IT496: Introduction to Data Mining

Lecture 26

The Backpropagation Algorithm

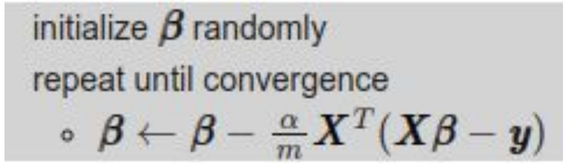
(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana

19th October 2023

(Batch) Gradient Descent

Let's start by reminding ourselves of (Batch) Gradient Descent for OLS regression



```
initialize $\boldsymbol{\beta}$ randomly
repeat until convergence
    ∘ $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \frac{\alpha}{m}\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y})$
```

- We see it making predictions $\hat{y} = X\beta$ for all the examples X and comparing them with target values, y.

- And we see it updating all the parameters β by an amount equal to the negative of the gradients, multiplied by the learning rate $\boldsymbol{\alpha}$.

Why Not Gradient Descent?

For neural networks, however, there is a problem.

- At the output layer, we can straightforwardly compute loss: we can get the network's output (its prediction) and we have the target value, because the training set is a labeled dataset.

- But we cannot straightforwardly compute loss at the hidden layers: we know what outputs their neurons produce but we do not know what they should produce (target values). The labeled dataset doesn't tell us the target values for hidden layer neurons.

The solution is to assume that each neuron in layer $\boldsymbol{l}$ is responsible for some part of the loss of the neurons it is connected to in layer $\boldsymbol{l+1}$.

We will refer to this amount as the '**error signal**'.

Backpropagation: Basic Idea

- This leads to the idea of an algorithm that makes two passes through the network:
  - a ***forward pass*** to make predictions: we feed **X** into the network and then work forwards through the network, computing activations layer by layer until we reach the output layer;
  - a ***backward pass*** to compute the gradients: we calculate loss at the output layer and then work backwards through the network computing error signals layer by layer until we reach the input layer.
- With these two steps completed, we have all the gradients, so we can update all the weights and biases.

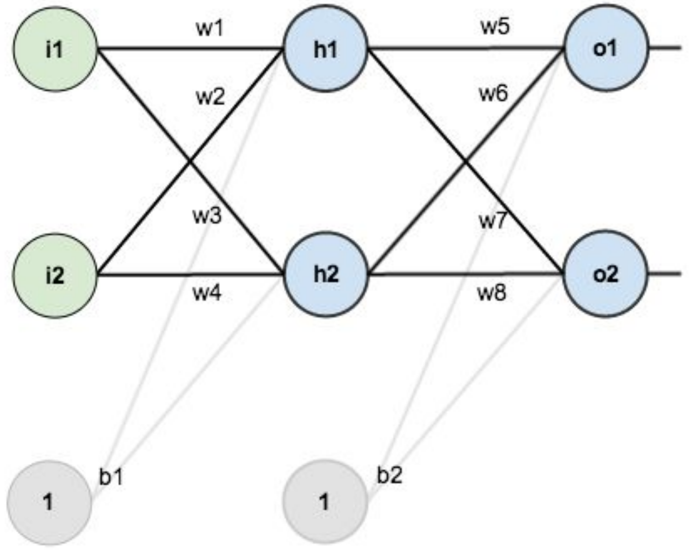
This description helps you see why the calculation of the gradients (and sometimes the entire algorithm) is referred to as **backpropagation** (or just **backprop**).

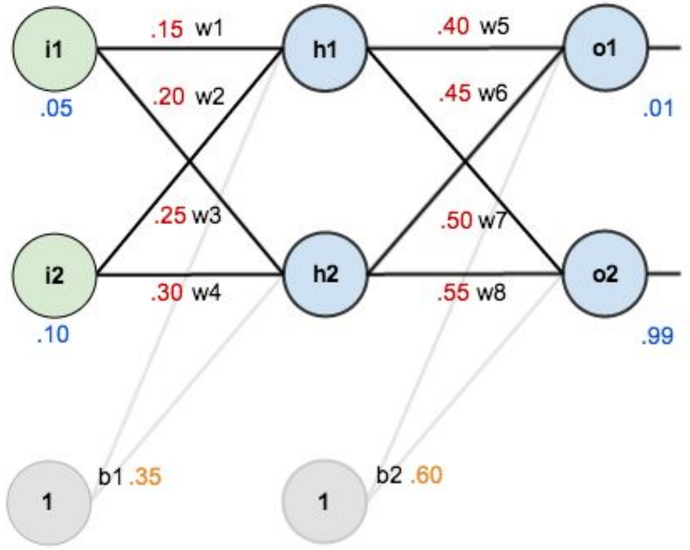
The Backpropagation Algorithm (High Level)

- ***Random initialization:*** initialize all weights and biases randomly
- ***Forward propagation:*** make predictions for all the training examples:
    - Layer by layer from layer 1 to layer *L*:
        - Calculate the inputs to the units in that layer (weighted sums plus biases)
        - Calculate the outputs of the units in that layer (using activation function)
- ***Backpropagation:***
    - Calculate the error signals Δ at layer *L*
    - Layer by layer in reverse from layer *L*-1 to layer 1:
        - Calculate the error signals Δ for the units in that layer
- ***Update all the weights and biases:*** $w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \times a_i \times \Delta_j^{(l)}$

  $b_j^{(l)} = b_j^{(l)} - \alpha \times 1 \times \Delta_j^{(l)}$

The Backpropagation Algorithm (Example)





Here are the initial weights, the biases, and training inputs/outputs:

Source: A Blog by Matt Mazur

The Backpropagation Algorithm (Example): Forward Pass

Here's how we calculate the total net input for h_1:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

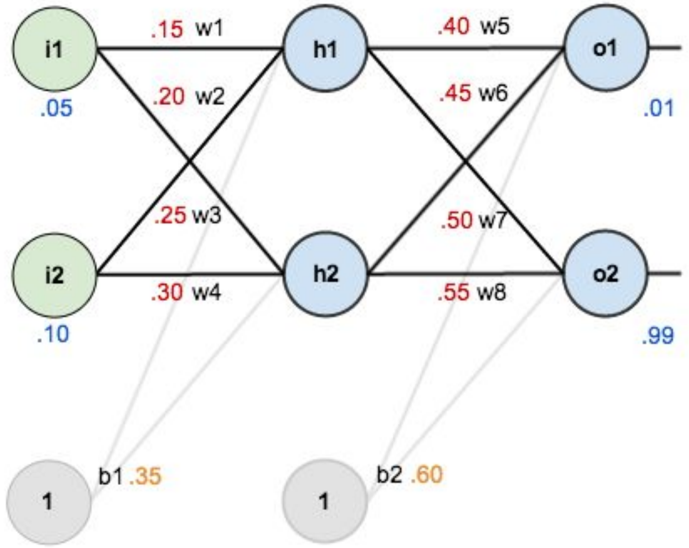
$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

We then squash it using the logistic function to get the output of h_1:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

Carrying out the same process for h_2 we get:

$$out_{h2} = 0.596884378$$



Source: A Blog by Matt Mazur

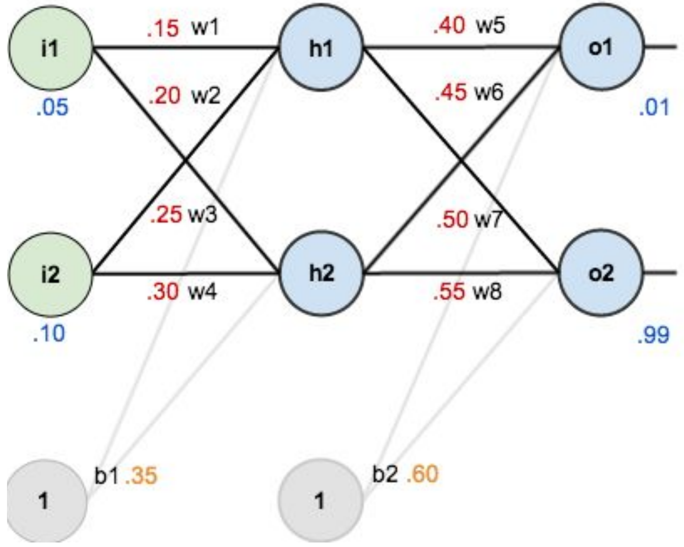
The Backpropagation Algorithm (Example): Forward Pass

Similarly, for the output layer neurons, using the output from the hidden layer neurons as inputs. Here's the output for o_1:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1$$
$$= 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$



Carrying out the same process for o_2 we get:

$$out_{o2} = 0.772928465$$

Source: A Blog by Matt Mazur

The Backpropagation Algorithm (Example): Forward Pass

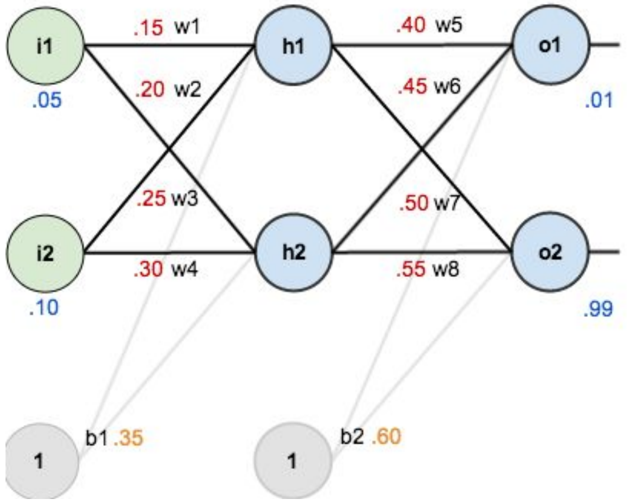
We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = 0.023560026$$

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$



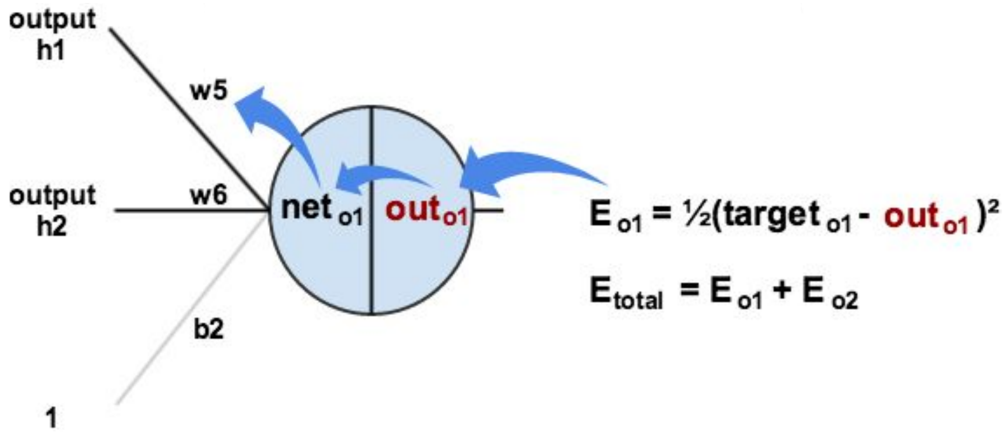
Source: A Blog by Matt Mazur

The Backpropagation Algorithm (Example): Backward Pass

Output Layer:

Consider w_5. We want to know how much a change in w_5 affects the *total error*. By applying the *chain rule*, we know that -

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$


output h1
w5
output h2
w6
b2
1
net o1
out o1
$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2$
$E_{total} = E_{o1} + E_{o2}$

Source: A Blog by Matt Mazur

The Backpropagation Algorithm (Example): Backward Pass

First, how much does the total error change with respect to the output?

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

Next, how much does the output of o_1 change with respect to its total *net* input?

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

Source: A Blog by Matt Mazur

The Backpropagation Algorithm (Example): Backward Pass

Finally, how much does the total net input of o_1 change with respect to w_5?

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

So, we update w_5 as per the gradient update rule:

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

We can repeat this process to get the new weights w_6, w_7, and w_8:

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

What about updating the bias b_2?

Source: A Blog by Matt Mazur

The Backpropagation Algorithm (Example): Backward Pass

Hidden Layer:

Next, we'll continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4.

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05$$

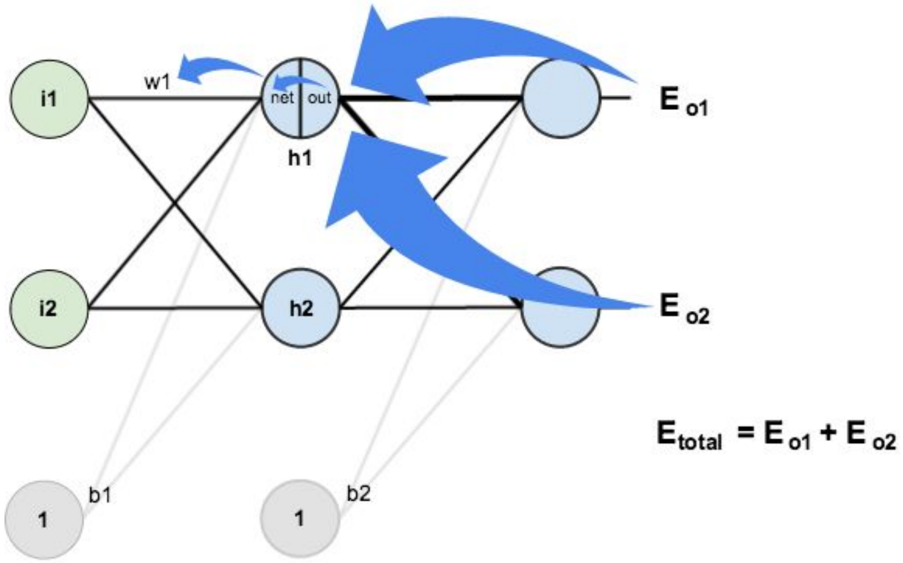
$$= 0.000438568$$

$$w_1^+ = 0.149780716$$

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$



Source: A Blog by Matt Mazur

The Backpropagation Algorithm (Example)

- When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of backpropagation, the total error is now down to 0.291027924.

- After repeating this process 10,000 times, for example, the error plummets to 0.0000351085.

- At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

Source: A Blog by Matt Mazur

Auto-differentiation (Autodiff)

- We have shown the update rules for a particular network and a particular loss function.
- We would get different update rules if we change:
  - the network, e.g. layers other than dense layers (batch normalization layers, convolutional layers, etc.); and/or
  - the loss function.
- Happily, we don't have to manually find the partial derivatives all over again.
- Neural networks consist of layers of operations, each with simple, known derivatives.

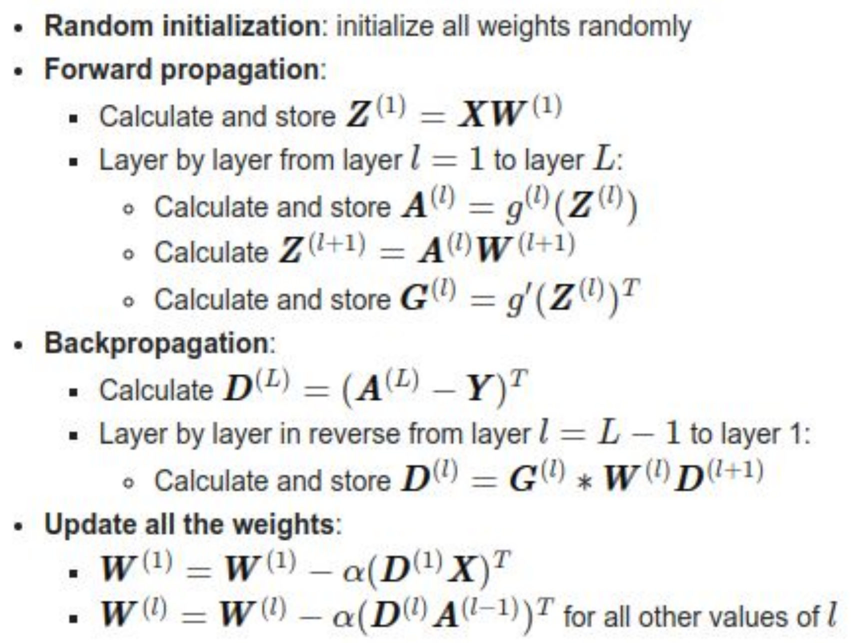
Auto-differentiation (Autodiff)

- Given that the network simply defines a composite function (see previous lecture), the derivatives for the whole network can be obtained automatically by repeated use of the ***chain rule***:
    - To differentiate a function of a function, `y = f(g(x))`, let `u = g(x)` so that `y = f(u)`, then

$$\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$$

- Hence, modern frameworks such as TensorFlow can compute gradients automatically in the backpropagation step.
- This is known as ***autodiff*** (or, for the way it is used by backprop, ***reverse mode autodiff***).

The Backpropagation Algorithm: Vectorized Form



- **Random initialization**: initialize all weights randomly
- **Forward propagation**:
  - Calculate and store $\boldsymbol{Z}^{(1)} = \boldsymbol{X}\boldsymbol{W}^{(1)}$
  - Layer by layer from layer $l = 1$ to layer L:
    - Calculate and store $\boldsymbol{A}^{(l)} = g^{(l)}(\boldsymbol{Z}^{(l)})$
    - Calculate $\boldsymbol{Z}^{(l+1)} = \boldsymbol{A}^{(l)}\boldsymbol{W}^{(l+1)}$
    - Calculate and store $\boldsymbol{G}^{(l)} = g'(\boldsymbol{Z}^{(l)})^T$
- **Backpropagation**:
  - Calculate $\boldsymbol{D}^{(L)} = (\boldsymbol{A}^{(L)} - \boldsymbol{Y})^T$
  - Layer by layer in reverse from layer $l = L - 1$ to layer 1:
    - Calculate and store $\boldsymbol{D}^{(l)} = \boldsymbol{G}^{(l)} * \boldsymbol{W}^{(l)}\boldsymbol{D}^{(l+1)}$
- **Update all the weights**:
  - $\boldsymbol{W}^{(1)} = \boldsymbol{W}^{(1)} - \alpha(\boldsymbol{D}^{(1)}\boldsymbol{X})^T$
  - $\boldsymbol{W}^{(l)} = \boldsymbol{W}^{(l)} - \alpha(\boldsymbol{D}^{(l)}\boldsymbol{A}^{(l-1)})^T$ for all other values of l

- For simplicity, biases are omitted.
- We can see in this more precise version that some of the things that are calculated on the forward pass get stored.
  - These things can the be used on the backward pass.
- Similarly, some of the things that are calculated on the backward pass get stored.
  - These things can then be used to update the weights.
- This makes backprop much more efficient than it otherwise would be.

Next lecture

Training Deep Neural Network

20th October 2023