1 Backpropagation

http://arxiv.org/pdf/1411.6191.pdf

We'll go through backpropagation in too much detail.

1. It's 30 years old.
2. It's the workhorse of deep learning.
3. It's a large contributor to state-of-the-art.
4. It's dead simple: gradient descent (computing derivatives) plus the chain rule.
5. But the brain doesn't use it. Neuroscientists have tried for 50 years to find evidence of backprop in the brain and are pretty convinced it's not there.

Backprop is more complicated than the brain; it has feedforward and feedbackward output. The brain seems to be working better. So we take another look and ask, can we simplify backprop?

1.1 Learning algorithm with linear outputs

Given a weight vector $\mathbf{w}$, an input $\mathbf{x}$, output $\langle \mathbf{w}, \mathbf{x} \rangle$. We are given a loss function, for example the **mean squared error**

$$\ell(\langle \mathbf{w}, \mathbf{x} \rangle, y) = \frac{1}{2}(y - \langle \mathbf{w}, \mathbf{x} \rangle)^2.$$

Minimizing it is doing linear regression. Other loss functions are logistic and linear loss. This is studied most in online learning (Elad Hazan). We'll use MSE because it's simplest.

We won't use a neural network to minimize linear loss, but the quantity naturally comes up.

We need a learning rule.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta \nabla_{\mathbf{w}} \ell_{\mathbf{w}}.$$

If loss is convex, this converges.

The only problem is that the class of functions is very restricted, so we would like to extend this to more complicated outputs than just dot products. The standard answer is to build a neural network.

Consider a neural network. The hidden nodes (rectlinear/rectifiers) have the following form

$$S_{\mathbf{w}^j}(\mathbf{x}) = \max\{0, \left\langle \mathbf{w}^j, \mathbf{x} \right\rangle\}.$$

This is the closest we can get to linear. If we didn't have nonlinear functions, we'd just get linear functions, and there would be no advantage to a network. Leshno showed that $\operatorname{span}\left\{S_w(x) : w \in \mathbb{R}^d\right\}$ is dense in $\mathbb{R}^d$. Assume the output layer is linear (but it doesn't matter). We have a loss $\{\text{output loss}\} \times \{\text{labels}\} \to \mathbb{R}$.

We have a function $F_{\mathbf{W}}(x)$ computed by the neural net. To update $\mathbf{w}$, set

$$\mathbf{w}_{t+1}^j = \mathbf{w}_t^j + \eta \nabla_{\mathbf{w}^j} \ell(F_{\mathbf{w}^j}(x), \text{label}).$$

Here $F_{\mathbf{w}}$ is not convex so we're not guaranteed to find a global minimum, but it's good enough.

We'll deconstruct backprop and find hidden degrees of freedom.

Applying the chain rule,

$$\nabla_{\mathbf{w}^j} \ell(F_{\mathbf{w}^j}(x), \text{label}) = \underbrace{\nabla_{\mathbf{x}^{\text{out}}} \ell(\mathbf{x}^{\text{out}}, \text{label})}_{1\times d} \cdot \underbrace{(\nabla_{\mathbf{x}^j} x^{\text{out}})}_{d\times 1} \cdot \underbrace{(\nabla_{\mathbf{w}^j} x^j)}_{n_j}$$

The scalar that's the product of the first 2 vectors is written δ^j. Our goal is to simplify this.

We compute $\nabla_{x^j} \mathbf{x}^{\text{out}} = ?$. Compute derivative of $S_{\mathbf{w}^j}(\mathbf{x}) = \max(0, \langle \mathbf{w}, \mathbf{x} \rangle)$,

$$\frac{\partial S_{\mathbf{w}}(x)}{\partial x_L} = \begin{cases} w_i, & \langle \mathbf{w}, \mathbf{x} \rangle > 0 \\ 0, & \text{else.} \end{cases} = w_i \mathbb{1}_j$$

where $\mathbb{1}_j$ is 1 if node j is active and 0 else.

Hence $\nabla_{\mathbf{x}} S_{\mathbf{w}^j}(\mathbf{x}) = \mathbf{w}\mathbb{1}_j$. We get

$$\nabla_{x^j} \mathbf{x}^{\text{out}} = \sum_{\{k: j\to k\}} w^{jk} \mathbb{1}_k \left(\sum_{\{l: k\to l\}} w^{kl} \mathbb{1}_l \left(\sum \cdots \right) \right) = \sum_{\substack{\text{active paths} \\ \text{from } j \text{ to } x^{\text{out}}}} (\text{weights on path}).$$

Go through every layer, add the weights if they acted.

Define $\pi^j = \nabla_{x^j} \mathbf{x}^{\text{out}} = \sum_{\text{active paths}} (\text{weights})$, the influence.

We don't care about $\nabla \ell$; this is the business end (interaction of neural network with data). We care about the structure of the neural network, $\nabla_{x^j} \mathbf{x}^{\text{out}}$, how the neural network communicates with itself. This is difficult to do in hardware. Why? You have a system with 2 regimes: input flows forwards, then backpropagate error. Feedforward phase and feedback phase, with a global clock. You need massive synchrony. It's not realistic to build an organism which does this.

It's hard to simplify derivatives and chain rule. We need a new angle. Instead of thinking about it as a giant organism, we'll see each node has its own loss and is doing a learning algorithm.

1.2 Each node is a learning algorithm

We have

$$\nabla_{\mathbf{w}^j} x^j = \mathbf{x}\mathbb{1}_j$$

(recall $x^j = S_{w^j}(\mathbf{x}) = \max(0, \langle \mathbf{w}^j, \mathbf{x} \rangle)$.

Let's take a step back.

Consider backpropagtion for 1 node. It does the following.

1. Receive input.

2. Output.
3. Receive backpropagated error.
4. Update weights.

Step	Quantity
Receive input	ϕ^j
Output	$\max\{0, \langle \mathbf{w}^j, \phi^j \rangle\}$
backpropagated error	δ^j
update weights	$\mathbf{w}_{t+1}^j = \mathbf{w}_t^j + \eta^j \delta^j \phi^j \mathbb{1}^j$

Define the rectilinear loss

$$\ell_{\mathrm{RL}}(\mathbf{w}, \phi, \delta) = \begin{cases} \delta \langle \mathbf{w}, \phi \rangle, & \text{if node is active} \\ 0, & \text{else.} \end{cases}$$

This is not convex but it is selectively convex. Every node either does nothing $\langle \mathbf{w}^j, \phi^j \rangle < 0$ or minimizes a linear loss using gradient descent.

This lets you prove error and generalization loss, even in adversarial settings. We get nice generalization bounds on the nodes.

How can we use this to simplify backprop? δ is computed by backprop. We can imagine replacing δ by another number. Let's mess with δ; replace it with something easier to compute. (Obviously you have to be careful, as δ was chosen to minimize global loss.)

Consider the special case where the output layer has 1 node and

$$\nabla_{w^j} \ell(NN, \text{label}) = \nabla_{x^{\text{out}}} (\sum (\text{weights on active paths}))(\nabla_{\mathbf{w}^j} x^j).$$

The first 2 quantities are scalars; multiplying gives δ^j.

We'll do something naive which works.

$\pi^j = \sum_{\text{all active paths}}$, $\tau^j = \sum_{\text{all active paths of length 1}} (\text{weight on path}) = \sum_{\{k:j\to k\}} w^{jk} \mathbb{1}_k$. As the network gets bigger, τ^j gets easier and easier relative to π^j.

We define a new algorithm called kickback.

$$\begin{aligned} \text{BP} &= \ell_{\mathrm{RL}}(\mathbf{w}^j, \phi^j, \delta) \\ \text{KB} &= \ell_{\mathrm{RL}}(\mathbf{w}^j, \phi^j, (\nabla_{\mathbf{x}^{\text{out}}} (\text{Poss}))\tau^j). \end{aligned}$$

The analogue in biology is neural modulators. They are diffuse signals received by a large part of the brain.

We have a global scalar, $\nabla_{x^{\text{out}}} (\text{loss})$. Node j just looks at adjacent nodes, and asks which one was active after I was active; sum up the weights to compute τ. τ^j is easy to compute because it's completely local information. Plus we get global information: whether we overshot or undershot. This is biologically plausible.

Kickback is no longer gradient descent, so we have no guarantee of error reduction. There is an easy way to fix this.

Definition 1.1: A node is **coherent** if $\tau^j > 0$.

A network is **coherent** if $\tau_j > 0$ for all j.

We need a condition for consistency: the effect of a neuron is always positive or always negative. All the positive rectifiers have positive weights and all the negative rectifiers have negative weights. This is a property of the network. The price you pay for throwing away backprop is constraint on the network.

Take a NN, initialize weights randomly, change the weights to be consistent, and forgot about it, without worrying about coherency. It worked.

Conclusion: Coherent initialization and kickback give results almost identical to backprop. RPROP was invented in 90's to speed up optimization; RMS-PROP is a more complicated version with many batches. Compute δ^j, and throw away everything except for the sign. It's a hack that works surprisingly well. About half of deep learning papers rely on RMS-PROP.

Coherency: Starting it at 1, it typically went down to .9; backprop didn't seem to care (it moved everywhere). Backprop both doesn't need and doesn't preserve coherence. Kickback needs and seems to preserve coherence. When it's not initialized for coherence, often it doesn't work (blows up), or does weird oscillations.

Everything about kickback is simpler than the brain except coherency. Excitatory neurons behave as they should, but inhibatory neurons don't work the way they should; if we can fix this then we have a plausible biological algorithm.

This only works with binary classification. Multiple classification is hard. "I'm interested in reinforcement learning, excited about TD-error, a scalar."

Mixture version of gradient descent. Each part on one computer? Kickback is more parallelizable.

It's not faster (only marginally faster) in software; it's faster in hardware.