Hadamard with neural networks

Morgan Maddren
mmaddren@soe.ucsc.edu
6/13/2013
Introduction

The Hadamard problem is difficult for gradient descent based algorithms to solve. Neural networks are not immune to this problem either, since they fundamentally use gradient descent as well. I first discuss why standard backpropagation-based learning in neural networks is insufficient to learn the Hadamard problem. Then I explore some modifications to standard backpropagation and an improved algorithm called RProp. I then develop a multiplicative update based on RProp that can solve the Hadamard for deep neural networks.

The problem to learn is an arbitrarily chosen column in a Hadamard matrix. The learner is given $k$ randomly chosen examples from an $n \times n$ matrix labelled by a single column chosen randomly beforehand. Gradient descent based approaches should not be able to achieve better than $1 - \frac{k}{n}$ squared error averaged over $n$ examples. In practice, a predictor will either learn the pattern exactly, and the error will be exceedingly small, or the learner will fail to learn the pattern and the error will hover above the $1 - \frac{k}{n}$ line.

The reason that this problem is so hard is that because the rows of the Hadamard matrix are orthogonal and the solution is a single column, no single example will ever uniquely identify which column the label came from. For this reason, even the best algorithm (the class of exponentiated gradient algorithms) will require in an $n \times n$ matrix at least $\lg(n)$ examples in the best case (but often more) to uniquely identify the correct column. For every row except the first row, half of the features are 1 and half are -1, and the same for the columns. Because the value the rows is constrained to {-1, 1}, multiple columns will do well for a given target, though only a single column will get 100% correctness. The average error for any column will be 0.5 on randomly chosen rows. Half of all columns will do equal to or better than 50% correctness however, which is where the difficulty of the problem lies. Gradient descent is unable to choose the single best because it is able to invent features that sum to the right value enough of the time to hit a local minima. The goal then is to find a way to modify neural network learning such that it is capable of learning the pattern as fast as exponentiated gradient descent.
Looking at RProp

To solve the Hadamard, I started with the RProp algorithm\(^2\). RProp, or “resilient backpropagation”, uses an independent learning rate ‘step value’ for each weight which is multiplied by a constant eta depending on whether it should be increased or decreased. RProp empirically shows significant improvement over traditional backprop, as it is able to move somewhat independently of the gradient, and the step value increases multiplicatively. RProp works by keeping track of the previous iteration’s gradients. When it computes a new gradient for a given weight, it compares the gradient’s signs. If the two signs are the same, it multiplies the step value by \(\text{eta}^+\), typically 1.2 \(^2\). If the signs don’t match, then it multiplies the step value by \(\text{eta}^-\), typically 0.5 \(^2\). It then applies the step in the direction (sign) of the current gradient.

For each \(w_{i,j}\):

- If \(\text{delta}_{i,j,t} \times \text{delta}_{i,j,t-1} > 0\): //signs are the same
  \n  \[\text{step}_{i,j,t} = \text{step}_{i,j,t-1} \times \text{eta}^+ //\text{typically 1.2}\]

- Else if \(\text{delta}_{i,j,t} \times \text{delta}_{i,j,t-1} < 0\): //signs differ
  \n  \[\text{step}_{i,j,t} = \text{step}_{i,j,t-1} \times \text{eta}^- //\text{typically 0.5}\]
  \[\text{delta}_{i,j,t} = 0 //\text{optional step for "iRProp"}\]

\[w_{i,j,t+1} = w_{i,j,t} - \text{sign(\text{delta}_{i,j,t})} \times \text{step}_{i,j,t}\]

Although RProp seems to capture some of the essence of multiplicative updates, it’s not quite there and it still falls short of being able to solve the Hadamard problem, ending up performing almost identically to traditional backpropagation. This is because although it may converge faster, it still keeps doesn’t overcome all of backpropagation’s limitations. The basic logic of RProp is very simple: if the gradient looks like you’re going in the right direction, you want to increase the weight, if it looks like you’ve overshot, you want to decrease the weight.

RProp updates independently of the gradient, so it is a very simple algorithm. Yet it converges much quicker than normal backpropagation. Because of these reasons, I believed this was a good starting place in looking for a way to solve the Hadamard matrix.
Determining the multiplicative update

To create a multiplicative update, I started with RProp’s weight updates, and then converted the update to a multiplicative one by finding the multiplicand that would have resulted from the additive update:

\[
\text{update} = \frac{(w_{i,j,t} + \text{rpropOutput})}{w_{i,j,t}}
\]

\[
w_{i,j,t+1} = w_{i,j,t} \times \text{update}
\]

After I had created a pseudo-multiplicative update, I gradually mixed in small amounts of other values. After inspecting the weight updates with numerous strategies, I also enforced the constraint that the weights must be positive. I believe this might be overfitting to the constraints of the problem however, and I am not entirely happy that I had to take this step.

After enforcing this constraint, I found that if I use the same eta+ and eta- from RProp, but simply multiply the weight by them instead, the neural network did converge and in fact also solved the Hadamard problem.

For each \(w_{i,j}\):

- If \(\delta_{i,j,t} \times \delta_{i,j,t-1} > 0\):
  \[
  w_{i,j,t+1} = w_{i,j,t} \times \text{eta}^+ //1.2
  \]

- Else if \(\delta_{i,j,t} \times \delta_{i,j,t-1} < 0\):
  \[
  w_{i,j,t+1} = w_{i,j,t} \times \text{eta}^- //0.5
  \]

- If \(w_{i,j,t+1} \leq 0\):
  \[
  w_{i,j,t+1} = w_{\min} //\text{very small value like 0.001}
  \]

Both the multiplicative update and the non-negativity of the weights are necessary to achieve an algorithm that can learn Hadamard, neither is sufficient on its own.
Results

I ran multiple learning algorithms over a neural network with 8 input nodes, 8 hidden nodes, and 1 output node. Other configurations of networks gave similar results. The multiplicative update outperforms the other updates in all cases, and stays below the $1-k/n$ line for the majority of the time. Each algorithm was trained until the error on the test data was below a certain threshold (0.001) and then tested on the entire set of data.
I also tested this approach on larger random matrices. It seems like RProp does better in this case, but the multiplicative approach still beats it. In this case, the network had 256 input nodes, 256 hidden nodes, and 1 output node. As before each algorithm was trained (except backpropagation didn’t converge fast enough and had to be stopped after 100 epochs, which explains why it doesn’t go down as fast as you might expect).
Conclusion

This multiplicative update approach is somewhat limited: by enforcing non-negative weights, I can no longer solve problems like XOR. I wish to see if I can use a positive and negative weight similar to EG+/− [3] to overcome this problem, but I was not able to complete that analysis by the time of this writing.

If the negative weight problem can be solved with this approach, then this update algorithm could be a good general purpose learner. I think it would also be good to see if there were any good ways to change the learning rate, although RProp is a good example that the learning rate being independent of the gradient isn’t necessarily bad. So then, are these results good enough to earn me that beer?
References