Backpropagation Algorithm

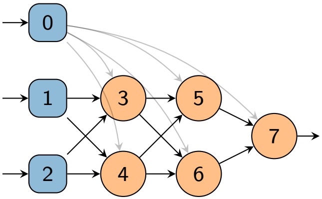
The Backpropagation Algorithm

---

procedure $\textsc{Backpropagate}((\mathbf{x}, y);\ \mathcal{N} = (V, A);\ \mathbf{w})$
```
/* Forward Propagation */
```
for each neuron j in input layer **do**
    $a_j \leftarrow x_j$
for each layer ℓ from 2 to L **do**
    for each neuron j in layer ℓ **do**
        $in_j \leftarrow \sum_{i:(i,j)\in A} w_{ij} a_i$
        $a_j \leftarrow g(in_j)$
```
/* Backpropagation */
```
for each neuron j in the output layer **do**
    $\Delta_j \leftarrow g'(in_j) \cdot (-2(y - a_j))$ ▷ Assumes squared error is the loss function
for each layer ℓ from $L-1$ down to 2 **do**
    for each neuron j in layer ℓ **do**
        $\Delta_j \leftarrow g'(in_j) \sum_{j':(j,j')\in A} w_{j,j'} \Delta_{j'}$

- Forward propagation computes the in_j and a_j values for every neuron
- Backward propagation computes the Δ_j values for every neuron

Some Calculations



$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \Delta_j a_i$$

$$\Delta_j = \begin{cases} -2(y - a_j) g'(in_j) & \text{if } j \text{ in output layer} \\ g'(in_j) \displaystyle\sum_{j' : (j, j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ in hidden layer} \end{cases}$$

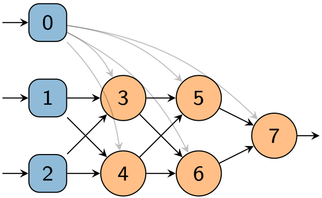
Computing some partial derivatives:

$$\begin{aligned} \frac{\partial}{\partial w_{5,7}} J(\mathbf{w}) &= \Delta_7 a_5 \\ &= -2(y - a_7) g'(in_7)\ a_5 \\ \frac{\partial}{\partial w_{3,5}} J(\mathbf{w}) &= \Delta_5 a_3 \\ &= \Delta_7\ w_{5,7} g'(in_5)\ a_3 \\ &= -2(y - a_7) g'(in_7)\ w_{5,7} g'(in_5)\ a_3 \end{aligned}$$

Computing some Δ_j's:

$$\begin{aligned} \Delta_7 &= -2(y - a_7) g'(in_7) \\ \Delta_5 &= g'(in_5) \sum_{j' : (5, j') \in A} \Delta_{j'} w_{5,j'} \\ &= \Delta_7\ w_{5,7} g'(in_5) \\ \Delta_6 &= g'(in_6) \sum_{j' : (6, j') \in A} \Delta_{j'} w_{6,j'} \\ &= \Delta_7\ w_{6,7} g'(in_6) \end{aligned}$$

Some More Calculations



$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \Delta_j a_i$$

$$\Delta_j = \begin{cases} -2(y - a_j) g'(in_j) & \text{if } j \text{ in output layer} \\ g'(in_j) \displaystyle\sum_{j' : (j,j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ in hidden layer} \end{cases}$$

Computing some partial derivatives:

$$\frac{\partial}{\partial w_{1,3}} J(\mathbf{w}) = \Delta_3 a_1$$

Computing some Δ_j's:

$$\Delta_7 = -2(y - a_7) g'(in_7)$$

$$\Delta_5 = \Delta_7 \, w_{5,7} g'(in_5)$$

$$\Delta_6 = \Delta_7 \, w_{6,7} g'(in_6)$$

$$\begin{aligned} \Delta_3 &= g'(in_3) \sum_{j' : (3,j') \in A} w_{3,j'} \Delta_{j'} \\ &= g'(in_3) \left(w_{3,5} \Delta_5 + w_{3,6} \Delta_6 \right) \\ &= g'(in_3) \big(w_{3,5} \Delta_7 w_{5,7} g'(in_5) \\ &\qquad + w_{3,6} \Delta_7 w_{6,7} g'(in_6) \big) \end{aligned}$$

Quick Recap: Partial Derivatives and Backpropagation

For a cost function $J(\mathbf{w})$ involving squared error (loss) on a **single** training example, we have:

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \left[\frac{\partial}{\partial in_j} J(\mathbf{w})\right] \left[\frac{\partial}{\partial w_{ij}} in_j\right] = \Delta_j a_i$$

for any weight w_{ij}, where

$$\Delta_j = \begin{cases} g'(in_j)(-2(y - a_j)) & \text{if } j \text{ belongs to output layer} \\ g'(in_j) \displaystyle\sum_{j' : (j,j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ belongs to hidden layer} \end{cases}$$

Forward propagation calculates the in_j and a_j values; **Backward propagation** provides an efficient way to calculate the Δ_j values.

Let's see how we can use this to **train** a neural network!

Neural Network Training

Training a Neural Network with Stochastic Gradient Descent

procedure $\textsc{NeuralNetTrain}(\{(\mathbf{x}_i, y_i)\}_{i=1}^n; \mathcal{N} = (V, A)$; learning rate $\alpha)$
    for each edge $(i, j) \in A$ **do**
        $w_{ij}^{(0)} \leftarrow \textsc{rand}(-\epsilon, \epsilon)$ ▷ Initialize each weight to a small random value
    $e \leftarrow 0$, $t \leftarrow 0$ ▷ Initialize epoch and iteration counters
    while stopping conditions not met **do**
        Create random permutation L of $\{1, 2, \ldots, n\}$
        for each i in L **do**
            Run $\textsc{Backpropagate}((\mathbf{x}_i, y_i), \mathcal{N}, \mathbf{w}^{(t)})$ to obtain all in_j, a_j, Δ_j values
            for each edge $(i, j) \in A$ **do**
                $w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \Delta_j a_i$ ▷ Update each edge weight
            $t \leftarrow t + 1$
        $e \leftarrow e + 1$ ▷ an epoch ends once we process all training examples once

- Using the same value for all weights is problematic, so randomize!
- Typical stopping conditions are reaching an epoch limit, getting errors close to zero for all training examples, and/or having minimal change in cost across an epoch

Mini-Batch Gradient Descent

Recall that **mini-batch gradient descent** offers a compromise between computing the gradient of $J(\mathbf{w})$ exactly versus estimating it using a single training point.

To use mini-batch gradient descent, we need to revisit our partial derivative calculations.

In SGD, we use a single training example $(\mathbf{x}, y)$ with:

$$J(\mathbf{w}) \approx \ell\left(y, h_{\mathbf{w}}(\mathbf{x})\right)$$

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \frac{\partial}{\partial w_{ij}} \ell\left(y, h_{\mathbf{w}}(\mathbf{x})\right) = \Delta_j a_i$$

where Δ_j and a_i are computed by running backpropagation on training example $(\mathbf{x}, y)$.

In mini-batch GD, we use a subset of training examples with indices in a set B:

$$J(\mathbf{w}) \approx \frac{1}{|B|} \sum_{n \in B} \ell\left(y_n, h_{\mathbf{w}}(\mathbf{x}_n)\right)$$

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} \frac{1}{|B|} \sum_{e \in B} \ell\left(y_e, h_{\mathbf{w}}(\mathbf{x}_e)\right) \\ &= \frac{1}{|B|} \sum_{e \in B} \frac{\partial}{\partial w_{ij}} \ell\left(y_e, h_{\mathbf{w}}(\mathbf{x}_e)\right) \\ &= \frac{1}{|B|} \sum_{e \in B} \left(\Delta_j^{(e)} a_i^{(e)}\right)\end{aligned}$$

where $\Delta_j^{(e)}$ and $a_i^{(e)}$ are the Δ_j and a_i values computed during backpropagation using training example $(\mathbf{x}_e, y_e)$, for each example index $e \in B$. (So we have to run backpropagation $|B|$ times!)

Corrections for Overfitting

Some remedies to overfitting:

- Get more training data!
  - Adjusting parameters to "memorize the noise" on a single training example is likely to increase error on other training examples, so the learning algorithm won't do this!
  - This is why "big data" has helped renew interest in neural networks.

- Add **regularization** to the cost function:

$$J(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(y_i, h_{\mathbf{w}}(\mathbf{x}_i)\right) + \lambda \sum_{(i,j) \in A} w_{ij}^2$$

Recall: λ is a **model hyperparameter** that needs to be picked ahead of time (we can use **cross-validation** to help pick λ).

Regularization in Neural Networks: Weight Decay

In stochastic gradient descent with training example $(\mathbf{x}, y)$, the partial derivatives of the regularized cost function are given by:

$$\begin{aligned}
\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} \left[\ell\left(y, h_{\mathbf{w}}(\mathbf{x})\right) + \lambda \sum_{(i',j')\in A} w_{i'j'}^2 \right] \\
&= \left[\frac{\partial}{\partial w_{ij}} \ell\left(y, h_{\mathbf{w}}(\mathbf{x})\right) \right] + \left[\lambda \sum_{(i',j')\in A} \frac{\partial}{\partial w_{ij}} w_{i'j'}^2 \right] \\
&= \Delta_j a_i + 2\lambda w_{ij}.
\end{aligned}$$

Then the weight update is:

$$w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \Delta_j a_i - 2\alpha\lambda w_{ij}.$$

This is called **weight decay**, because the weights tend to move **towards zero** unless pushed back by $\Delta_j a_i$.