1)$: The sigmoid function $\sigma(v) = (1 + e^{-v})^{-1}$ has derivative

$\frac{d\sigma}{dv} = e^{-v}(1 + e^{-v})^{-2}$ homework!

$$\frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} = e^{-z_1^{(1)}} \left(1 + e^{-z_1^{(1)}}\right)^{-2}$$

Putting all these components together and with the

$$\frac{\partial R}{\partial w_{11}^{(1)}} = -2\left(y - a_1^{(2)}\right) w_{11}^{(2)} z_1^{(1)} \left(1 + e^{-z_1^{(1)}}\right)^{-2} x_1$$

Yes this is a long equation to write, but is not difficult to compute. It helps for out steepest descent algorithm, how far to step in the $w_{11}^{(1)}$ axis of our long $\boldsymbol{\theta}$ vector.

A similar technique allows us to obtain all the remaining partial derivatives in $\nabla R(\boldsymbol{\theta})$

The partial derivatives can be coded as a kind of “plug and play approach”, making it relatively easy to code, e.g. every element of $\nabla R(\boldsymbol{\theta})$ has the factor $-2\left(y - a_1^{(2)}\right)$

Summary: Back propagation

- Our neural net has a starting parameters $\boldsymbol{\theta} = (w_{11}^{(1)},\ w_{12}^{(1)}, \ldots,\ b_1^{(2)})$.
- The cost for this starting point called the loss function $R(\boldsymbol{\theta})$.
- We iteratively find better position vectors $\boldsymbol{\theta}$ using a steepest descent optimisation routine.
- Each iteration tells us to make one step downhill in the best direction.
- The best direction is $-\nabla R(\boldsymbol{\theta}) = -\begin{pmatrix} \frac{\partial R}{\partial w_{11}^{(1)}} \\ \vdots \\ \frac{\partial R}{\partial b_1^{(2)}} \end{pmatrix}$.
- The magnitude of each component in this $\nabla R(\boldsymbol{\theta})$ tells which parameters are the most important in terms of reducing R
- Each component of $\nabla R(\boldsymbol{\theta})$ is computed by repeatedly applying the chain rule.