Neural Networks: Back-Propagation algorithm

Andrzej Kordecki

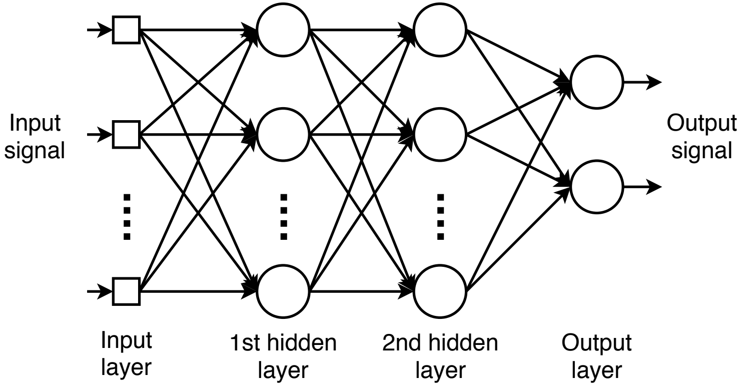
Neural Networks (ML.ANK385 and ML.EM05): Lecture 06
Division of Theory of Machines and Robots
Institute of Aeronautics and Applied Mechanics
Faculty of Power and Aeronautical Engineering
Warsaw University of Technology

Table of Contents

Back-Propagation algorithm

Back-Propagation algorithm

The development of the back-propagation algorithm in the mid-1980s represented a landmark in the training of multilayer neural network.



Back-Propagation algorithm

The training proceeds in two phases:

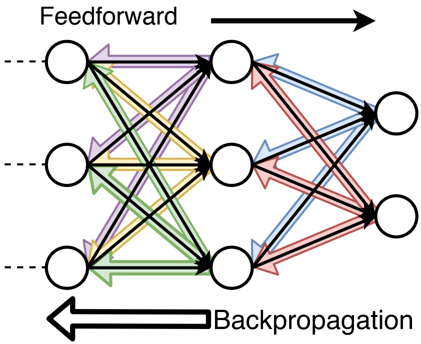
1. In the forward phase - synaptic weights of the network are fixed and the function signal is propagated through the network from input, layer by layer, until it reaches the output.
2. In the backward phase - error signal is produced by comparing the output of the network with a desired response. The resulting error signal is propagated through the network, again layer by layer, but this time the propagation is performed in the backward direction. In this second phase, successive adjustments are made to the synaptic weights of the network.

In the back-Propagation algorithm are identify two kinds of signals: function signals and error signal.

Back-Propagation algorithm

The training proceeds in two phases:

1. Forward pass - weights remain unaltered in direction:
   - input layer → hidden layers → output layer
2. Backward pass - error calculation for each neuron and weight update in direction:
   - input layer ← hidden layers ← output layer



Learning in Multi-Layer neural network

We want to adjust the network weights $w_{ij}^{(n)}$ in order to minimize the sum-squared loss/error function:

$$E(w) = \frac{1}{2}\sum_{p}\sum_{j}(t_j(p) - y_j^{(N)}(x(p), w))^2$$

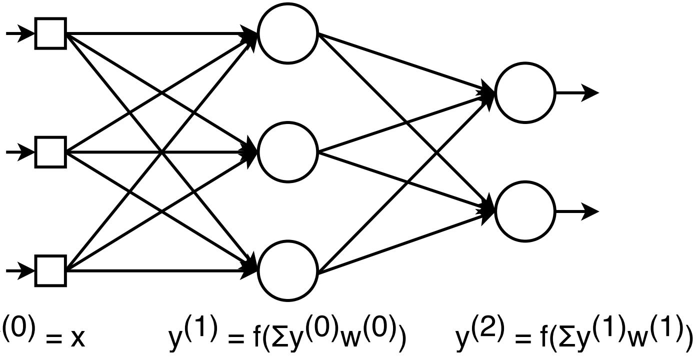
The gradient descent weight updates:

$$w_{h,g}^{new} = w_{h,g}^{old} - \eta \frac{\partial E}{\partial w_{h,g}}\Big|_{w_{h,g}=w_{h,g}^{old}}$$

- Only outputs $y_j^{(N)}$ of the final layer that appear in the loss function,
- The final layer outputs depend on all the layers of weight - we need to adjust them all.

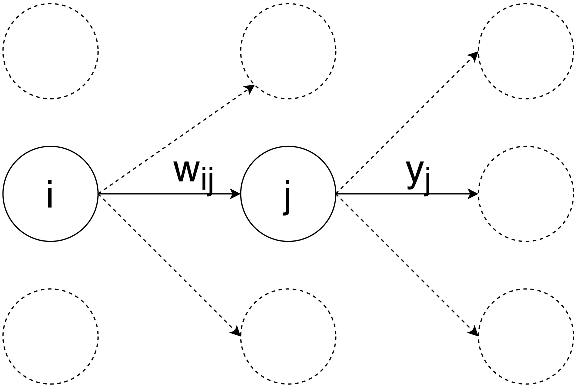
Learning in Multi-Layer neural network

Consider two layer neural network ($N = 2$):


$y^{(0)} = x$
$y^{(1)} = f(\Sigma y^{(0)} w^{(0)})$
$y^{(2)} = f(\Sigma y^{(1)} w^{(1)})$

Learning in Multi-Layer neural network

Multi-Layer Network connections and weights notation:



Learning in Multi-Layer neural network

The output equation two layer neural network:

$$y_k^{(2)} = f(\sum_j y_j^{(1)} w_{j,k}^{(1)}) = f(\sum_j f(\sum_i x_i w_{i,j}^{(0)}) w_{j,k}^{(1)})$$

where $y_k^{(n)}$ - the output of k-neuron in n-layer, i and j - number of inputs and outputs for appropriate neuron.

To update the weights we need to give the derivatives with respect to the two sets of weights $w^{(1)}$ and $w^{(2)}$:

$$\frac{\partial E}{\partial w_{h,g}^{(m)}} = \sum_p \sum_j \frac{\partial}{\partial w_{h,g}^{(m)}} (t_j(p) - y_j^{(N)}(x(p), w))^2$$

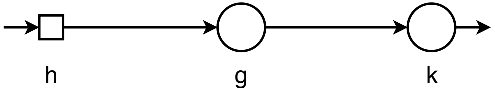
Learning in Multi-Layer neural network

The weight update is base on derivatives of 1st and 2nd layer:

$$\frac{\partial E}{\partial w_{h,g}^{(m)}} = -\sum_p \sum_j (t_j - y_j^{(2)}) \frac{\partial y_j^{(2)}}{\partial w_{h,g}^{(m)}}$$

$$\Downarrow$$

Consider only one signal path:



$$\frac{\partial y_k^{(2)}}{\partial w_{g,k}^{(1)}} = \frac{\partial}{\partial w_{g,k}^{(1)}} f(\sum_j f(\sum_i x_i w_{i,j}^{(0)}) w_{j,k}^{(1)}) = f'(\sum_j y_j^{(1)} w_{j,k}^{(1)}) y_g^{(1)}$$

$$\frac{\partial y_k^{(2)}}{\partial w_{h,g}^{(0)}} = f'(\sum_j y_j^{(1)} w_{j,k}^{(1)}) f'(\sum_i x_i w_{i,g}^{(0)}) w_{g,k}^{(1)} x_h$$

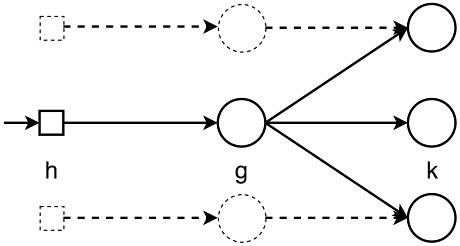
Learning in Multi-Layer neural network

The weight update is base on derivatives of 1st and 2nd layer:

$$\frac{\partial E}{\partial w_{h,g}^{(m)}} = -\sum_p \sum_j (t_j - y_j^{(2)}) \frac{\partial y_j^{(2)}}{\partial w_{h,g}^{(m)}}$$

$$\Downarrow$$

Error signal propagate from all the outputs of neural network:



Learning in Multi-Layer neural network

The weight update is base on derivatives of 1st and 2nd layer:

$$\frac{\partial E}{\partial w_{h,g}^{(m)}} = -\sum_p \sum_j (t_j - y_j^{(2)}) \frac{\partial y_j^{(2)}}{\partial w_{h,g}^{(m)}}$$

$$\Downarrow$$

$$\frac{\partial y_k^{(2)}}{\partial w_{g,k}^{(1)}} = \frac{\partial}{\partial w_{g,k}^{(1)}} f(\sum_j f(\sum_i x_i w_{i,j}^{(0)}) w_{j,k}^{(1)}) = f'(\sum_j y_j^{(1)} w_{j,k}^{(1)}) y_g^{(1)}$$

$$\frac{\partial y_k^{(2)}}{\partial w_{h,g}^{(0)}} = f'(\sum_j y_j^{(1)} w_{j,k}^{(1)}) f'(\sum_i x_i w_{i,g}^{(0)}) w_{g,k}^{(1)} x_h$$

$$\Downarrow$$

$$\frac{\partial E}{\partial w_{h,g}^{(0)}} = -\sum_p \sum_k (t_k - y_k^{(2)}) f'(\sum_j y_j^{(1)} w_{j,k}^{(1)}) f'(\sum_i x_i w_{i,g}^{(0)}) w_{g,k}^{(1)} x_h$$

Learning in Multi-Layer neural network

Substitute derivatives into the weight update equations:

$$\begin{aligned}\Delta w_{h,g}^{(1)} &= \eta \textstyle\sum_p (t_g - y_g^{(2)}) f'(\sum_j y_j^{(1)} w_{j,g}^{(1)}) y_h^{(1)} \\ \Delta w_{h,g}^{(0)} &= \eta \textstyle\sum_p \sum_k (t_k - y_k^{(2)}) f'(\sum_j y_j^{(1)} w_{j,k}^{(1)}) f'(\sum_i x_i w_{i,g}^{(0)}) w_{g,k}^{(1)} x_h\end{aligned}$$

It is convenient to define:

$$\delta_k^{(2)} = (t_k - y_k^{(2)}) f'(\sum_j y_j^{(1)} w_{j,k}^{(1)})$$

We can then write the weight update rules as:

$$\begin{aligned}\Delta w_{h,g}^{(1)} &= \eta \textstyle\sum_p \delta_g^{(2)} y_h^{(1)} \\ \Delta w_{h,g}^{(0)} &= \eta \textstyle\sum_p (\sum_k \delta_k^{(2)} w_{g,k}^{(1)}) f'(\sum_i x_i w_{i,g}^{(0)}) x_h\end{aligned}$$

The weight changes at the first layer now take on the same form as the final layer, but the 'error' at each unit g is back-propagated from each of the output units k.

Back-Propagation algorithm

It is easy to extend our gradient descent algorithm to work for any number of hidden layers:

- for output layer:

$$\delta_k^{(N)} = (t_k - y_k^{(N)}) f'(\textstyle\sum_j y_j^{(N-1)} w_{j,k}^{(N-1)})$$

- for hidden layers:

$$\delta_k^{(n)} = (\textstyle\sum_k \delta_k^{(n+1)} w_{g,k}^{(n)}) f'(\textstyle\sum_j y_j^{(n-1)} w_{j,k}^{(n-1)})$$

Weight update equation can be written as:

$$\Delta w_{h,g}^{(n)} = \eta \sum_p \delta_g^{(n+1)} y_h^{(n)}$$

Back-Propagation algorithm

In case of Sigmoid activation function:

- for output layer:

$$\delta_k^{(N)} = (t_k - y_k^{(N)})y_k^{(N)}(1 - y_k^{(N)})$$

- for earlier layers - the back-propagation formula for the local gradient:

$$\delta_k^{(n)} = (\textstyle\sum_k \delta_k^{(n+1)} w_{g,k}^{(n)})y_k^{(n)}(1 - y_k^{(n)})$$

Weight update equation can be written as:

$$\Delta w_{h,g}^{(n)} = \eta \sum_p \delta_g^{(n+1)} y_h^{(n)}$$

Cross-entropy loss function

How learning will change if we use multi-class cross-entropy loss function:

$$E(t, y) = -\sum t_p \log y(w)$$

Where t_p is 1 at the index of the true label of p-sample.
Logarithm derivative:

$$\frac{d}{dx} \log f(x) = \frac{1}{f(x)} \frac{d}{dx} f(x)$$

Cross-Entropy derivative:

$$\frac{d}{dw}(-\sum t_p \log y(w)) = -\sum \frac{t_p}{y(w)} \frac{d}{dw} y(w)$$

Cross-entropy loss function

The difference is in calculation of the last layer of network error.

- In case of MSE:

$$\frac{\partial E}{\partial w_{h,g}^{(m)}} = -\sum_p \sum_j (t_j - y_j^{(2)}) \frac{\partial y_j^{(2)}}{\partial w_{h,g}^{(m)}}$$

- In case of cross-entropy:

$$\frac{\partial E}{\partial w_{h,g}^{(m)}} = -\sum_p \sum_j \frac{t_j}{y_j^{(2)}} \frac{\partial y_j^{(2)}}{\partial w_{h,g}^{(m)}}$$

Cross-entropy in Back-Propagation algorithm

Therefore the loop will be as follow:

- for output layer:

$$\delta_k^{(N)} = \frac{t_k}{y_k^{(N)}} f'(\textstyle\sum_j y_j^{(N-1)} w_{j,k}^{(N-1)})$$

- for hidden layers:

$$\delta_k^{(n)} = (\textstyle\sum_k \delta_k^{(n+1)} w_{g,k}^{(n)}) f'(\sum_j y_j^{(n-1)} w_{j,k}^{(n-1)})$$

Weight update equation can be written as:

$$\Delta w_{h,g}^{(n)} = \eta \sum_p \delta_g^{(n+1)} y_h^{(n)}$$

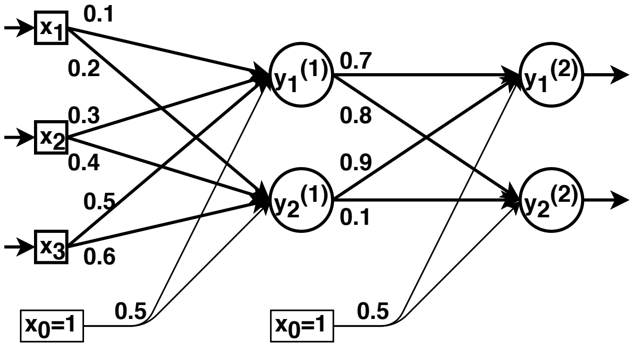
Back-Propagation algorithm

The algorithm:

1. Initialization: prepare training set with (x_i, y_i) sample, set up network structure, choose activation function, generate initial weights.
2. Select error function $E(w_{i,j})$ and learning rate η,
3. Apply the weight update equation $\Delta w_{h,g}^{(n)}$ to each weight $w_{i,j}$ for each training pattern p:
    a. Forward Computation. Compute the error signal $(t_k - y_k^{(N)})$ for initial or previous iteration weights,
    b. Backward Computation. Propagate error backward to each neuron of NN and update weights values.
4. Iteration. Iterate the forward and backward computations (point 3) by presenting new epochs of training examples to the NN until the chosen stopping criterion is met.

Example

Example: Update network weight with use back-propagation algorithm.



The input and target values for this problem are: $x_1 = 1$, $x_2 = 4$, $x_3 = 5$ and $t_1 = 0.1$, $t_2 = 0.05$; learning rate $\eta = 0.01$.

Example

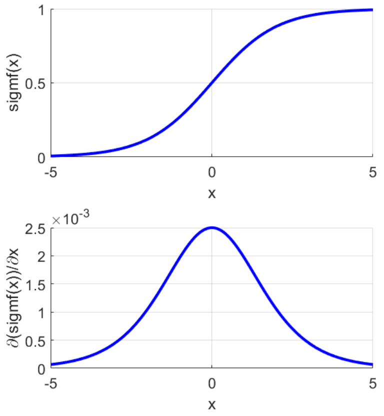
All neurons use sigmoid activation function:

- Output:

$$f(x) = \frac{1}{1+e^{-x}}$$

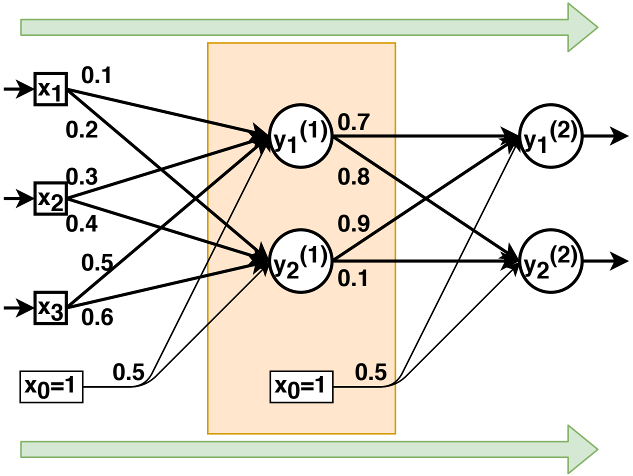
- Derivative:

$$\frac{\partial f(x)}{dx} = f(x)(1-f(x))$$



Example

Forward pass - 1st layer.



Example

Forward pass:

- 1st layer - functions:

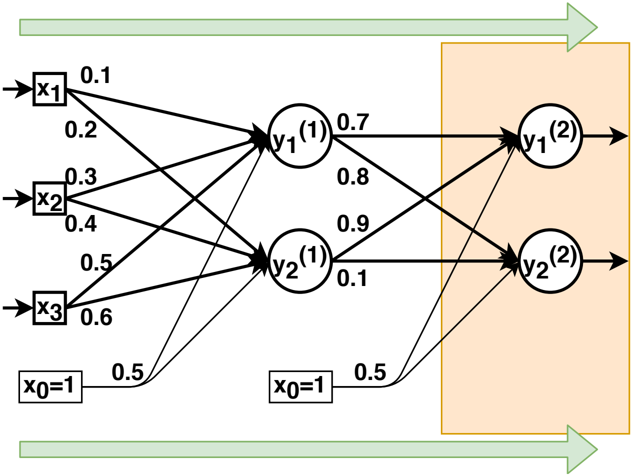
$$v_1^{(1)} = w_0^{(0)} x_0 + w_{1,1}^{(0)} x_1 + w_{2,1}^{(0)} x_2 + w_{3,1}^{(0)} x_3$$
$$y_1^{(1)} = f(v_1^{(1)})$$
$$v_2^{(1)} = w_0^{(0)} x_0 + w_{1,2}^{(0)} x_1 + w_{2,2}^{(0)} x_2 + w_{3,2}^{(0)} x_3$$
$$y_2^{(1)} = f(v_2^{(1)})$$

- 1st layer - values:

$$v_1^{(1)} = 0.5 * 1 + 0.1 * 1 + 0.3 * 4 + 0.5 * 5 = 4.3$$
$$y_1^{(1)} = f(4.3) = 0.9866$$
$$v_2^{(1)} = 0.5 * 1 + 0.2 * 1 + 0.4 * 4 + 0.6 * 5 = 5.3$$
$$y_2^{(1)} = f(5.3) = 0.9950$$

Example

Forward pass - 2nd layer.



Example

Forward pass:

- 2nd layer - functions:

$$\begin{aligned} v_1^{(2)} &= w_0^{(1)} x_0 + w_{1,1}^{(1)} y_1^{(1)} + w_{2,1}^{(1)} y_2^{(1)} \\ y_1^{(2)} &= f(v_1^{(2)}) \\ v_2^{(2)} &= w_0^{(1)} x_0 + w_{1,2}^{(1)} y_1^{(1)} + w_{2,2}^{(1)} y_2^{(1)} \\ y_2^{(2)} &= f(v_2^{(2)}) \end{aligned}$$

- 2nd layer - values:

$$\begin{aligned} v_1^{(2)} &= 0.5 * 1 + 0.7 * 0.9866 + 0.9 * 0.9950 = 2.0862 \\ y_1^{(2)} &= f(2.0862) = 0.8896 \\ v_2^{(2)} &= 0.5 * 1 + 0.8 * 0.9866 + 0.1 * 0.9950 = 1.3888 \\ y_2^{(2)} &= f(1.3888) = 0.8004 \end{aligned}$$

Example

Forward pass:

- Loss function (N=2):

$$E = \tfrac{1}{2}[(t_1 - y_1^{(2)})^2 + (t_2 - y_2^{(2)})^2]$$
$$E = \tfrac{1}{2}[(0.1 - 0.8896)^2 + (0.05 - 0.8004)^2] = 0.5933$$

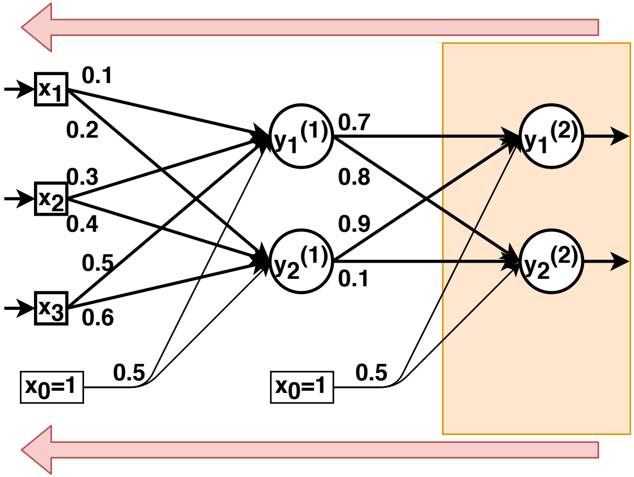
- Difference value for each neuron:

$$e_1 = t_1 - y_1^{(2)} = 0.1 - 0.8896 = -0.7896$$
$$e_2 = t_2 - y_2^{(2)} = 0.05 - 0.8004 = -0.7504$$

This is the end of forward pass.

Example

Backward pass - 2nd layer.



Example

Backward pass:

- 2nd layer (last layer) - $w_{1,1}^{(1)}$:

$$\frac{dE}{dw_{1,1}^{(1)}} = \frac{dE}{dy_1^{(2)}} \frac{dy_1^{(2)}}{dv_1^{(2)}} \frac{dv_1^{(2)}}{dw_{1,1}^{(1)}}$$

$$\frac{dE}{dw_{1,1}^{(1)}} = -(t_1 - y_1^{(2)})(y_1^{(2)}(1 - y_1^{(2)}))y_1^{(1)}$$

$$\frac{dE}{dw_{1,1}^{(1)}} = -(0.1 - 0.8896)(0.8896(1 - 0.8896))(0.9866)$$

$$\frac{dE}{dw_{1,1}^{(1)}} = 0.0765$$

Analogous to the formula:

$$\delta_k^{(N)} = (t_k - y_k^{(N)})f'(\sum_j y_j^{(N-1)} w_{j,k}^{(N-1)})$$

Example

Backward pass:

- 2nd layer (last layer) - $w_{1,2}^{(1)}$:

$$\frac{dE}{dw_{1,2}^{(1)}} = \frac{dE}{dy_2^{(2)}} \frac{dy_2^{(2)}}{dv_2^{(2)}} \frac{dv_2^{(2)}}{dw_{1,2}^{(1)}}$$

$$\frac{dE}{dw_{1,2}^{(1)}} = -(t_2 - y_2^{(2)})(y_2^{(2)}(1 - y_2^{(2)}))y_1^{(1)}$$

$$\frac{dE}{dw_{1,2}^{(1)}} = -(0.05 - 0.8004)(0.8004(1 - 0.8004))(0.9866)$$

$$\frac{dE}{dw_{1,2}^{(1)}} = 0.1183$$

Example

Backward pass:

- 2nd layer (last layer) - $w_{2,1}^{(1)}$:

$$\frac{dE}{dw_{2,1}^{(1)}} = \frac{dE}{dy_1^{(2)}} \frac{dy_1^{(2)}}{dv_1^{(2)}} \frac{dv_1^{(2)}}{dw_{2,1}^{(1)}} = 0.0772$$

- 2nd layer (last layer) - $w_{2,2}^{(1)}$:

$$\frac{dE}{dw_{2,2}^{(1)}} = \frac{dE}{dy_2^{(2)}} \frac{dy_2^{(2)}}{dv_2^{(2)}} \frac{dv_2^{(2)}}{dw_{2,2}^{(1)}} = 0.1193$$

Example

Backward pass:

- 2nd layer (last layer) - $w_0^{(1)}$ (bias):

$$\frac{dE}{dw_0^{(1)}} = \frac{dE}{dy_1^{(2)}}\frac{dy_1^{(2)}}{dv_1^{(2)}}\frac{dv_1^{(2)}}{dw_0^{(1)}} + \frac{dE}{dy_2^{(2)}}\frac{dy_2^{(2)}}{dv_2^{(2)}}\frac{dv_2^{(2)}}{dw_0^{(1)}}$$

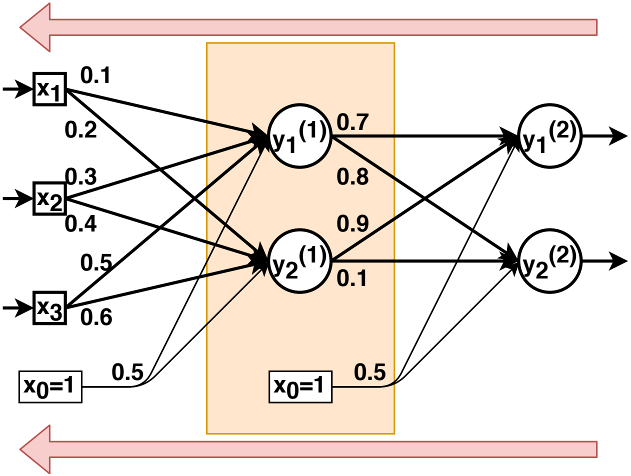
$$\frac{dE}{dw_0^{(1)}} = -(0.1 - 0.8896)(0.8896(1 - 0.8896))(1)$$
$$-(0.05 - 0.8004)(0.8004(1 - 0.8004))(1)$$
$$\frac{dE}{dw_0^{(1)}} = 0.1974$$

The error derivative of $w_0^{(1)}$ affect the error through both $y_1^{(2)}$ and $y_2^{(2)}$.

We will now backpropagate into deeper layer to compute the weight update of the parameters connecting the input layer to the hidden layer.

Example

Backward pass - 1st layer.



Example

Backward pass:

- 1st layer (last layer) - $w_{1,1}^{(0)}$:

$$\frac{dE}{dw_{1,1}^{(0)}} = \frac{dE}{dy_1^{(1)}} \frac{dy_1^{(1)}}{dv_1^{(1)}} \frac{dv_1^{(1)}}{dw_{1,1}^{(0)}}$$

$$\frac{dv_1^{(1)}}{dw_{1,1}^{(0)}} = x_1 = 1$$

$$\frac{dy_1^{(1)}}{dv_1^{(1)}} = (y_1^{(1)}(1 - y_1^{(1)})) = (0.9866)(1 - 0.9866) = 0.0132$$

The signal $y_1^{(1)}$ affects both $y_1^{(2)}$ and $y_2^{(2)}$:

$$\frac{dE}{dy_1^{(1)}} = \frac{dE}{dy_1^{(2)}} \frac{dy_1^{(2)}}{dv_1^{(2)}} \frac{dv_1^{(2)}}{dy_1^{(1)}} + \frac{dE}{dy_2^{(2)}} \frac{dy_2^{(2)}}{dv_2^{(2)}} \frac{dv_2^{(2)}}{dy_1^{(1)}}$$

Example

Backward pass:

- 1st layer (last layer) - $w_{1,1}^{(0)}$:

$$\frac{dE}{dy_1^{(1)}} = \frac{dE}{dy_1^{(2)}}\frac{dy_1^{(2)}}{dv_1^{(2)}}\frac{dv_1^{(2)}}{dy_1^{(1)}} + \frac{dE}{dy_2^{(2)}}\frac{dy_2^{(2)}}{dv_2^{(2)}}\frac{dv_2^{(2)}}{dy_1^{(1)}}$$

$$\frac{dE}{dy_1^{(1)}} = -(t_1 - y_1^{(2)})[(y_1^{(2)}(1 - y_1^{(2)}))]w_{1,1}^{(1)}$$

$$-(t_2 - y_2^{(2)})[(y_2^{(2)}(1 - y_2^{(2)}))]w_{1,2}^{(1)}$$

$$\frac{dE}{dy_1^{(1)}} = -(0.1 - 0.8896)(0.8896(1 - 0.8896))(0.7)$$

$$-(0.05 - 0.8004)(0.8004(1 - 0.8004))(0.8)$$

$$\frac{dE}{dw_{1,1}^{(0)}} = 0.1502$$

Analogous to the formula:

$$\delta_k^{(n)} = (\textstyle\sum_k \delta_k^{(n+1)} w_{g,k}^{(n)}) f'(\textstyle\sum_j y_j^{(n-1)} w_{j,k}^{(n-1)})$$

Example

Backward pass:

- 1st layer (last layer) - $w_{1,1}^{(0)}$:

$$\frac{dE}{dw_{1,1}^{(0)}} = \frac{dE}{dy_1^{(1)}} \frac{dy_1^{(1)}}{dv_1^{(1)}} \frac{dv_1^{(1)}}{dw_{1,1}^{(0)}} = (0.1502)(0.0132)(1) = 0.0020$$

Example

Backward pass:

- 1st layer (last layer) - $w_{1,2}^{(0)}$:

$$\frac{dE}{dw_{1,2}^{(0)}} = \frac{dE}{dy_2^{(1)}} \frac{dy_2^{(1)}}{dv_2^{(1)}} \frac{dv_2^{(1)}}{dw_{1,2}^{(0)}}$$

$$\frac{dv_2^{(1)}}{dw_{1,2}^{(0)}} = x_1 = 1$$

$$\frac{dy_2^{(1)}}{dv_2^{(1)}} = (y_2^{(1)}(1 - y_2^{(1)})) = 0.9950(1 - 0.9950) = 0.0050$$

The signal $y_2^{(1)}$ affects both $y_1^{(2)}$ and $y_2^{(2)}$:

$$\frac{dE}{dy_2^{(1)}} = \frac{dE}{dy_1^{(2)}} \frac{dy_1^{(2)}}{dv_1^{(2)}} \frac{dv_1^{(2)}}{dy_2^{(1)}} + \frac{dE}{dy_2^{(2)}} \frac{dy_2^{(2)}}{dv_2^{(2)}} \frac{dv_2^{(2)}}{dy_2^{(1)}}$$

Example

Backward pass:

- 1st layer (last layer) - $w_{1,2}^{(0)}$:

$$\frac{dE}{dy_2^{(1)}} = \frac{dE}{dy_1^{(2)}}\frac{dy_1^{(2)}}{dv_1^{(2)}}\frac{dv_1^{(2)}}{dy_2^{(1)}} + \frac{dE}{dy_2^{(2)}}\frac{dy_2^{(2)}}{dv_2^{(2)}}\frac{dv_2^{(2)}}{dy_2^{(1)}}$$

$$\frac{dE}{dy_2^{(1)}} = -(t_1 - y_1^{(2)})[(y_1^{(2)}(1 - y_1^{(2)}))]w_{2,1}^{(1)}$$

$$-(t_2 - y_2^{(2)})[(y_2^{(2)}(1 - y_2^{(2)}))]w_{2,2}^{(1)}$$

$$\frac{dE}{dy_2^{(1)}} = -(0.1 - 0.8896)(0.8896(1 - 0.8896))(0.9)$$

$$-(0.05 - 0.8004)(0.8004(1 - 0.8004))(0.1)$$

$$\frac{dE}{dw_{1,2}^{(0)}} = 0.0818$$

$$\frac{dE}{dw_{1,2}^{(0)}} = \frac{dE}{dy_2^{(1)}}\frac{dy_2^{(1)}}{dv_2^{(1)}}\frac{dv_2^{(1)}}{dw_{1,2}^{(0)}} = (0.0818)(0.0050)(1) = 0.0004$$

Example

Backward pass:

- 1st layer (hidden layer) - $w_{2,1}^{(0)}$:

$$\frac{dE}{dw_{2,1}^{(0)}} = \frac{dE}{dy_1^{(1)}} \frac{dy_1^{(1)}}{dv_1^{(1)}} \frac{dv_1^{(1)}}{dw_{2,1}^{(0)}} = 0.0079$$

- 1st layer (hidden layer) - $w_{2,2}^{(0)}$:

$$\frac{dE}{dw_{2,2}^{(0)}} = \frac{dE}{dy_2^{(1)}} \frac{dy_2^{(1)}}{dv_2^{(1)}} \frac{dv_2^{(1)}}{dw_{2,2}^{(0)}} = 0.0016$$

Example

Backward pass:

- 1st layer (hidden layer) - $w_{3,1}^{(0)}$:

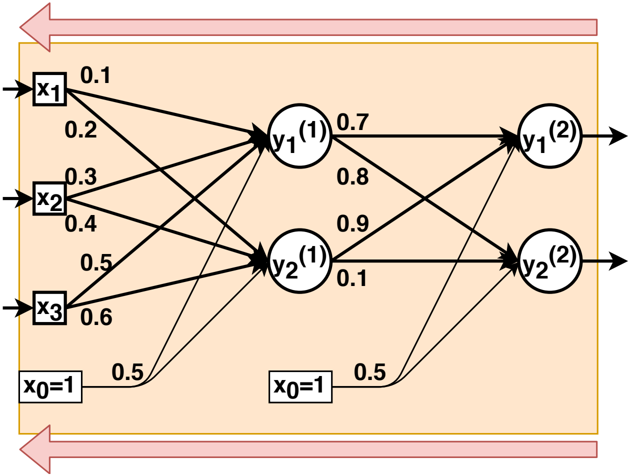
$$\frac{dE}{dw_{3,1}^{(0)}} = \frac{dE}{dy_1^{(1)}} \frac{dy_1^{(1)}}{dv_1^{(1)}} \frac{dv_1^{(1)}}{dw_{3,1}^{(0)}} = 0.0099$$

- 1st layer (hidden layer) - $w_{3,2}^{(0)}$:

$$\frac{dE}{dw_{3,2}^{(0)}} = \frac{dE}{dy_2^{(1)}} \frac{dy_2^{(1)}}{dv_2^{(1)}} \frac{dv_2^{(1)}}{dw_{3,2}^{(0)}} = 0.0020$$

Example

Weight update.



Example

Weight update:

$$w_{1,1}^{(0),new} = w_{1,1}^{(0),old} - \eta \frac{dE}{dw_{1,1}^{(0),old}} = 0.1 - (0.01)(0.0020) = 0.1000$$

$$w_{1,2}^{(0),new} = w_{1,2}^{(0),old} - \eta \frac{dE}{dw_{1,2}^{(0),old}} = 0.2 - (0.01)(0.0004) = 0.2000$$

$$w_{2,1}^{(0),new} = w_{2,1}^{(0),old} - \eta \frac{dE}{dw_{2,1}^{(0),old}} = 0.3 - (0.01)(0.0079) = 0.2999$$

$$w_{2,2}^{(0),new} = w_{2,2}^{(0),old} - \eta \frac{dE}{dw_{2,2}^{(0),old}} = 0.4 - (0.01)(0.0016) = 0.4000$$

$$w_{3,1}^{(0),new} = w_{3,1}^{(0),old} - \eta \frac{dE}{dw_{3,1}^{(0),old}} = 0.5 - (0.01)(0.0099) = 0.4999$$

$$w_{3,2}^{(0),new} = w_{3,2}^{(0),old} - \eta \frac{dE}{dw_{3,2}^{(0),old}} = 0.6 - (0.01)(0.0020) = 0.6000$$

$$w_{1,1}^{(1),new} = w_{1,1}^{(1),old} - \eta \frac{dE}{dw_{1,1}^{(1),old}} = 0.7 - (0.01)(0.0765) = 0.6992$$

$$w_{1,2}^{(1),new} = w_{1,2}^{(1),old} - \eta \frac{dE}{dw_{1,2}^{(1),old}} = 0.8 - (0.01)(0.1183) = 0.7988$$

Example

Weight update:

$$w_{2,1}^{(1),new} = w_{2,2}^{(1),old} - \eta \frac{dE}{dw_{2,1}^{(1),old}} = 0.9 - (0.01)(0.0772) = 0.8992$$
$$w_{2,2}^{(1),new} = w_{2,2}^{(1),old} - \eta \frac{dE}{dw_{2,2}^{(1),old}} = 0.1 - (0.01)(0.1193) = 0.9881$$
$$w_{0}^{(0),new} = w_{0}^{(0),old} - \eta \frac{dE}{dw_{0}^{(0),old}} = 0.5 - (0.01)(0.0008) = 0.5000$$
$$w_{0}^{(1),new} = w_{0}^{(1),old} - \eta \frac{dE}{dw_{0}^{(1),old}} = 0.5 - (0.01)(0.1975) = 0.4980$$

We repeat that over and over many times until the error converge to some small value.

Back-Propagation characteristics

Hidden layers

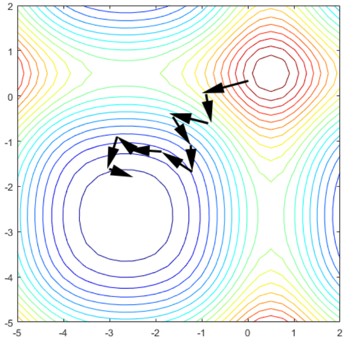
The hidden neurons act as feature detectors:

- perform a nonlinear transformation on the input data into a new space called the feature space.
- gradually "discover" the salient features that characterize the training data.

Number of hidden layers:

- under-fitting - too few hidden units will generally leave high training and generalization errors.
- over-fitting - too many hidden units will result in low training errors, but will make the training unnecessarily slow with poor generalization unless some technique, like regularization is used to prevent it.
- just right - low error and good generalization.

Hidden layers



We use more neurons near input layer:

- All neurons in the multilayer neural network should ideally learn at the same rate,
- The later network layers will tend to have larger local gradients than the earlier layers,
- The activations of units with many connections feeding into or out of them tend to change faster than units with fewer connections,

Hidden layers

The best number of hidden units depends in a complex way on many factors, including:

- numbers of input and output units,
- amount of the training data and amount of noise in them,
- complexity of the regression or classification,
- type of hidden unit activation function,
- training algorithm and similar problems.

Virtually all "rules of thumb" are not a good solution:

- MLP can approximate ANY function,
- best strategy is to try a range of numbers of hidden units and see which works best.

What is a Batch?

The batch size (or simply the batch) is a hyperparameter that defines the number of samples to work through before updating the neural network parameters. The training dataset can be divided into one or more batches:

- Batch (or deterministic) training: Batch Size = Size of Training Set,
- Stochastic (or online) training: Batch Size = 1,
- Mini-Batch training: 1 < Batch Size < Size of Training Set.

The term online is usually reserved for the case where the examples are drawn from a stream of continually created examples.

What is a Batch?

The size of the batch is a trade off between:

- reduction of calculation time,
- reduction of parameter updates variance (stability of results).

We prefer to use large batch size, but there are two main reasons to use smaller batch size:

- Smaller batch sizes are noisy, offering a regularizing effect and lower generalization error.
- Smaller batch sizes make it easier to fit one batch worth of training data in memory (i.e. when using a GPU).

What is a Batch?

Batch characteristics:

- A batch size of 32 means that 32 samples from the training dataset will be used to estimate the error gradient before the model weights are updated.
- Popular batch sizes include 32, 64, and 128 samples.

Mini-batch training advantages:

- reduces the variance of the parameter updates, which can lead to more stable convergence,
- make use of highly optimized matrix optimizations to increase speed of calculations.

Batch in Back-propagation algorithm

Batch learning accumulates error. What does it mean?
The error accumulation by average or sum are equivalent, in the sense that there exist pairs of learning rates for which they produce the same update. The average update for a batch of size N:

$$\overline{\Delta w_{h,g}} = \frac{1}{N}\sum_{i=1}^{N} -\eta \frac{\partial E_i}{\partial w_{h,g}}$$

But, this an equivalent of changing the value of learning rate:

$$\overline{\Delta w_{h,g}} = -\frac{\eta}{N}\sum_{i=1}^{N} \frac{\partial E_i}{\partial w_{h,g}} = \zeta \sum_{i=1}^{N} \frac{\partial E_i}{\partial w_{h,g}}$$

Linear activation function?

The network outputs are non-binary, then:

- it is appropriate to have a linear output activation function rather than a Sigmoid.
- So why not use linear activation functions on the hidden layers as well?

The outputs of two layer NN layer are given by:

$$y_k^{(2)} = f(\sum_j y_j^{(1)} w_{j,k}^{(1)}) = f(\sum_j f(\sum_i x_i w_{i,j}^{(0)}) w_{j,k}^{(1)})$$

If the hidden layer activation is linear $f(x) = x$:

$$y_k^{(2)} = f(\sum_j \sum_i x_i w_{i,j}^{(0)} w_{j,k}^{(1)}) = f(\sum_i x_i (\sum_j w_{i,j}^{(0)} w_{j,k}^{(1)}))$$

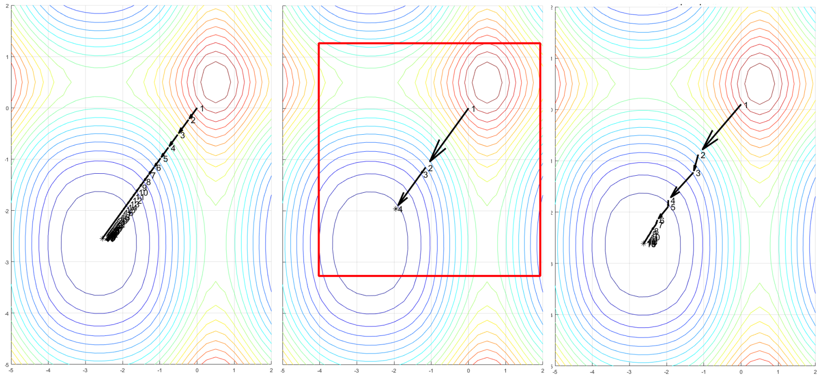
This is equivalent to a linear single layer network.

Improving Back-Propagation algorithm

Improved search of the extreme

Improving search of minimum:

- direction of optimization (previous iterations),
- learning rate (optimized 'jump' length along direction).



Learning with Momentum

General idea behind improving search of minimum:

- If the error is reduced greatly by the current update, the learning rate can be increased and optimization direction preserved.
- If the error is not reduced, the learning rate can be decreased and optimization direction changed.
- Gradient Descent would move quickly down the walls, but very slowly through the valley floor

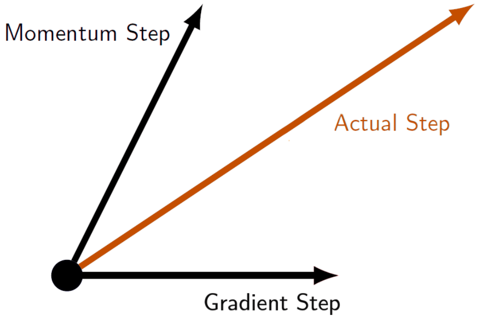
Learning with Momentum

In the momentum method increment of the weight can be modified by the updating history (values in previous iteration). If we label everything by the time t:

$$\Delta w_{h,g}^{(n)}(t) = \eta \sum_{p} \delta_{g}^{(n)} y_{h(t)}^{(n-1)} + \alpha \Delta w_{h,g}^{(n)}(t-1)$$

where α is usually a positive number called the momentum constant. We simply add a momentum term to weight update equation, which is the weight change of the previous step times - a momentum.

Learning with Momentum

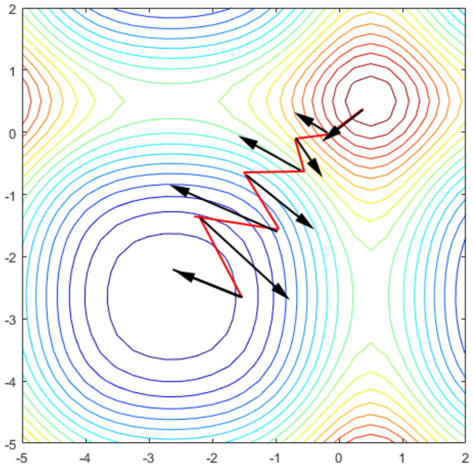


Learning with Momentum

- For the time t series to be convergent, the momentum constant must be restricted to the range 0-1. When is zero, the back-propagation operates without momentum.
- When the partial derivative has the same algebraic sign on consecutive iterations, the norm of $\Delta w(t)$ grows in magnitude. The inclusion of momentum in the back-propagation algorithm tends to accelerate descent in steady downhill directions.
- When the partial derivative has opposite signs on consecutive iterations, the exponentially weighted sum $\Delta w(t)$ shrinks in magnitude. The inclusion of momentum in the back-propagation algorithm has a stabilizing effect in directions that oscillate in sign.

Learning with Momentum

This allows you to reach the minimum even in difficult cases



Learning rate

The learning rate reflects how much we allow the change weights to follow the opposite direction of the error gradient, but:

- it can be very difficult to set a proper value (it should not be too small or too large),
- high-dimensional non-convex nature of neural networks optimization could lead to different sensitivity on each dimension (different rate of vale changes for each weight),

Solution is to choose different learning rate for each dimension.

Rprop

The idea behind changes of larning rate can explained with use of Rprop alghorithm. Rprop combines the idea of only using the sign of the gradient with the idea of adapting the step size individually for each weight.

- Check signs of the last two continues gradients of the weight. If they have the same sign, that means, we're going in the right direction, and should increase the step size multiplicatively (e.g by a factor of 1.2). If they're different, we should decrease the step size multiplicatively (e.g. by a factor of 0.5).
- Limit the step size between assumed range of values and apply the weight update.

M. Riedmiller, H. Braun, Rprop - A Fast Adaptive Learning Algorithm, Proc. of the International Symposium on Computer and Information Science VII, 1992

Rprop

The algorithm designed for full-batch optimization. Why Rprop doesn't work with mini-batches?

Answer: It is against the idea behind stochastic gradient descent, which is when we have small enough learning rate, it averages the gradients over successive mini-batches. Consider the weight, that gets the gradient 0.1 on 9 mini-batches, and the gradient of -0.9 on 10th mini-batch. What we'd like is to those gradients to roughly cancel each other out.

Adagrad

The one of the earlier algorithms that have been used to mitigate this problem for deep neural networks is the AdaGrad algorithm:

- algorithm adaptively scaled the learning rate for each dimension,
- smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features,
- larger updates (i.e. high learning rates) for parameters associated with infrequent occurring features.

J. Duchi, E. Hazan, Y.Singer, Adaptive subgradient methods for online learning and stochastic optimization, Journal of Machine Learning Research, 2011.

Adagrad

The Adagrad uses a different learning rate for every parameter at every time step t:

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{\varepsilon I + G(t)}} g(t)$$

where w is the parameter to be updated, η is the initial learning rate, ε is some small quantity that used to avoid the division of zero, I is the identity matrix, $g(t)$ is the gradient estimate in time-step t that we can get with the equation:

$$g(t) = \frac{1}{n} \sum_{i=1}^{n} \nabla_w E(x_i, y_i, w(t))$$

Adagrad

The key element of this algorithm is in the diagonal matrix $G(t)$ which is the sum of the outer product of the gradients until time-step t which is defined by

$$G(t) = \sum_{j=1}^{t} g(j)g(j)^T$$

In case of update of single parameter (weight) we can write previous equation as follow:

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{\varepsilon + \sum_{j=1}^{t} g(j)^2}} g(t)$$

Adagrad

There are case that the effective learning rate is decreased too fast:

- The Adagrad accumulate gradients from the beginning of training. The accumulation of the squared gradients is in the denominator and every added term is positive. The accumulated sum keeps growing during training. Therefore, the learning rate can eventually become infinitesimally small.
- This issue was mitigated by other presented algorithms.

Adadelta

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.

- restricts the window of accumulated past gradients to some fixed size K,
- sum of gradients is recursively defined as a decaying average of all past squared gradients.

M. Zeiler, ADADELTA: An Adaptive Learning Rate Method, Computer Science, 2012

Adadelta

The running average $R[g^2]_t$ at time step t then depends on the previous average and the current gradient (similar to momentum):

$$R[g^2]_t = \gamma R[g^2]_{t-1} + (1-\gamma) g_t^2 .$$

Usually γ is set to around 0.9. The updates in terms of the parameter update vector:

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{\varepsilon I + R[g^2]_t}} g(t) = w(t) - \frac{\eta}{RMS[g]_t} g(t)$$

The main advantage of AdaDelta is that we do not need to set a default learning rate.

Adadelta

The authors define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$R[\Delta\theta^2]_t = \gamma R[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2.$$

The root mean squared error of parameter updates:

$$RMS[\Delta\theta] = \sqrt{R[\Delta\theta^2]_t + \epsilon}$$

We are replacing the learning rate:

$$w(t+1) = w(t) - \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g(t)$$

RMSprop

The central idea of RMSprop is keep the moving average of the squared gradients for each weight. And then we divide the gradient by square root the mean square (those RMSprop).

- RMSprop is focussed on updating the learning.
- Adadelta is not focus on updating the learning rate. The authors explains that the accumulation of square of gradients can be approximated by RMS of gradient and learning rate can be omitted.

G. Hinton, lecture 6, online course "Neural Networks for Machine Learning".

RMSprop

The first update vector of Adadelta and RMSprop is identical:

$$R[g^2]_t = \gamma R[g^2]_{t-1} + \gamma g_t^2.$$

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{\varepsilon I + R[g^2]_t}} g(t)$$

The authors propose default values of $\gamma = 0.9$, and $\eta = 0.001$.

Adam

Adaptive Moment Estimation (Adam) is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks.

- The algorithms leverages the power of adaptive learning rates methods to find individual learning rates for each parameter.
- The algorithms keeps an exponentially decaying average of past **squared gradients** (similar to Adadelta and RMSprop).
- The algorithms also keeps an exponentially decaying average of past **gradients** (similar to momentum).

Adam

We compute the decaying averages of past m_t (first moment, the mean) and past squared gradients v_t (second moment, the uncentered variance):

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned}$$

As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

Adam

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\bar{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\bar{v}_t = \frac{v_t}{1-\beta_2^t}$$

They we update the parameters in the same way as in Adadelta and RMSprop, which yields the Adam update rule:

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{\bar{v}_t} + \varepsilon}\bar{m}_t$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε.

Questions