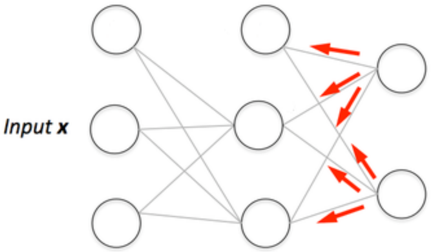
Backpropagation Step by Step

03 Nov 2019

Less than 100 pages covering Kotlin syntax and features in straight and to the point explanation.



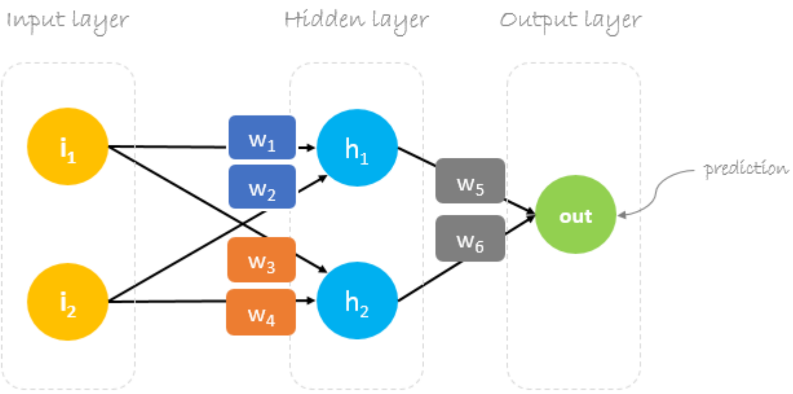
If you are building your own neural network, you will definitely need to understand how to train it. Backpropagation is a commonly used technique for training neural network. There are many resources explaining the technique, but this post will explain backpropagation with concrete example in a very detailed colorful steps.

> You can see visualization of the forward pass and backpropagation here.
> You can build your neural network using **netflow.js**

Overview

In this post, we will build a neural network with three layers:

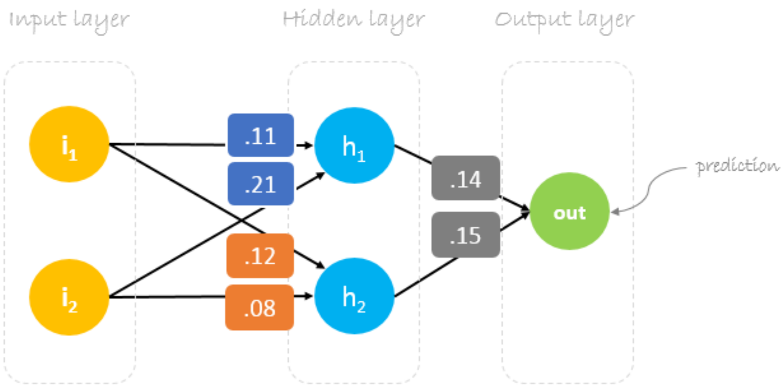
- **Input** layer with two inputs neurons
- One **hidden** layer with two neurons
- **Output** layer with a single neuron



Weights, weights, weights

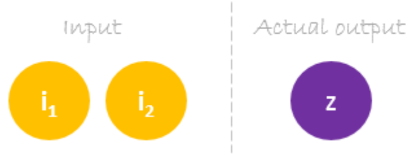
Neural network training is about finding weights that minimize prediction error. We usually start our training with a set of randomly generated weights.Then, backpropagation is used to update the weights in an attempt to correctly map arbitrary inputs to outputs.

Our initial weights will be as following: `w1 = 0.11`, `w2 = 0.21`, `w3 = 0.12`, `w4 = 0.08`, `w5 = 0.14` and `w6 = 0.15`



Dataset

Our dataset has one sample with two inputs and one output.

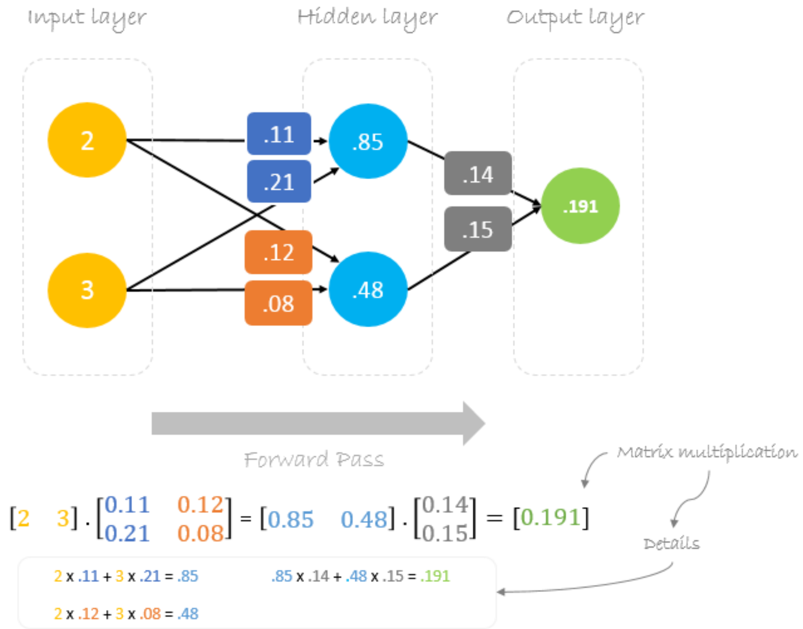


Our single sample is as following `inputs=[2, 3]` and `output=[1]`.

Input
Actual output
2
3
1

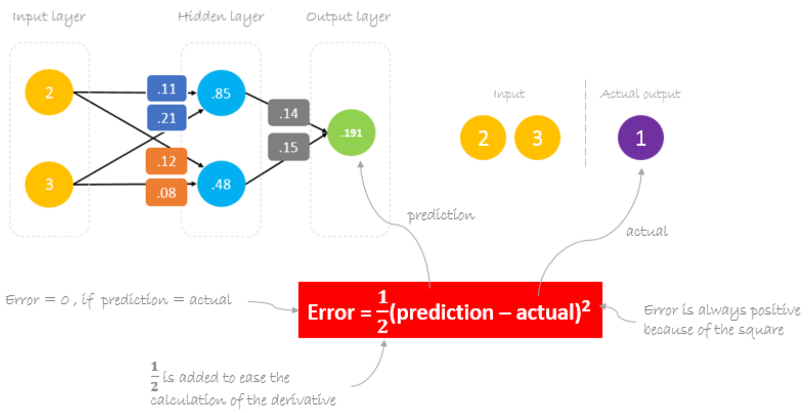
Forward Pass

We will use given weights and inputs to predict the output. Inputs are multiplied by weights; the results are then passed forward to next layer.


Input layer
Hidden layer
Output layer
2
3
.11
.21
.12
.08
.85
.48
.14
.15
.191
Forward Pass
Matrix multiplication
Details
$[2 \quad 3] \cdot \begin{bmatrix} 0.11 & 0.12 \\ 0.21 & 0.08 \end{bmatrix} = [0.85 \quad 0.48] \cdot \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} = [0.191]$
2 x .11 + 3 x .21 = .85
2 x .12 + 3 x .08 = .48
.85 x .14 + .48 x .15 = .191

Calculating Error

Now, it's time to find out how our network performed by calculating the difference between the actual output and predicted one. It's clear that our network output, or **prediction**, is not even close to **actual output**. We can calculate the difference or the error as following.


Input layer
Hidden layer
Output layer
2
3
.11
.21
.12
.08
.85
.48
.14
.15
.191
Input
Actual output
2
3
1
prediction
actual
Error = 0 , if prediction = actual
Error = $\frac{1}{2}$(prediction – actual)^2
Error is always positive because of the square
$\frac{1}{2}$ is added to ease the calculation of the derivative

$$\text{Error} = \frac{1}{2}(0.191 - 1.0)^2 = 0.327$$

Reducing Error

Our main goal of the training is to reduce the **error** or the difference between **prediction** and **actual output**. Since **actual output** is constant, “not changing”, the only way to reduce the error is to change **prediction** value. The question now is, how to change **prediction** value?

By decomposing **prediction** into its basic elements we can find that **weights** are the variable elements affecting **prediction** value. In other words, in order to change **prediction** value, we need to change **weights** values.

$$\text{prediction} = \text{out}$$

$$\Downarrow$$

$$\text{prediction} = (h_1)\, w_5 + (h_2)\, w_6$$

$$h_1 = i_1 w_1 + i_2 w_2$$
$$h_2 = i_1 w_3 + i_2 w_4$$

$$\Downarrow$$

$$\text{prediction} = (i_1\, w_1 + i_2\, w_2)\, w_5 + (i_1\, w_3 + i_2\, w_4)\, w_6$$

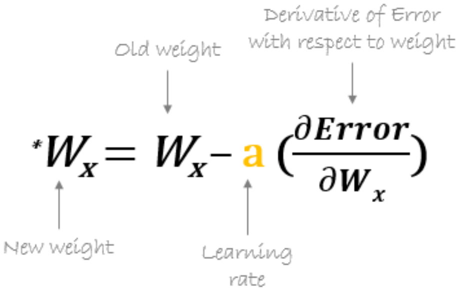
to change ***prediction*** value,
we need to change ***weights***

> The question now is **how to change\update the weights value so that the error is reduced?**
> The answer is **Backpropagation!**

Backpropagation

Backpropagation, short for “backward propagation of errors”, is a mechanism used to update the **weights** using gradient descent. It calculates the gradient of the error function with respect to the neural network’s weights. The calculation proceeds backwards through the network.

> **Gradient descent** is an iterative optimization algorithm for finding the minimum of a function; in our case we want to minimize th error function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point.

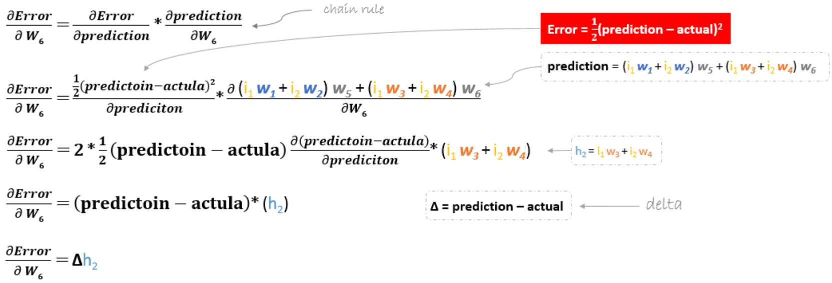


$$^{*}W_x = W_x - a\left(\frac{\partial Error}{\partial W_x}\right)$$

For example, to update `w6`, we take the current `w6` and subtract the partial derivative of **error** function with respect to `w6`. Optionally, we multiply the derivative of the **error** function by a selected number to make sure that the new updated **weight** is minimizing the error function; this number is called ***learning rate***.

$$^{*}W_6 = W_6 - a\left(\frac{\partial Error}{\partial W_6}\right)$$

The derivation of the error function is evaluated by applying the chain rule as following



$$\frac{\partial Error}{\partial W_6} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial W_6}$$

$$\frac{\partial Error}{\partial W_6} = \frac{\frac{1}{2}(predictoin - actula)^2}{\partial prediciton} * \frac{\partial\,(i_1 w_1 + i_2 w_2)\, w_5 + (i_1 w_3 + i_2 w_4)\, w_6}{\partial W_6}$$

$$\frac{\partial Error}{\partial W_6} = 2 * \frac{1}{2}(\mathbf{predictoin} - \mathbf{actula})\,\frac{\partial(predictoin - actula)}{\partial prediciton} * (i_1 w_3 + i_2 w_4)$$

$$\frac{\partial Error}{\partial W_6} = (\mathbf{predictoin} - \mathbf{actula}) * (h_2)$$

$$\frac{\partial Error}{\partial W_6} = \Delta h_2$$

chain rule

Error = $\frac{1}{2}$(prediction – actual)^2

prediction = $(i_1 w_1 + i_2 w_2)\, w_5 + (i_1 w_3 + i_2 w_4)\, w_6$

$h_2 = i_1 w_3 + i_2 w_4$

Δ = prediction – actual

delta

So to update `w6` we can apply the following formula

$$^{*}W_6 = W_6 - a\,\Delta h_2$$

Similarly, we can derive the update formula for `w5` and any other weights existing between the output and the hidden layer.

$$^{*}W_5 = W_5 - a\,\Delta h_1$$

However, when moving backward to update `w1`, `w2`, `w3` and `w4` existing between input and hidden layer, the partial derivative for the error function with respect to `w1`, for example, will be as following.

$$\frac{\partial Error}{\partial\, W_1} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial h_1} * \frac{\partial h_1}{\partial W_1}$$

chain rule

$$\text{Error} = \tfrac{1}{2}(\text{prediction} - \text{actual})^2$$

$$\text{prediction} = (h_1)\, w_5 + (h_2)\, w_6$$

$$h_1 = i_1 w_1 + i_2 w_2$$

$$\frac{\partial Error}{\partial W_1} = \frac{\partial \frac{1}{2}(predictoin - actula)^2}{\partial prediciton} * \frac{\partial\, (h_1)\, w_5 + (h_2)\, w_6}{\partial h_1} * \frac{\partial\, i_1\, w_1 + i_2\, w_2}{\partial w_1}$$

$$\frac{\partial Error}{\partial W_1} = 2 * \frac{1}{2}(\mathbf{predictoin - actula})\, \frac{\partial (predictoin - actula)}{\partial prediciton} * (w_5) * (i_1)$$

$$\frac{\partial Error}{\partial W_1} = (\mathbf{predictoin - actula}) * (w_5 i_1)$$

$$\Delta = \text{prediction} - \text{actual}$$

delta

$$\frac{\partial Error}{\partial W_1} = \Delta\, w_5 i_1$$

We can find the update formula for the remaining weights `w2`, `w3` and `w4` in the same way.

In summary, the update formulas for all weights will be as following:

updated weights

$$
\begin{aligned}
{}^*w_6 &= w_6 - a\,(h_2\ .\ \Delta)\\
{}^*w_5 &= w_5 - a\,(h_1\ .\ \Delta)\\
{}^*w_4 &= w_4 - a\,(i_2\ .\ \Delta w_6)\\
{}^*w_3 &= w_3 - a\,(i_1\ .\ \Delta w_6)\\
{}^*w_2 &= w_2 - a\,(i_2\ .\ \Delta w_5)\\
{}^*w_1 &= w_1 - a\,(i_1\ .\ \Delta w_5)
\end{aligned}
$$

We can rewrite the update formulas in matrices as following

$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - a\,\Delta \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - \begin{bmatrix} ah_1\Delta \\ ah_2\Delta \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - a\,\Delta \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} . \begin{bmatrix} w_5 & w_6 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - \begin{bmatrix} a\, i_1 \Delta w_5 & a\, i_1 \Delta w_6 \\ a\, i_2 \Delta w_5 & a\, i_2 \Delta w_6 \end{bmatrix}$$

Backward Pass

Using derived formulas we can find the new **weights**.

> **Learning rate:** is a hyperparameter which means that we need to manually guess its value.

$$\Delta = 0.191 - 1 = \text{-}0.809$$

Delta = prediction - actual

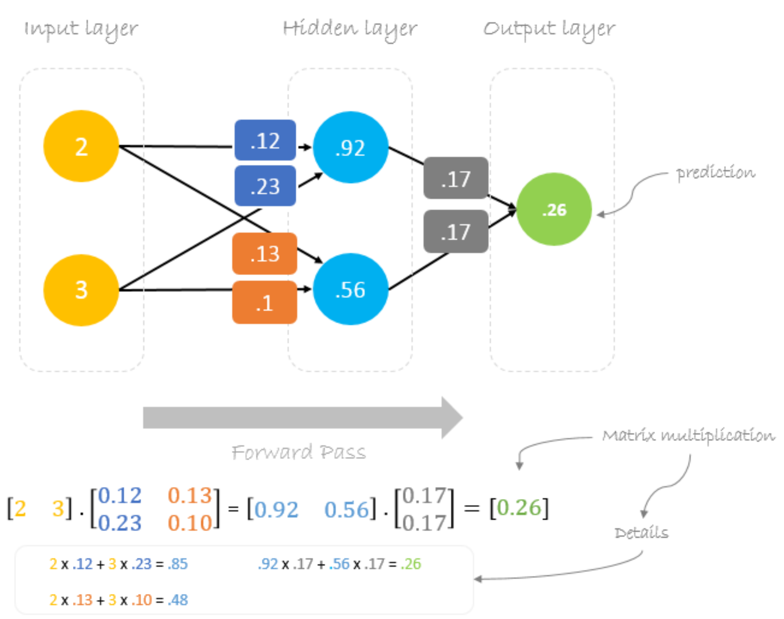
$$a = 0.05$$

Learning rate, we smartly guess this number

$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} - 0.05(\text{-}0.809) \begin{bmatrix} 0.85 \\ 0.48 \end{bmatrix} = \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} - \begin{bmatrix} \text{-}0.034 \\ \text{-}0.019 \end{bmatrix} = \begin{bmatrix} 0.17 \\ 0.17 \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} .11 & .12 \\ .21 & .08 \end{bmatrix} - 0.05(-0.809) \begin{bmatrix} 2 \\ 3 \end{bmatrix} . \begin{bmatrix} 0.14 & 0.15 \end{bmatrix} = \begin{bmatrix} .11 & .12 \\ .21 & .08 \end{bmatrix} - \begin{bmatrix} \text{-}0.011 & \text{-}0.012 \\ \text{-}0.017 & \text{-}0.018 \end{bmatrix} = \begin{bmatrix} .12 & .13 \\ .23 & .10 \end{bmatrix}$$

Now, using the new **weights** we will repeat the forward passed

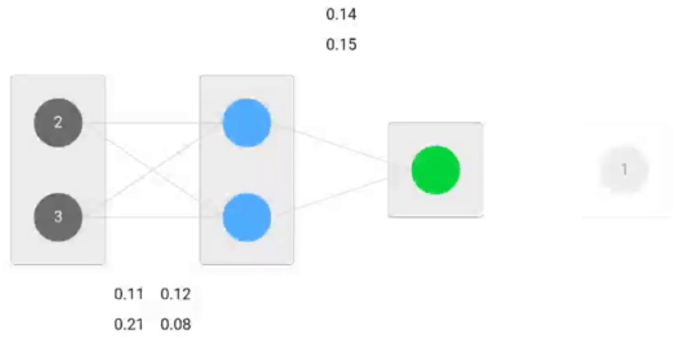


We can notice that the **prediction** `0.26` is a little bit closer to **actual output** than the previously predicted one `0.191`. We can repeat the same process of backward and forward pass until **error** is close or equal to zero.

Backpropagation Visualization

You can see visualization of the forward pass and backpropagation here.

You can build your neural network using **netflow.js**


0.14
0.15
2
3
1
0.11 0.12
0.21 0.08