Using a single weight matrix for Back-Propagation in Neural Networks

In my Neural Network I have combined all of the weight matrices into one large matrix: e.g A 3 layer matrix usually has 3 weight matrices W1, W2, W3, one for each layer. I have created one large weight matrix called W, where W2 and W3 are appended onto the end of W1. If W1 has 3 columns, W2 has 3 columns, and W3 has 2 columns, my matrix W will have 8 columns.

The number of layers/ Number of Inputs/Outputs is stored as a global variable.

This means I can use the feedforward code with only 2 input arguments, as the feedforward code splits W up into W1, W2, W3...etc, inside the function.

```
Output_of_Neural_Net = feedforward(Input_to_Neural_Net,W)
```

I also store the training data as a global variable. This means I can use the cost function with only one input argument.

```
cost = costfn(W)
```

The purpose of this is so that I can use built in MATLAB functions to minimise the cost function and therefore obtain the W that gives the network that best approximates my training data.

I have tried `fminsearch(@costfn,W)` and `fminunc(@costfn,W)` . Both give mediocre results for the function I am trying to approximate, although `fminunc` is slightly better.

I now want to try Back-Propagation to train this network, to see if it does a better job, however most implementations of this are for networks with multiple weight matrices, making it more complicated.

My question is: Will I be able to implement back propagation with my single appended weight matrix, and how can I do this?

I feel like using a single weight matrix should make the code simpler, but I can't work out how to implement it, as all other examples I have seen are for multiple weight matrices.

Additional Information

The network will be a function approximator with between 8 and 30 inputs, and 3 outputs. The function it is approximating is quite complicated and involves the inverse of elliptic integrals (and so has no analytical solution). The inputs and outputs of the network will be normalised so that are between 0 and 1.

matlab | machine-learning | neural-network | mathematical-optimization | backpropagation

edited Mar 10 at 20:46

asked Mar 10 at 18:59

Blue7

233 10

add comment

start a bounty

1 Answer

There are several problems with the approach you are describing.

First, from what you have described, there really is no simplification of the feed-forward code or the backpropagation code. You are just combining three weight matrices into one, which allows the `feedforward` and `costfn` functions to take fewer arguments but you still have to unpack `W` inside those functions to implement the forward and backpropagation logic. The feedforward and backpropagation logic requires evaluation of the activation function and its derivative in each layer so you can't represent this as a simple matrix multiplication.

The second issue is that you are constraining the structure of your neural network by packing three weight matrices into one by appending columns. The number or rows and columns in the weight matrix correspond to the number of neurons and inputs in the layer, respectively. Suppose you have `M` inputs to your network and `N` neurons in the first layer. Then `W1` will have shape `(N, M)` . In general, for a fully-connected network, layer two weights (`W2`) will have shape `(K, N)` , where `N` is the number of inputs (which is constrained by the number of outputs from the first layer) and `K` is the number of neurons in the second layer.

The problem is that since you are creating one combined weight matrix by appending columns, `K` (the number of rows in the second weight matrix) will have to be the same as the number of rows/neurons from the first layer, and so on for successive layers. In other words, your network will have shape `M x N x N x N` (`M` inputs, then `N` neurons in each layer). This is a bad constraint to have on your network since you typically don't want the same numbers of neurons in the hidden layers as in the output layer.

Note that for simplification, I have ignored bias inputs but the same issues exist even if they are included.

edited Mar 10 at 19:46

answered Mar 10 at 19:41

bogatron
3,882 1 5 13

Thanks for your answer. Yes I have to unpack `W` yo implement the `feedforward` function, and as `costfn` has the `feedforward` function inside it this does to. However it is the only way to use the `fminsearch` and `fminunc` functions as these can only optimise a single matrix. I think I understand why I can't do this for the back-prop algorithm though. And yes, I am constraining the structure of my network, but the only constraint is that the number of hidden neurons in each layer is the same as the number of inputs. I can have a different number of neurons in the output layer... – Blue7 Mar 10 at 20:00

...because the matrix W can be unpacked into different size matrices inside the feedforward code. The global variables define how the matrix is split. I have heard that having the same number of hidden neurons as inputs is a typically good starting point, is this not true? – Blue7 Mar 10 at 20:03

For example, `W1` could be a `(5 x 4)` matrix (4 inputs + 1 bias node, connecting to 4 hidden neurons in the next layer). `W2` the same, and then `W3` a `(5 x 2)` matrix (4 previous neurons + 1 bias node, connecting to 2 outputs). These are then be appended into a single `(5 x 10)` matrix, ready for optimisation. – Blue7 Mar 10 at 20:11

Ok, then you have your rows/columns swapped relative to my response. But that then means that your network is constrained to have a shape like `N x N x N x M`, where the first `N` is the number of inputs and `M` (number of outputs) is 2. How many neurons to use per layer depends on the nature of the problem, number of classes, etc. In some cases you want significantly more neurons in the first layer to provide many decision boundaries for the subsequent layers to use. In other cases (or layers), you may want fewer neurons to reduce the dimensionality of the data space. – bogatron Mar 10 at 20:24

1 It isn't very hard. You can randomize the weights, based on the input/output range of your activation function. This is easiest if you also scale your inputs to that range. If you can code one hidden layer, you can do 1,000 just as easily so make it generic. Only the output layer is special (regarding backpropagation). The feedforward part is easy: matrix multiplication, then apply activation function. Your question is tagged `matlab` but here is an example in python. – bogatron Mar 10 at 21:06

add / show **1** more comment

Not the answer you're looking for? Browse other questions tagged matlab | machine-learning | neural-network | mathematical-optimization | backpropagation **or ask your own question.**