Andrew Gibiansky :: *Math* → [Code]

Recurrent Neural Networks

Friday, March 21, 2014

We've previously looked at backpropagation for standard feedforward neural networks, and discussed extensively how we can optimize backpropagation to learn faster. We've also looked at convolutional neural networks, neural networks which can exploit the geometric layout of the input data.

Now, we'll extend these techniques to neural networks that can learn patterns in sequences, commonly known as *recurrent neural networks*. These are used for data which is inherently ordered and context sensitive, where later elements of the sequence depend on previous ones. Examples include text and audio.

Mathematical Formulation

Recurrent neural networks learn from *sequences*. A sequence is defined as a list of (x_i, y_i) pairs, where x_i is the input at time i and y_i is the desired output. Note that that is a *single* sequence; the entire data set consists of many sequences.

In addition to the data in our data set, each time step has another input: the hidden state h_{i-1} from the previous time step. In this way, the recurrent neural network can maintain some internal context as it progresses forward in the sequence. Thus, to summarize, at time i the recurrent network has:

- Input vector x_i (data)
- Output vector y_i (data)
- Predicted output vector $\hat{y}_i$ (computed through forward propagation)
- Hidden state h_i

When looking only at a single timestep, the recurrent network looks like a simple one-hidden-layer feed forward network. It has an input layer for x_i, an output layer for y_i, and another input layer for the previous hidden state h_{i-1}. Finally, it has one hidden layer between these. The only unusual thing is that we have two input layers; both of the input layers are are connected to the hidden layer as if they were really just a single layer.

Thus, we have three separate matrices of weights:

- Input-to-hidden weights W_{hx}
- Hidden-to-hidden weights W_{hh}
- Hidden-to-output weights W_{yh}

As you may easily guess, the forward propagation equations for this network are quite simple:

$$h_i = \sigma(W_{hh} h_{i-1} + W_{hx} x_i + b_h)$$
$$\hat{y}_i = W_{yh} h_i$$

There are several things to note here. First of all, note that the predicted outputs are not subject to the nonlinearity. We may want to predict things other than the things in the range of the nonlinearity, so instead we do not apply the nonlinearity. For specific use cases of recurrent nets, this can be amended, and a nonlinearity specific to the problem can be chosen. Finally, note that these equations are the same as the equations for a single hidden layer feed forward network, with the caveat that the input layer is broken into two pieces x_i and h_{i-1}.

Backpropagation Through Time

In order to learn the weights for our recurrent network, we have to generalize our backpropagation algorithm to neural networks. Since this backpropagation algorithm works on sequences in time, it is known as *backpropagation through time*. We can derive it in the same way we did before - define an error metric and then take partial derivatives of the error with respect to the weights. Alternatively, we can also just base our algorithm off the normal backpropagation algorithm by expanding the neural network in time.

For each training epoch, we start by training on shorter sequences, and then train on progressively longer sequences. If our max sequence length is N, then we will first train on all 1-length sequences (the first element only), then all 2-element sequences (the first element followed by the second element), and so on, until we are training on N-length sequences.

For each length of sequence k, we unfold the network into a normal feedforward network that has k hidden layers. However, unlike a normal hidden layer, each hidden layer also takes an input - the input into the neural network at that time step. Thus, the network actually has $k + 1$ different inputs: an initial hidden state and k inputs, one per time step.

At this point, we can proceed with standard backpropagation. The only issue is that when we do backpropagation, the weights between each layer will change in different ways. In order to enforce that the weights are the same (which they must be, since we actually have a network through time, which generalizes to any number of steps), we just average the weight updates at each layer.

This algorithm can be summarized as follows:

Backpropagation Through Time:

Initialization: Initialize weight matrices W_{hx} (input-to-hidden weights), W_{hh} (hidden to hidden weights), and W_{yh} to random values.

Repeat until convergence:

1. Let i be the current sequence length, starting at 1.
2. Unfold your neural network in time for i time steps to create a standard feed-forward net.
3. Set the inputs to your feed-forward net to be the initial hidden state (a vector of zeros) and the input at every time step. Note that your network will have inputs at every hidden layer as well as at the initial input layer.
4. Perform forward and backward propagation as in a normal network.

5. Average all the gradients at each layer, so that the weight update to the matrices is the same in each time step.

6. Repeat steps 1 through 5 for $i = 1$ through $i = N$, where N is the maximum number of elements in your sequence data set.

Structural Damping

Recall that applying Hessian-free optimization, at each step we proceed by expanding our function f about the current point out to second order:

$$f(x + \Delta x) \approx \tilde{f}\,(x + \Delta x) = f(x) + f'(x)^T \Delta x + \Delta x^T H \Delta x,$$

where H is the Hessian of f. In order to ensure that this function actually has a minimum, we must have H be positive semidefinite, so we replace H by a positive semidefinite approximation of the Hessian known as the Gauss-Newton matrix G.

However, even with this second order approximation, problems can arise. If conjugate gradient strays too far from the original x, the curvature estimate G becomes innacurate, and may lead to worse convergence (or even steps in the wrong direction). In order to combat this problem, Martens suggests a method of *structural damping*.

Structural damping reduces this problem by penalizing large deviations from x. Thus, instead of having the objective function $\tilde{f}\,(x)$, the objective function is instead given by

$$\tilde{f}_{\,d}\,(x + \Delta x) = \tilde{f}\,(x + \Delta x) + \lambda ||\Delta x||^2$$

This penalizes large deviations from x, as $||\Delta x||^2$ is the magnitude of the deviation. The parameter λ allows one to tune how strong this penalty is. Also, note that since we can rewrite

$$\lambda ||\Delta x||^2 = \lambda \Delta x^T I \Delta x,$$

we can incorporate this easily into our original objective simply by adding a diagonal matrix to G:

$$\tilde{f}_{\,d}\,(x + \Delta x) = f(x) + f'(x)^T \Delta x + \Delta x^T (G + \lambda I) \Delta x.$$

The choice of λ, however, is a fairly difficult topic. Since the scale of G may change, a constant λ is inappropriate, as it may be too big (in which case it would override the second order elements of G) or too small (in which case it wouldn't matter). Instead, Marten's recommends a Levenberg-Marquardt style heuristic (named after the heuristics used in the Levenberg-Marquardt algorithm, which uses a similar damping parameter). The heuristic starts with $\lambda = \lambda_0$ (for some previously chosen λ_0) and at each step scales λ either up or down to match the scale of G.

Specifically, Martens suggests using the reduction ratio ρ:

$$\rho \equiv \frac{f(x + \Delta x) - f(x)}{\tilde{f}\,(x + \Delta x) - \tilde{f}\,(x)}$$

This reduction ratio measures how accurate the approximation $\tilde{f}$ for f is. Before taking a step to $x + \Delta x$, compute the reduction ratio. If the reduction ratio is high ($\rho > \frac{3}{4}$), then move to $x + \Delta x$

and reduce λ by multiplication by $\frac{2}{3}$. If the reduction ratio is low ($\rho < \frac{1}{4}$), do *not* move to $x + \Delta x$, but instead increase λ by multiplication by $\frac{3}{2}$, and try again. Finally, if ρ is between these two thresholds, move to $x + \Delta x$, but do not adjust λ. This heuristic constantly adjusts λ such that the quadratic approximation remains reasonable.

Before moving on to modifications to structural damping, let us recap:

> ***Structural Damping***:
>
> **Initialization**: Start out with $\lambda = \lambda_0$.
>
> **Compute** ρ: Before each step of conjugate gradient starting at x going in the direction Δx, compute the *reduction ratio*
>
> $$\rho \equiv \frac{f(x + \Delta x) - f(x)}{\tilde{f}(x + \Delta x) - \tilde{f}(x)}.$$
>
> **Adjust** λ **and** x:
>
> 1. If $\rho > \frac{3}{4}$, update λ via $\lambda \leftarrow \frac{2}{3}\lambda$ and x via $x \leftarrow x + \Delta x$.
> 2. If $\rho < \frac{1}{4}$, update λ via $\lambda \leftarrow \frac{3}{2}\lambda$, but do not update x (do not take the step).
> 3. Otherwise, if $\frac{1}{4} \leq \rho \leq \frac{3}{4}$, do not update λ but update x via $x \leftarrow x + \Delta x$.
>
> **Iterate**: Repeat the computation of ρ and update of λ for each step of the conjugate gradient. Every time conjugate gradient starts anew, reset λ to λ_0.

Structural Damping for Recurrent Networks

In applying Hessian free optimization to recurrent neural networks, Martens and Sutskever found that the structural damping could be improved significantly by taking a different approach.

In recurrent networks, the weight matrices are used at every time step, meaning that the same matrix is used many many times in different places. As a result, a very small perturbation to this matrix can result in a *very* large different output difference. This yields the intuition that in addition to penalizing the size of the deviation via $\lambda \Delta x^T \Delta x$, we should also penalize causing a large deviation in the hidden state via some $\mu S_x(\Delta x)$.

To make $S_x(\Delta x)$ penalize large changes in the hidden state, we can define it as

$$S_x(\Delta x) = D(h(x), h(x + \Delta x)),$$

where h is the output of the hidden layer neurons (the hidden state) and D is a distance function. (Note that this should be summed over *all* the hidden layer outputs, at each time step *i*.) Using this penalty, note that a devition Δx will be penalized more if it yields a larger distance between the previous hidden layer outputs and the new hidden layer outputs, and that this will increase the stability of each step.

However, while $\lambda \Delta x^T \Delta x$ is quadratic and can be included via just adding λI to the quadratic optimization objective, $S_x(\Delta x)$ is not, in general, a quadratic function of Δx. In order to include it in our quadratic objective, we expand it to second order about x, as we did with our original function f:

$$S_x(\Delta x) \approx S_x(x) + \Delta x^T S_x'(x) + \Delta x^T H \Delta x.$$

However, since S_x is a distance function, $S_x(x)$ is clearly zero; similarly, the minimum of S_x is clearly achieved at x itself, which means that $S_x'(x)$ must also be zero. Thus, our approximation reduces to

$$S_x(\Delta x) \approx \Delta x^T H \Delta x.$$

In order to maintain the positive definiteness of our optimization objective, we do the same thing we did before and replace the Hessian with the Gauss-Newton matrix G, leading to our final approximation

$$S_x(\Delta x) \approx \tilde{S}_x(\Delta x) = \Delta x^T G_{S_x} \Delta x.$$

Now that we have established our structural damping as a modification to our quadratic object, we must figure out how to compute the structural damping penalty. In order to do so, we simply add an output to our recurrent neural network: in addition to outputting $\hat{y}_i$ at each time step, the neural network also outputs the current value at the hidden layer, namely h_i. Then, we can include our structural damping simply by including an error term for every h_i in the final error, where the error for h_i at the candidate point $x + \Delta x$ is the distance $D(h_i(x), h_i(x + \Delta x))$. It turns out that including this extra error term is fairly simple, and only requires computing the Hessian of $D \circ \sigma$, and that everything else is already done anyways (as it is needed for the Gauss-Newton matrix-vector product for the normal outputs). In the original paper, Martens and Sutskever work through this slightly more rigorously and provide specific algorithms that result from these techniques for different activation and erro functions.

With these modifications, Hessian-free optimization becomes quite practical for the purposes of training recurrent neural networks for sequences with many timesteps and long-range contextual effects.

Multiplicative Recurrent Networks

The formulation above is the standard formulation of recurrent neural networks. However, alternative approaches have certainly been proposed. Recently, for instance, Sutskever et. al. proposed a *multiplicative recurrent neural network* architecture for generating text.

In this formulation, they claim that a limitation of recurrent networks is that the way the hidden state advances cannot be modified by the input. In their application, they are attempting to generate text from a recurrent neural network, and the input at each step is a single character. Ideally, they would be able to use a separate hidden-to-hidden weight matrix for each possible input, as follows:

$$\begin{aligned} W_i &= \sum x_i^j W_{hh}^{(j)} \\ h_i &= \sigma(W_i h_{i-1} + W_{hx} x_i + b_h) \\ \hat{y}_i &= W_{yh} h_i \end{aligned}$$

where W is the current weight update matrix, x_i^j is the jth component of x_i, and $W_{hh}^{(j)}$ is the jth W_{hh} variant. ($W_{hh}^{(j)}$ is thus a third rank tensor.)

However, for even moderately large hidden and input layer, the storage and computation cost imposed by this tensor is immense, so they instead factorize it and write that:

$$W_i = W_{hf} \cdot \text{diag}(W_{fx} x_i) \cdot W_{fh}$$

This has full expressive power if the dimension of f is large enough. However if we reduce the dimension, we get an approximation to this third rank tensor, which is good enough for these purposes. The backpropagation algorithm for this may be derived in a similar manner as before, or done automatically in symbolic manipulation or automatic differentiation software.

Conclusion

Recurrent neural networks are a powerful tool which allow neural networks to handle arbitrary length sequence data. Of course, they require that the sequence be a contextual sequence, in which the context is entirely generated by things in the preceeding portion of the sequence. Though this is fairly large simplifying assumption, it turns out that recurrent neural networks are still very powerful.

The backpropagation algorithms for them are no more complex than for previously discussed neural networks, and can be derived manually via painstaking application of the chain rule or automatically via automatic differentiation software.

Historically, recurrent neural networks have been very difficult to train, as the large number of layers imposes significant costs and makes first order algorithms impractical. However, it has been shown that when using Hessian-free optimization, recurrent neural networks can be trained well and trained fairly efficiently, and can thus be quite practical.

Although there is a common formulation, the exact formulation of the recurrent neural network may depend on the problem at hand and may require significant experimentation and tuning, as Sutskever et. al. demonstrated for text generation.

Friday, March 21, 2014 - Posted in machine-learning

« Gauss Newton Matrix Matrix Multiplication »

Contact

If you've got questions, comments, suggestions, or just want to talk, feel free to email me at andrew.gibiansky on Gmail.

Recent Posts