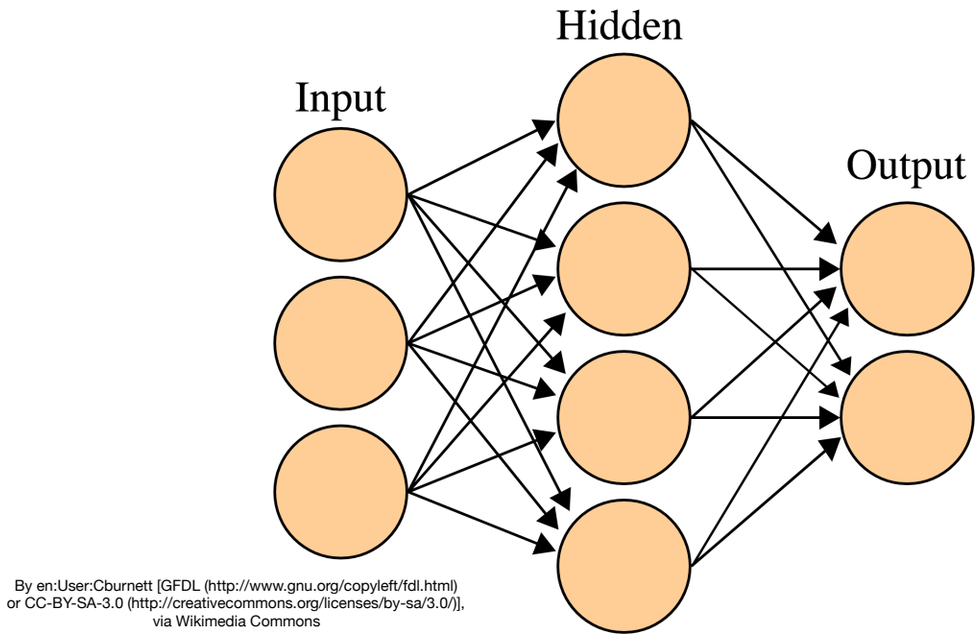
Neural Network Training

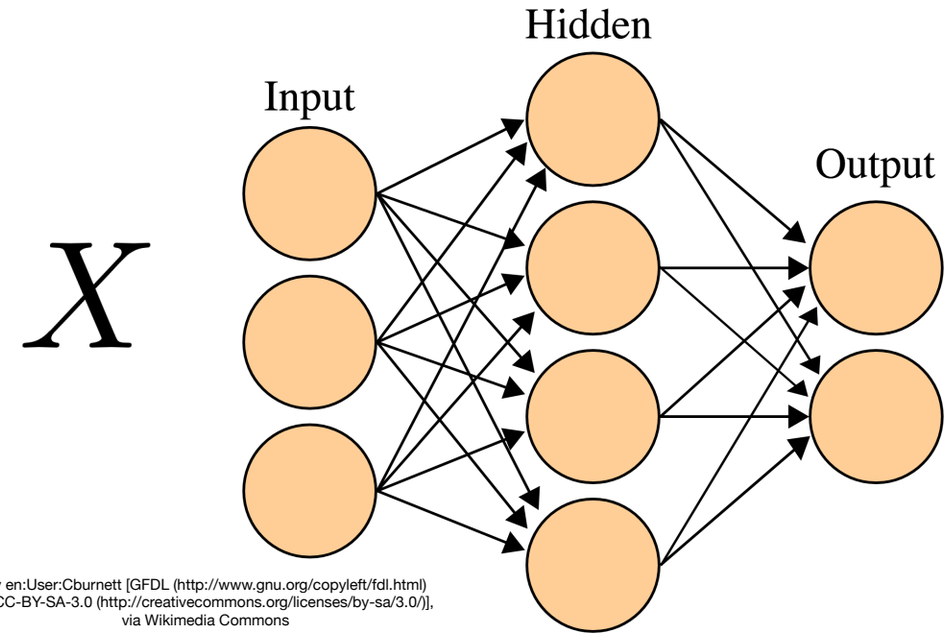
 Romeo Kienzler

So in order to get an idea how neural networks are trained we'll use very simple neural network in python.



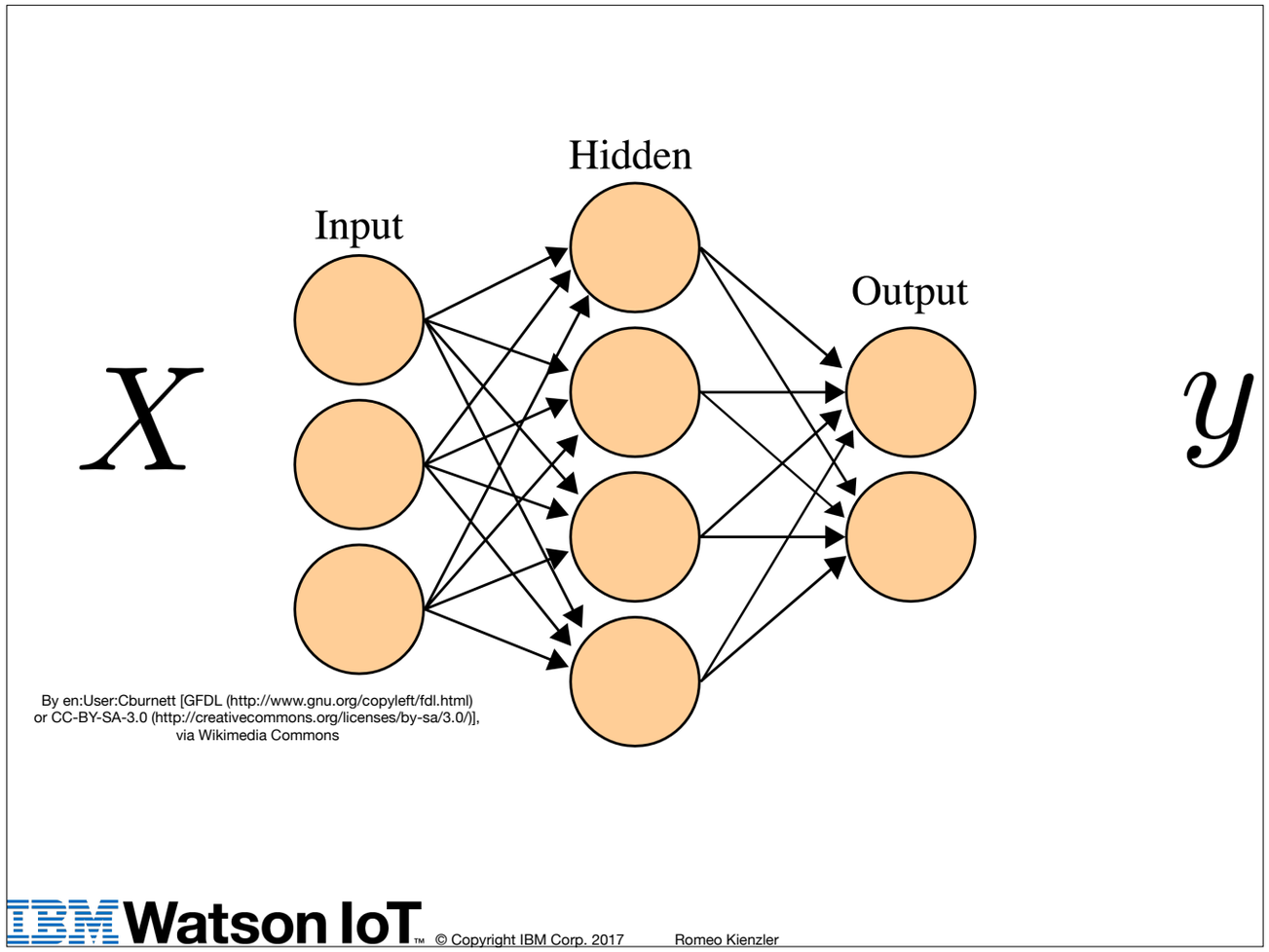
By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

So first of all let’s recap how the so called forward pass of a feed forward neural network is computed



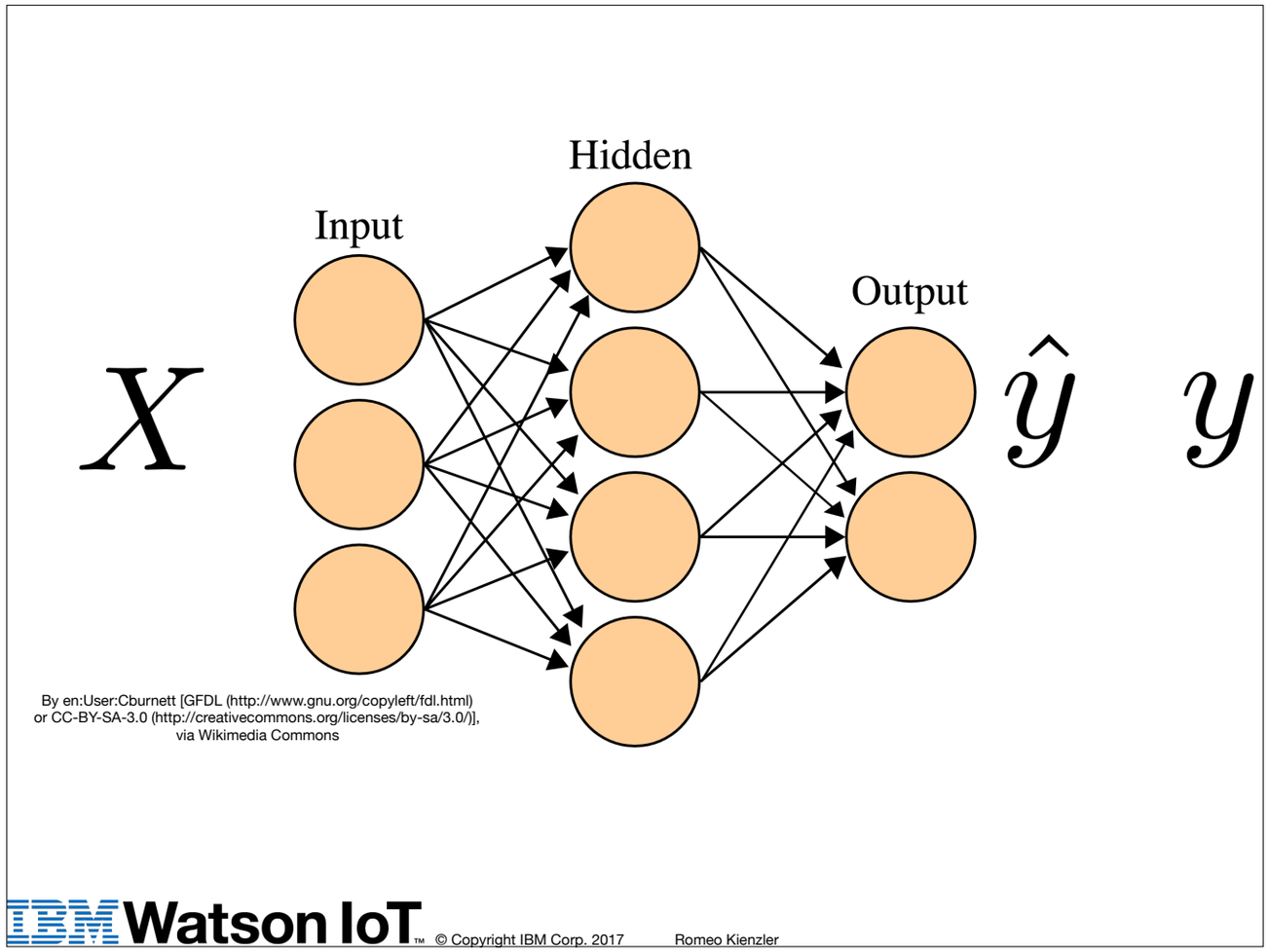
By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

Remember that you have an input vector X…



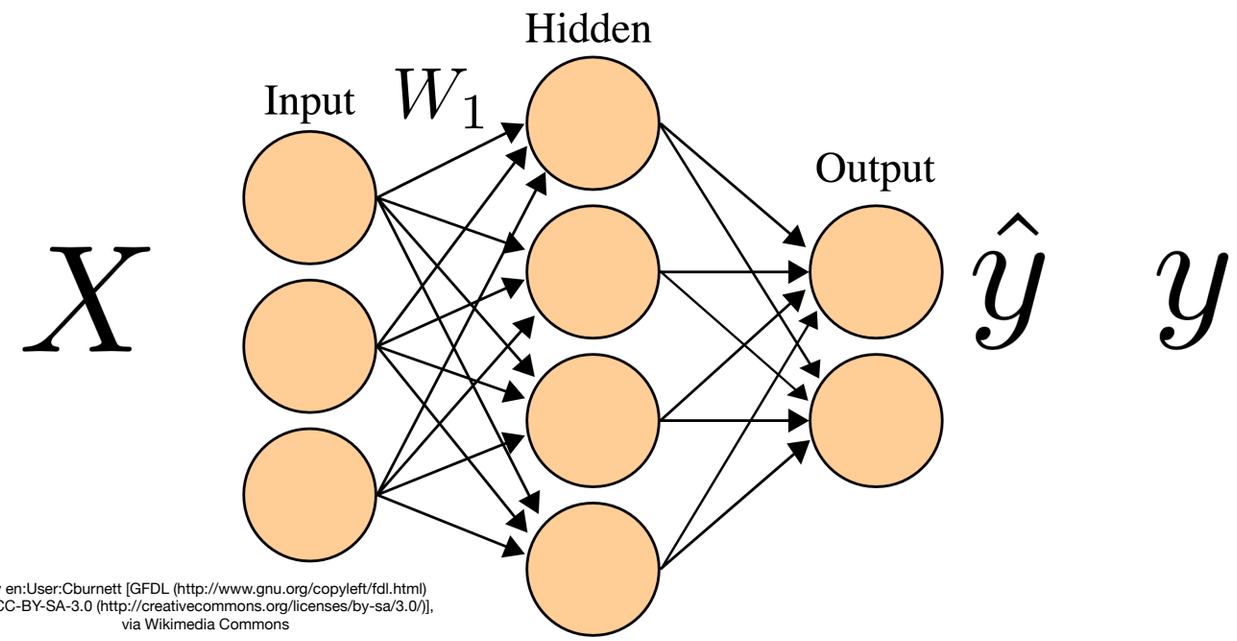
By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

…and a label vector y. Especially in the case of classification tasks y is a vector. For regression y is often a scalar, but for the math it doesn't matter since we are working with vectors and matrices anyway.



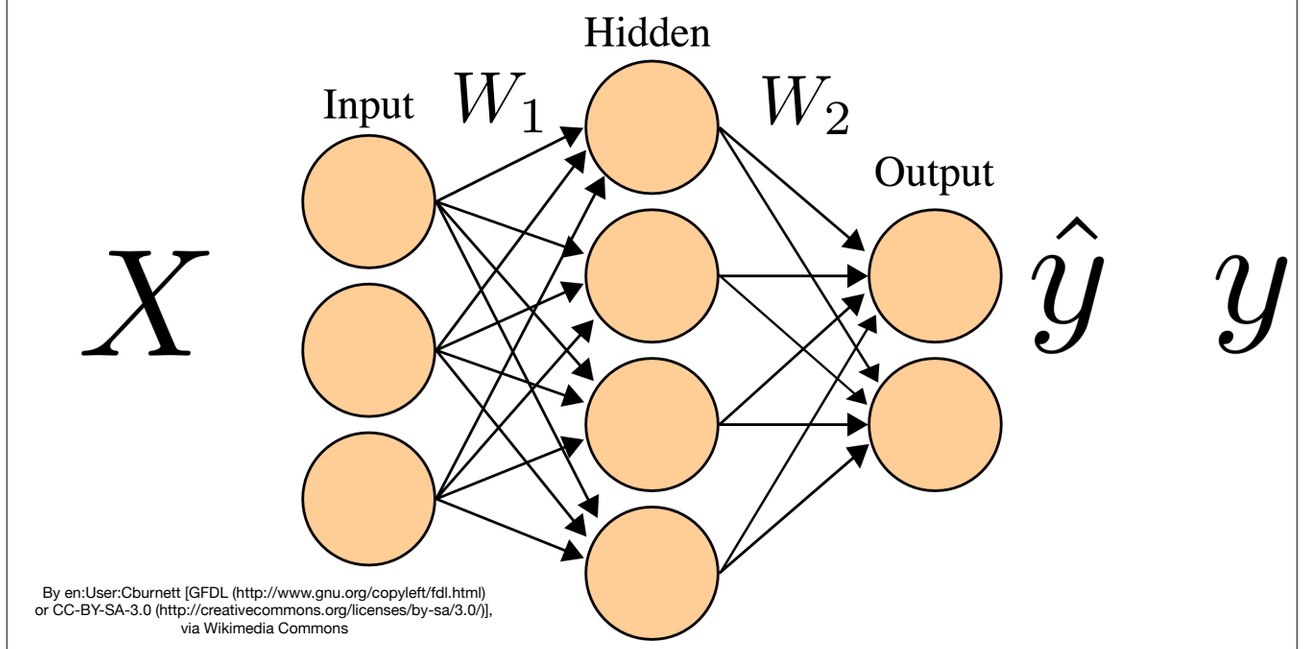
By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

So the task of the neural network is now to compute y hat from the input vector X…



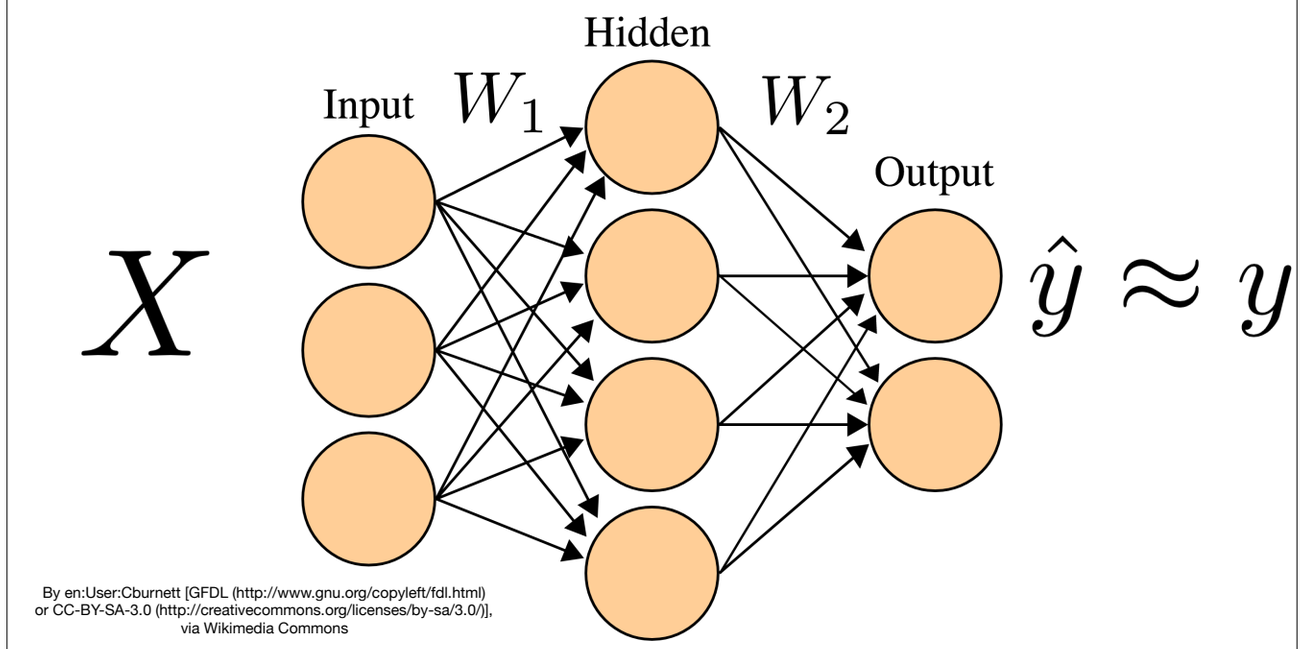
By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

… using the weight matrices double u one and



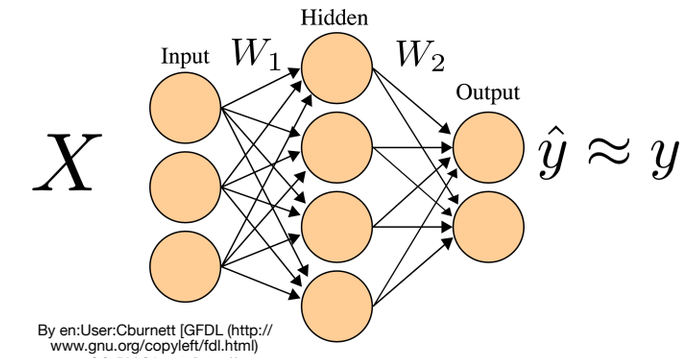
By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

…double u two …



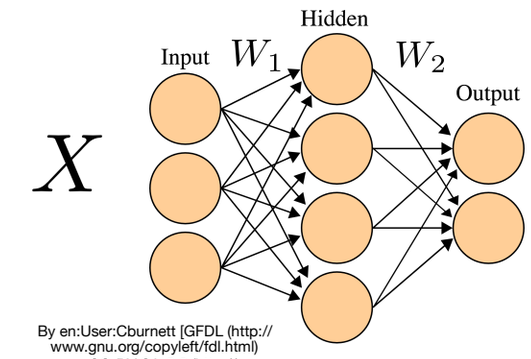
By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

…such that y hat and y are as close as possible for all vectors in the training data set X



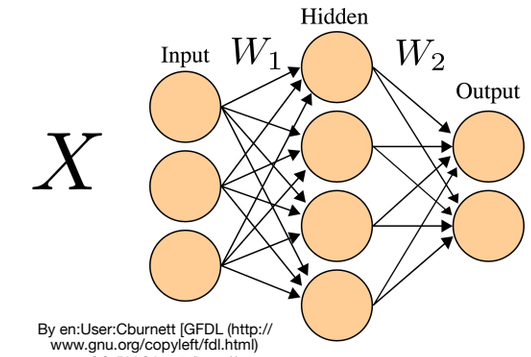
By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

Let’s actually have a look at the math. So to compute from left to right the first thing you do is



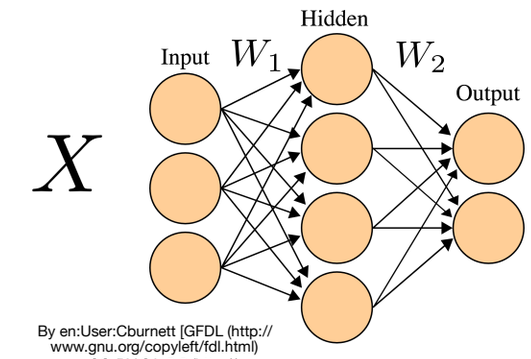
$$z_2 = XW_1$$

multiplying X by double u one to obtain zee two



$$z_2 = XW_1$$
$$a_2 = f(z_2)$$

Since we are squashing intermediate layer calculations with activation functions we have to apply the activation function to zee two to obtain a two



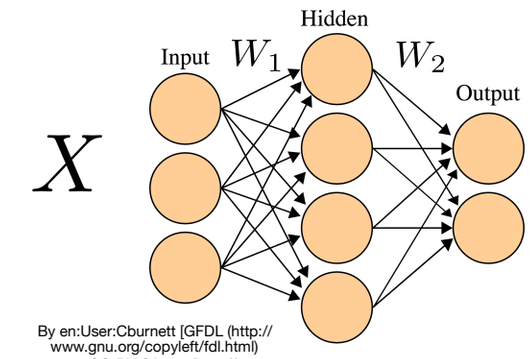
$$f(x) = sigmoid(x)$$

$$z_2 = XW_1$$

$$a_2 = f(z_2)$$

$$z_3 = a_2W_2$$

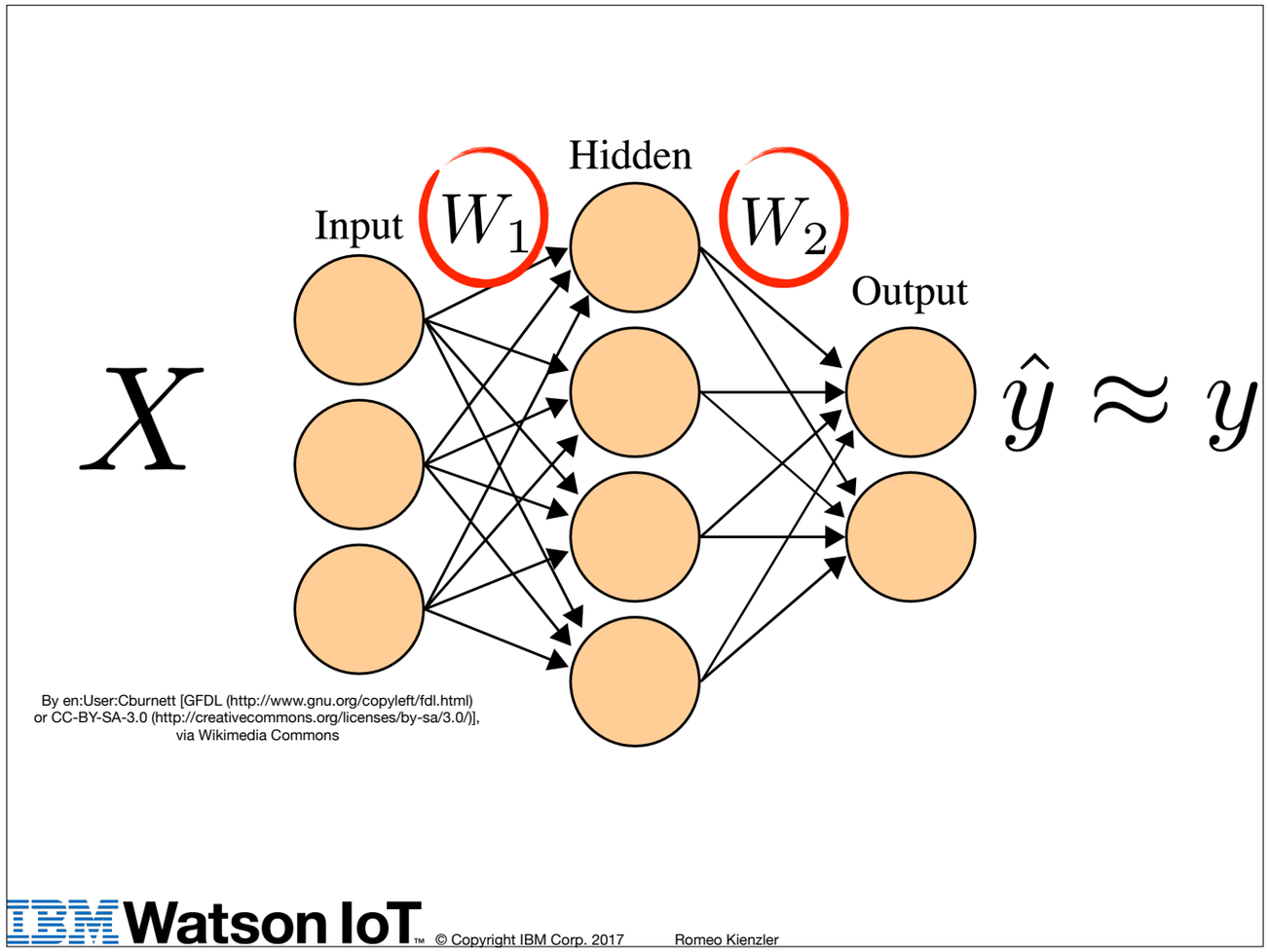
then we compute the output layer. so note that we are multiplying a two by double u two. This is some sort of stacking the computations on top of each other.



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

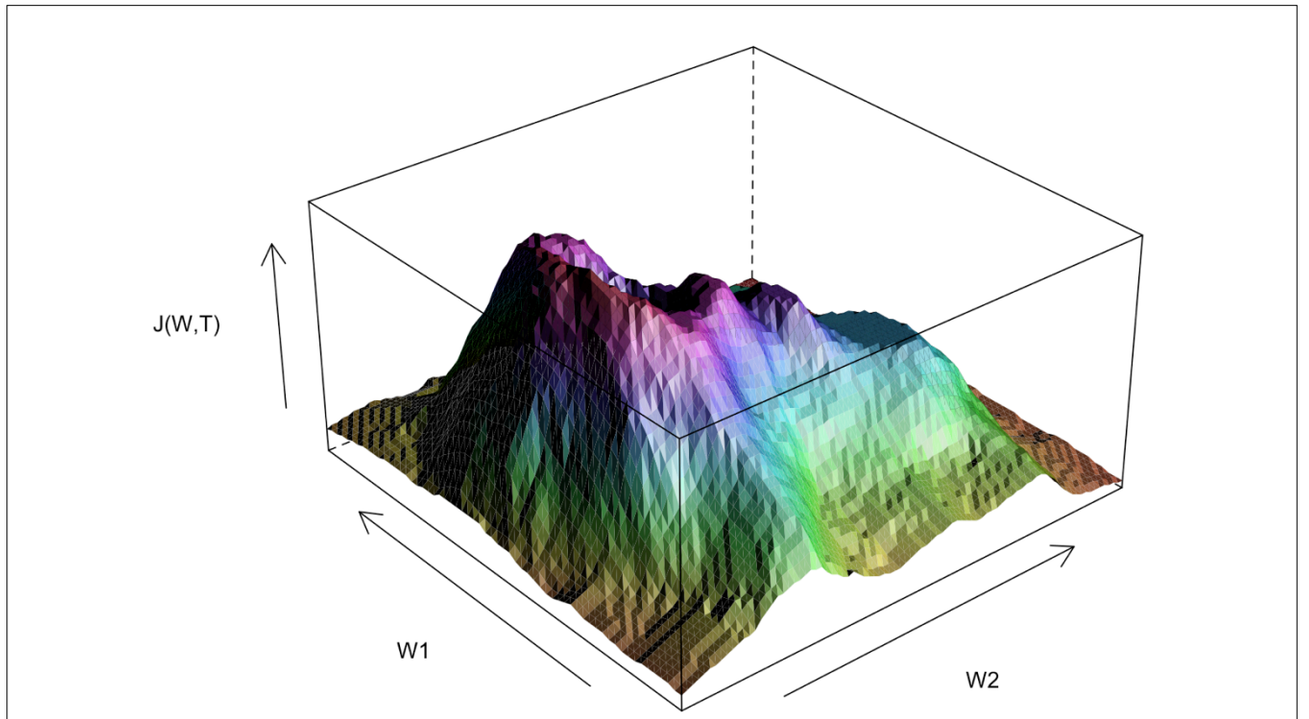
$$f(x) = sigmoid(x)$$
$$z_2 = XW_1$$
$$a_2 = f(z_2)$$
$$z_3 = a_2 W_2$$
$$\hat{y} = a_3 = f(z_3)$$

and finally we compute y hat by applying the activation function to zee 3 and we are done.

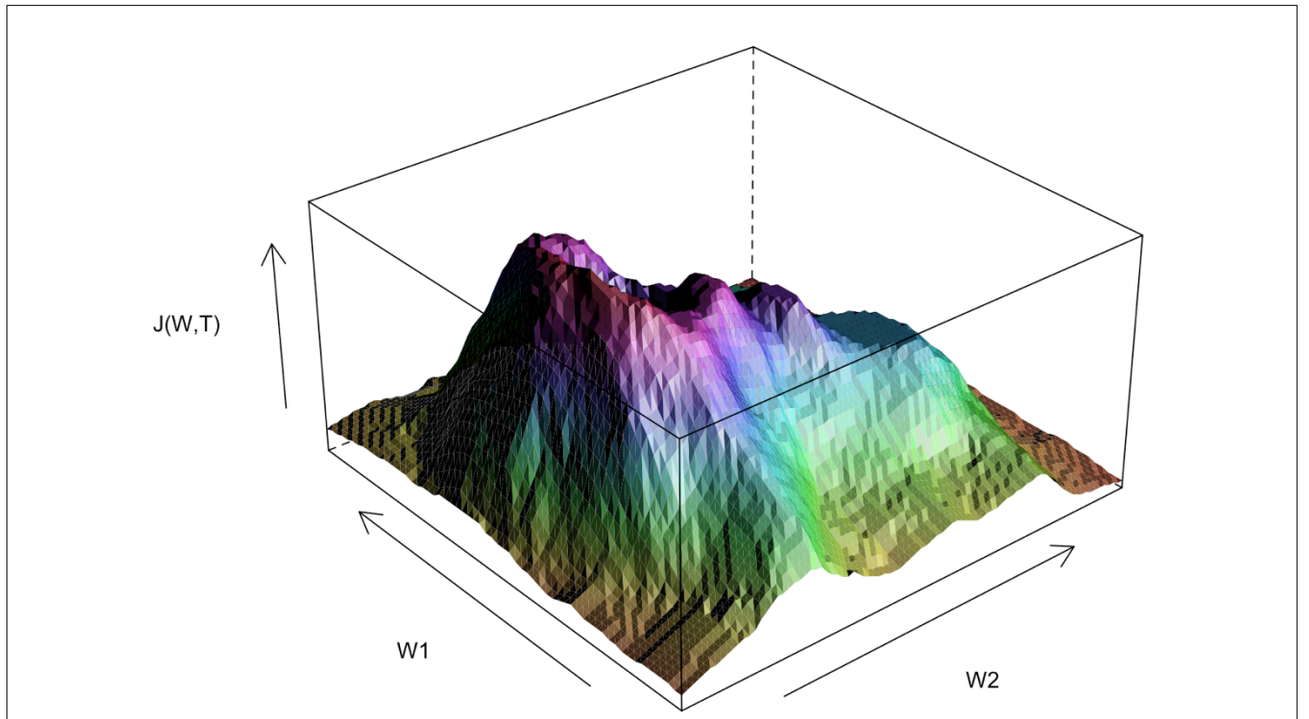


By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

So our weight matrices double u one and double u two are still randomly initialised. Therefore y hat and y are different. So now we have to train this neural network to do a better job. And training basically means updating the weights in order to obtain the best y hat on a given x with respect to y for all data points in X. But how do we find the optimal values for the weights?



This is a plot over two weight values and the cost. So the cost basically tells you how good or bad your weights are chosen. The lower the cost the better the neural network performs. The cost function is called J. Note that it depends on double u and T. So double u is the weight matrix of the neural network. This is the one we want to optimise over. And T stands for the training data which stays always the same. Therefore J is only dependent on double u. This is a two dimensional example, but in reality this so called optimisation space is very high dimensional and impossible to plot. So how can we now find good values for double u? Of course we simply can’t check for every possible combination of values for double u. Or can we?



So on way of doing this is called grid search. So in this example we choose a discrete set of values within double u one and double u two and just apply the cost function J to all of those combinations and the one combination with the lowest cost is an optimal value set of double u.

$v = [-1, 1]$

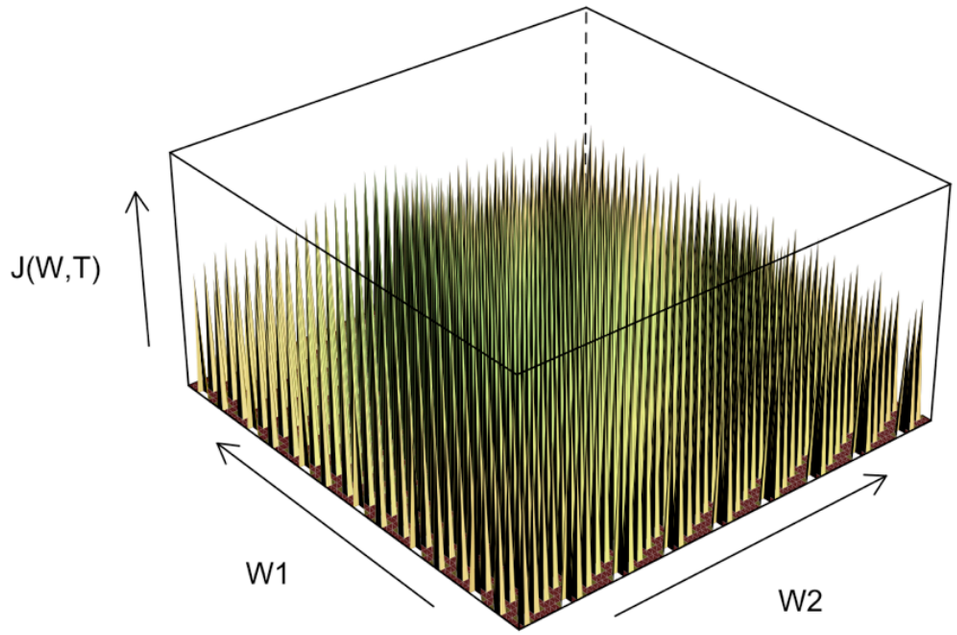
Now imagine that for each individual scalar weight we can assign a value between minus one and one.

$$v = [-1, 1]$$
$$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$$

Now we create a grid over this continuous range with a step size of zero dot one.

$$v = [-1, 1]$$
$$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$$

Now we create a grid over this continuous range with a step size of zero dot one.



So this basically from a grid - therefore this method is called grid search

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$

And now imagine we have only a single scalar weight value to optimise over. So we have to test twenty one individual values

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$

If we add one more weight we already have to test four hundred eleven

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$

In thee dimensions we are on nine thousand sixty one combinations

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$

In four dimensions we already have to test and compute a hundred ninety four thousand four hundred eight one values

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$

..and…

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$

..as we see this number keeps growing…

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$
$dim(8) = 37822859361$

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$
$dim(8) = 37822859361$
$dim(9) = 794280046581$

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$
$dim(8) = 37822859361$
$dim(9) = 794280046581$
$dim(10) = 16679880978201$

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$
$dim(8) = 37822859361$
$dim(9) = 794280046581$
$dim(10) = 16679880978201$
$dim(11) = 350277500542221$

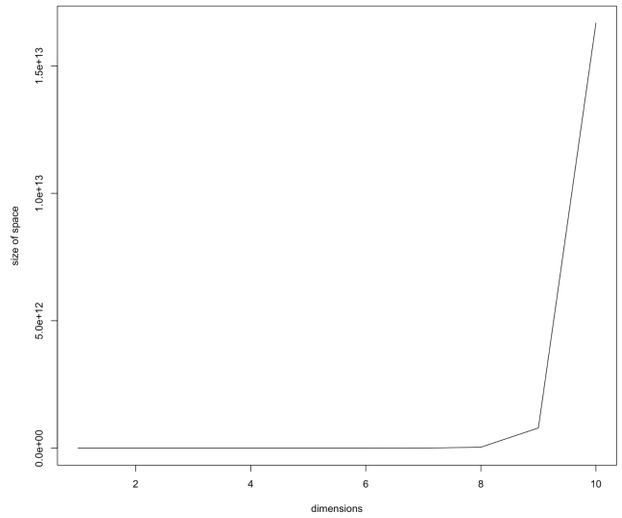
$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$
$dim(8) = 37822859361$
$dim(9) = 794280046581$
$dim(10) = 16679880978201$
$dim(11) = 350277500542221$
$dim(12) = 7355827511386641$

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$
$dim(8) = 37822859361$
$dim(9) = 794280046581$
$dim(10) = 16679880978201$
$dim(11) = 350277500542221$
$dim(12) = 7355827511386641$
$dim(13) = 1.544723777391195e17$

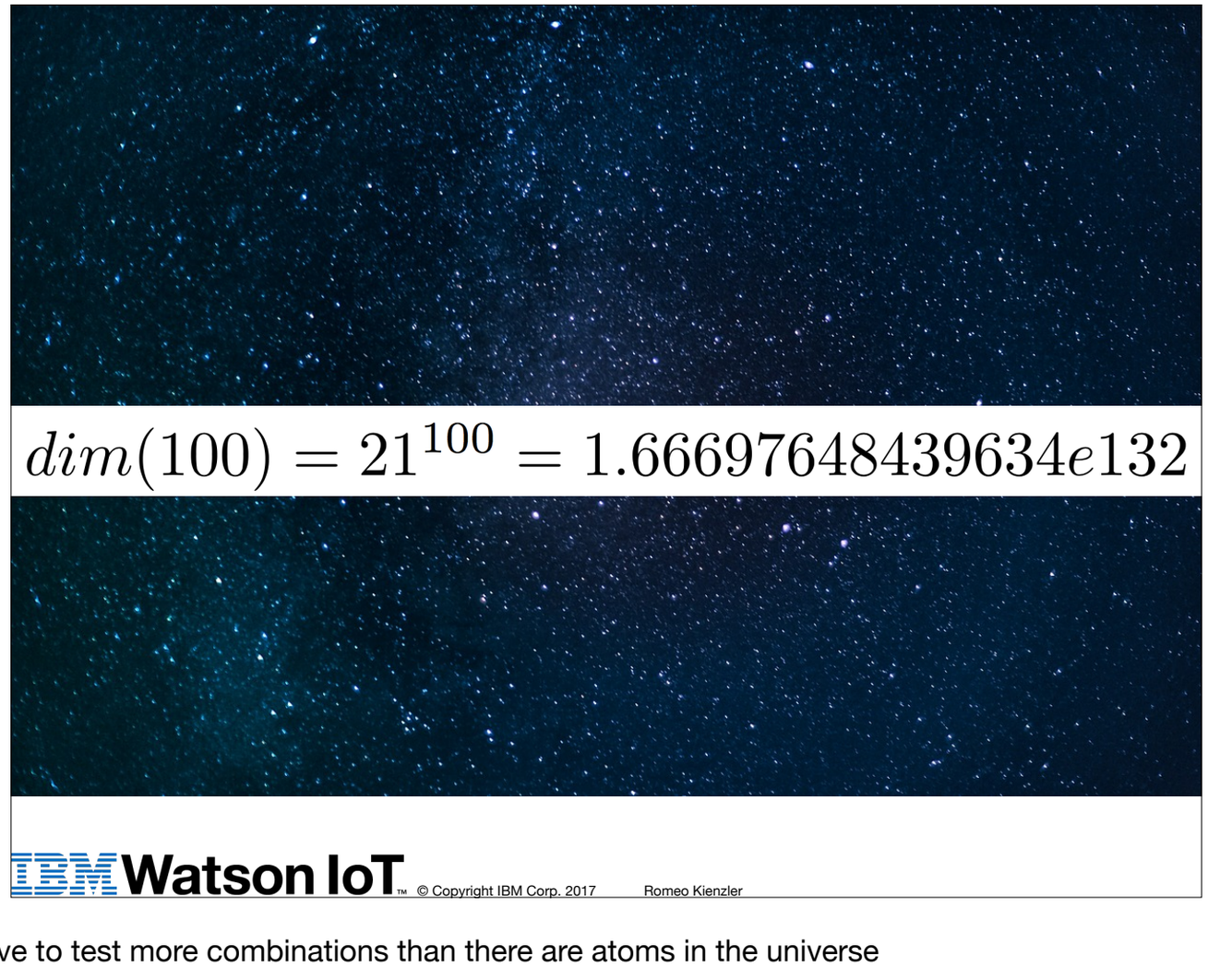
$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$
$dim(8) = 37822859361$
$dim(9) = 794280046581$
$dim(10) = 16679880978201$
$dim(11) = 350277500542221$
$dim(12) = 7355827511386641$
$dim(13) = 1.544723777391195e17$
$dim(14) = 3.243919932521508e18$

$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$
$dim(8) = 37822859361$
$dim(9) = 794280046581$
$dim(10) = 16679880978201$
$dim(11) = 350277500542221$
$dim(12) = 7355827511386641$
$dim(13) = 1.544723777391195e17$
$dim(14) = 3.243919932521508e18$
$dim(15) = 6.812231858295167e19$

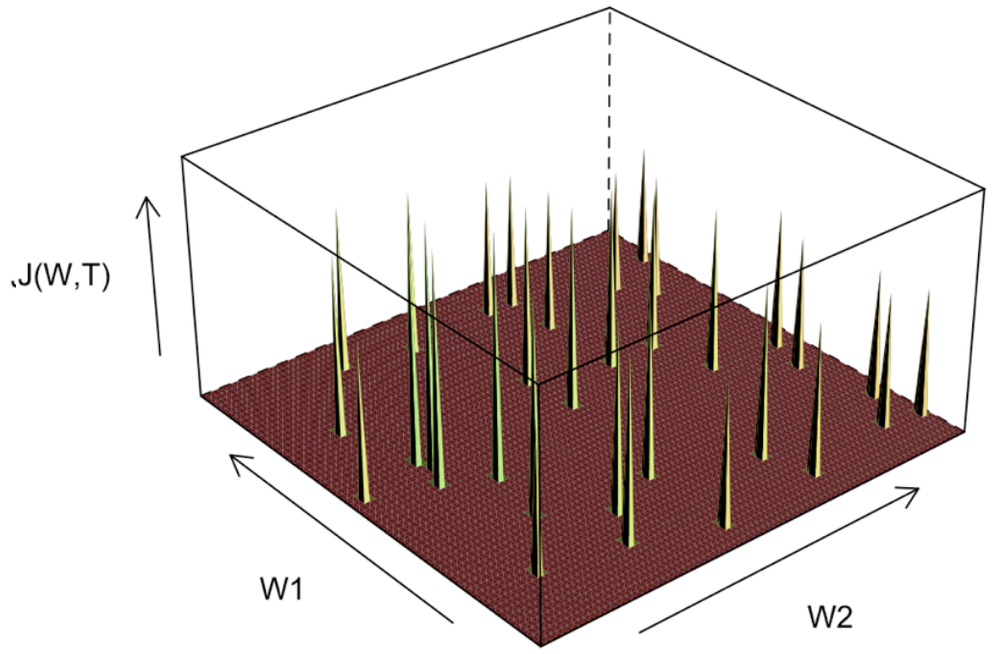
$v = [-1, 1]$
$grid = [-1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
$dim(1) = 21$
$dim(2) = 411$
$dim(3) = 9261$
$dim(4) = 194481$
$dim(5) = 4084101$
$dim(6) = 85766121$
$dim(7) = 1801088541$
$dim(8) = 37822859361$
$dim(9) = 794280046581$
$dim(10) = 16679880978201$
$dim(11) = 350277500542221$
$dim(12) = 7355827511386641$
$dim(13) = 1.544723777391195e17$
$dim(14) = 3.243919932521508e18$
$dim(15) = 6.812231858295167e19$
$dim(16) = 1.430568690241985e21$



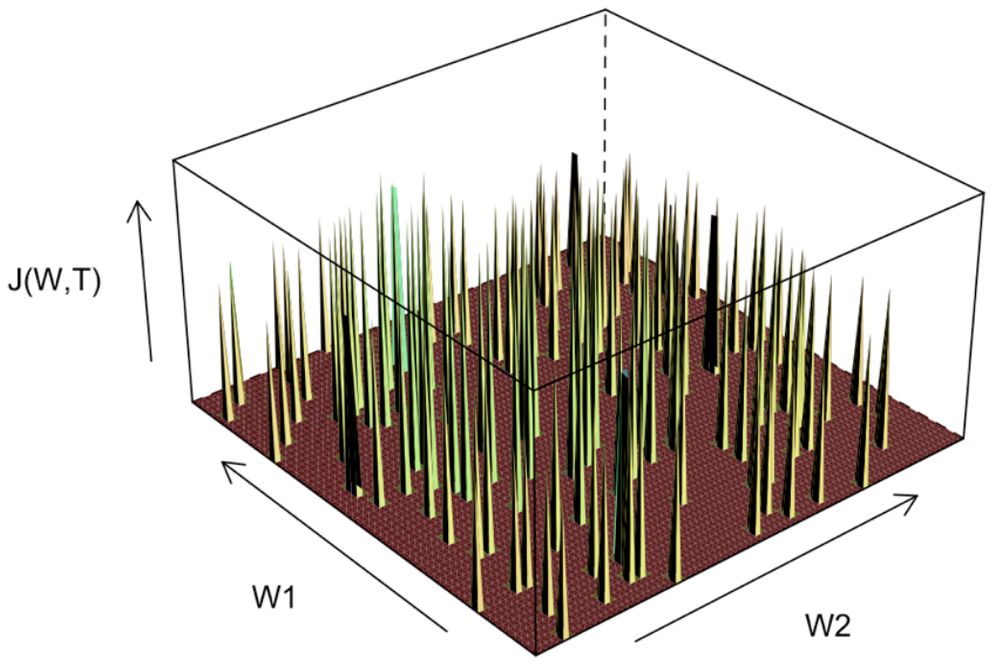
This number grows exponentially….



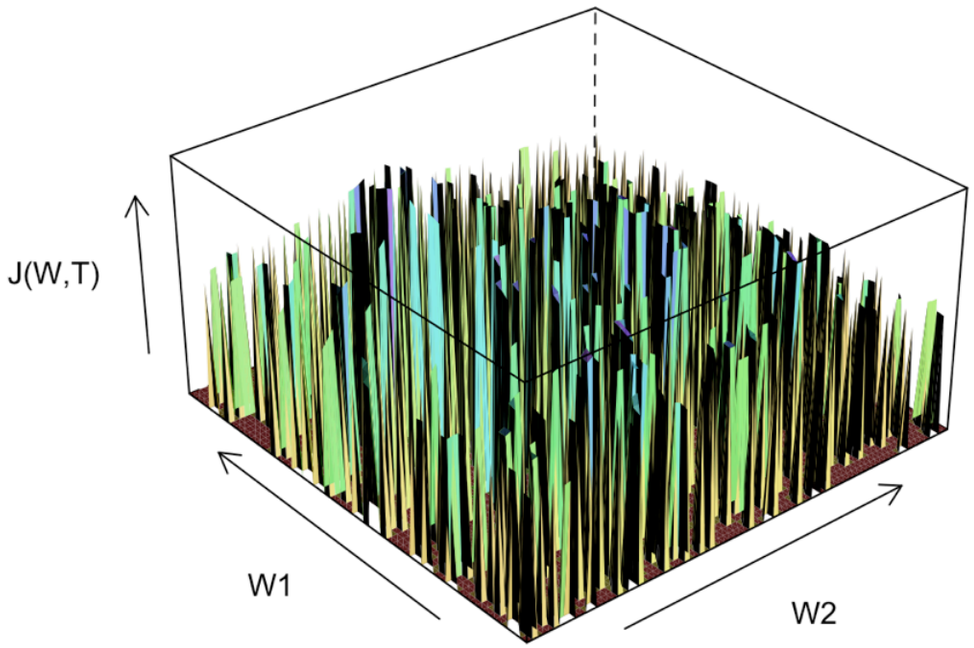
With only a hundred dimensions you have to test more combinations than there are atoms in the universe



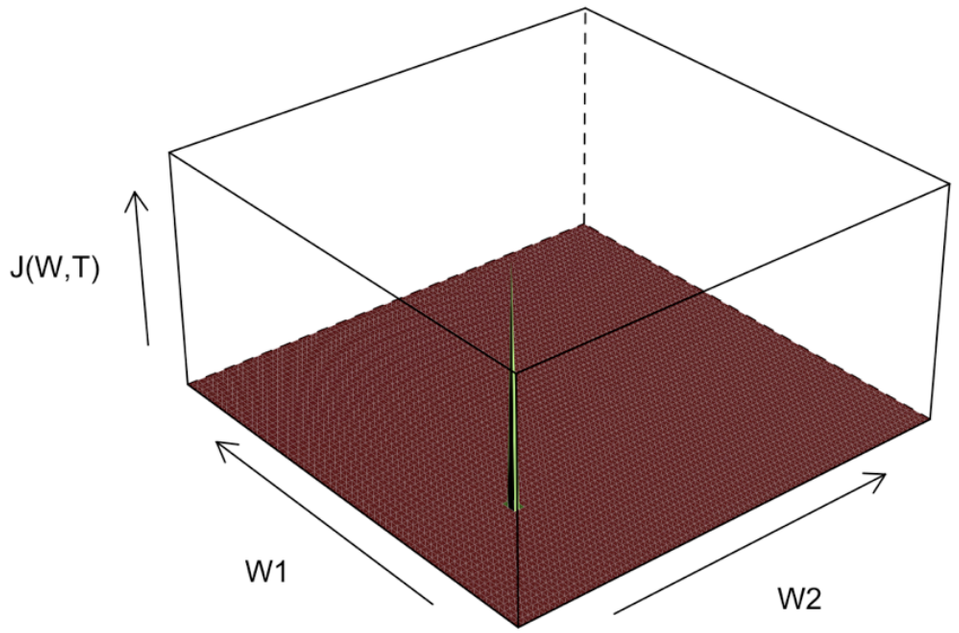
So this obviously is not a good way for neural network training. So what if we not strictly follow a grid and just randomly select value combinations for the weights?



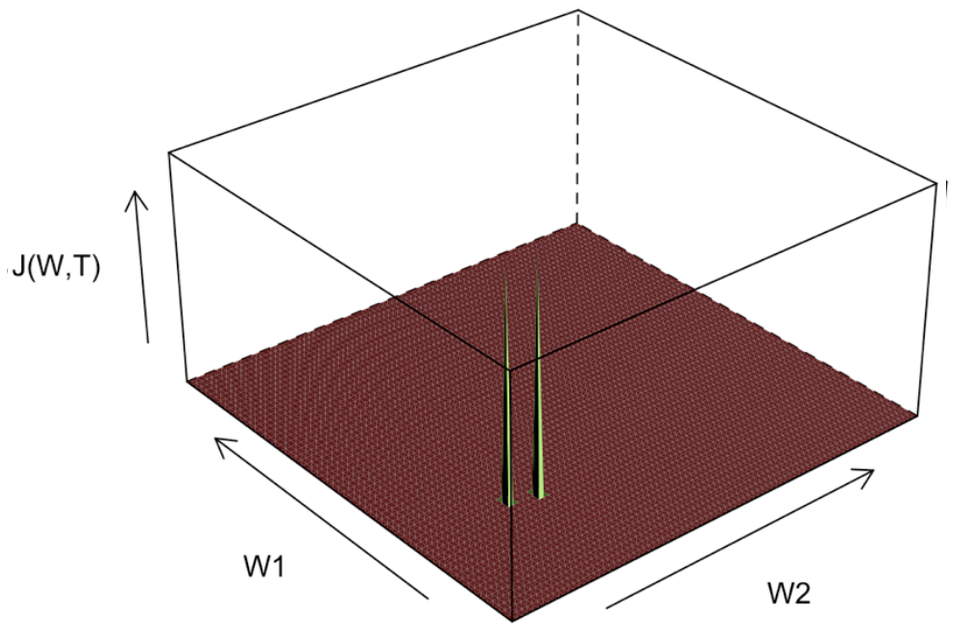
So it turns out that this method - with is called monte carlo - outperforms grid search



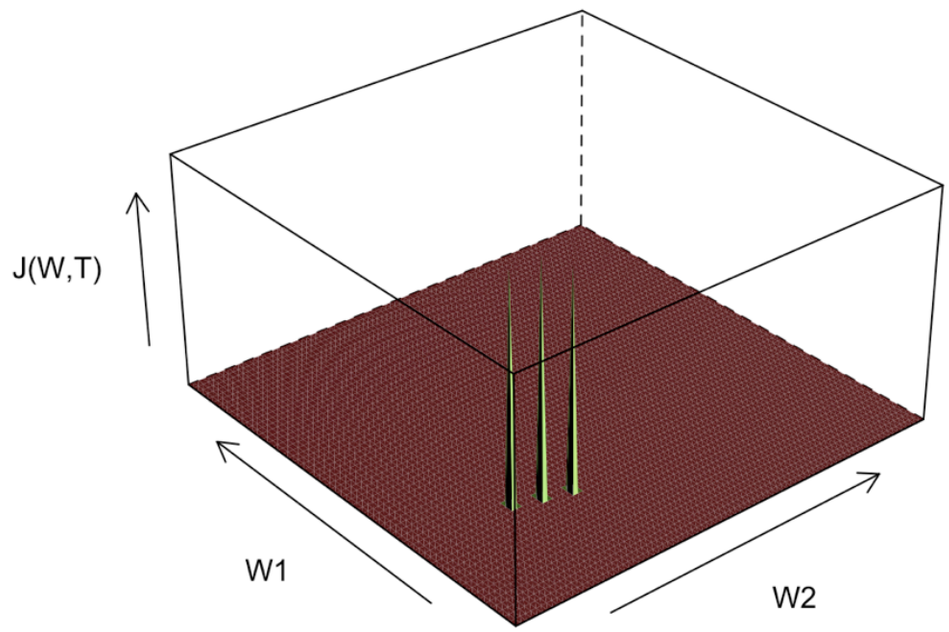
But still - in order to get a good approximation for the cost function - we need to test a lot of values which is not feasible for neural network training



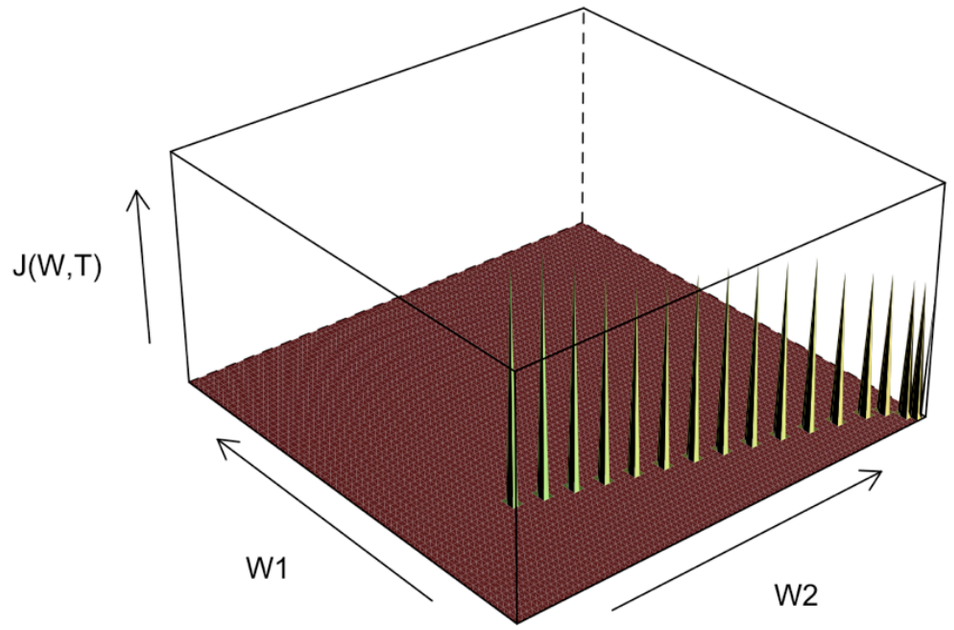
So what if we had a possibility - when evaluating a single value - to know in which direction we should go in order to test for the next value?



We then could climb down the ladder



..one step by another…



..until we reached an optimal value - this method is called gradient descent - and with all it’s variations it’s the de-facto standard in neural network training.

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$

 Romeo Kienzler

So how does this work? Let’s have a look at our cost function J - in this case the so-called quadratic cost - which basically sums the square of the difference between the prediction of the neural network and the actual, real value. This difference is called error. Therefore this is also called the sum of squared errors. There exists other cost functions as well like …

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$

cross entropy

cross entropy

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$

cross entropy exponential cost

exponential cost

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$

cross entropy
exponential cost
hellinger distance

hellinger distance and many more. Choosing an appropriate cost function depends on your data and neural network topology, is part of the so called hyper or meta parameter space and ideal selection is often considered as black magic. Or in other words, trial and error.

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$
$$\hat{y} = a_3 = f(z_3)$$

So if we now have a look how y hat is calculated we notice that it is dependent on a activation function f and on z three

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$
$$\hat{y} = a_3 = f(z_3)$$
$$\hat{y} = a_3 = f(a_2 W_2)$$

zee three is calculated by the activation of the previous layer multiplied by the weights of the actual layer

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$
$$\hat{y} = a_3 = f(z_3)$$
$$\hat{y} = a_3 = f(a_2 W_2)$$
$$\hat{y} = a_3 = f(f(z_2)W_2)$$

If we replace a two by it's definition we obtain the following form

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$
$$\hat{y} = a_3 = f(z_3)$$
$$\hat{y} = a_3 = f(a_2 W_2)$$
$$\hat{y} = a_3 = f(f(z_2)W_2)$$
$$\hat{y} = a_3 = f(f(XW_1)W_2)$$

and if we replace zee two it’s x multiplied by the weights of layer one we obtain this form

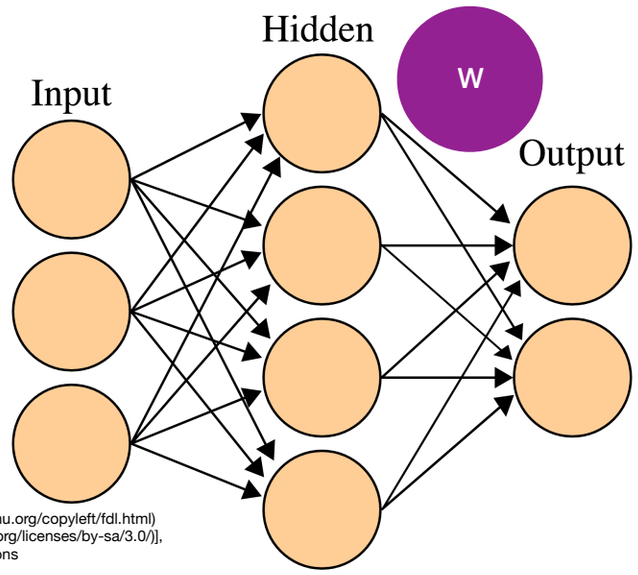
$$J = \sum \frac{1}{2}(y - \hat{y})^2$$
$$\hat{y} = a_3 = f(z_3)$$
$$\hat{y} = a_3 = f(a_2 W_2)$$
$$\hat{y} = a_3 = f(f(z_2) W_2)$$
$$\hat{y} = a_3 = f(f(X W_1) W_2)$$
$$J = \sum \frac{1}{2}(y - f(f(X W_1) W_2))^2$$

Finally we put this form into J and we obtain a cost function J based on all individual computations of the forward pass of the neural network

$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$

So now we'll use the so-called back propagation method. This calculates the neural network computations backwards and it does so by back-propagating the error y minus y hat.

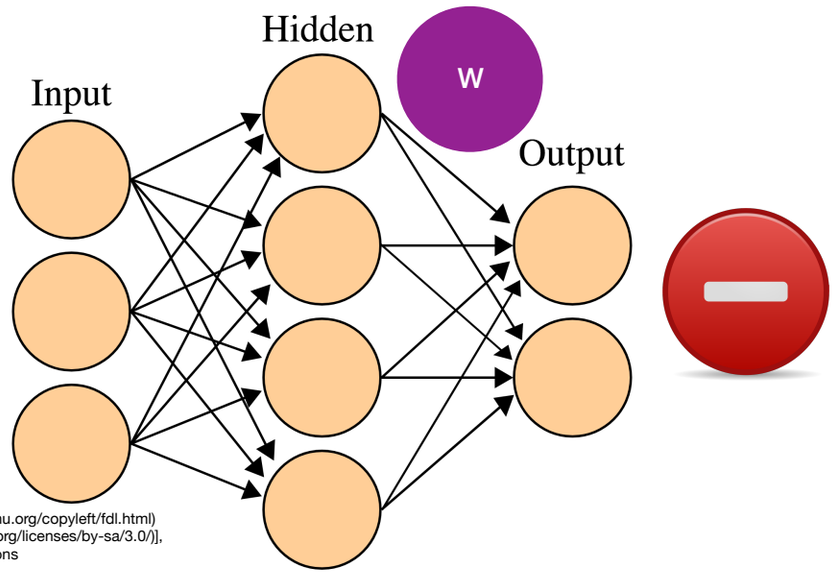
$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

Intuitively, this means that on the reverse computation of the neural network, big weight values end up in back propagating big amounts of the error the the upstream layers where as small weight values only let small amounts of the error being back propagated.

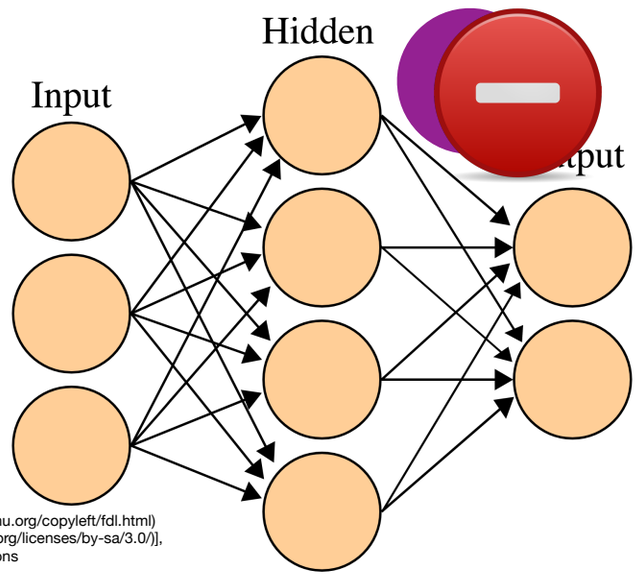
$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

So let’s assume on a given data row we end up with a certain amount of error indicated by the red circle.

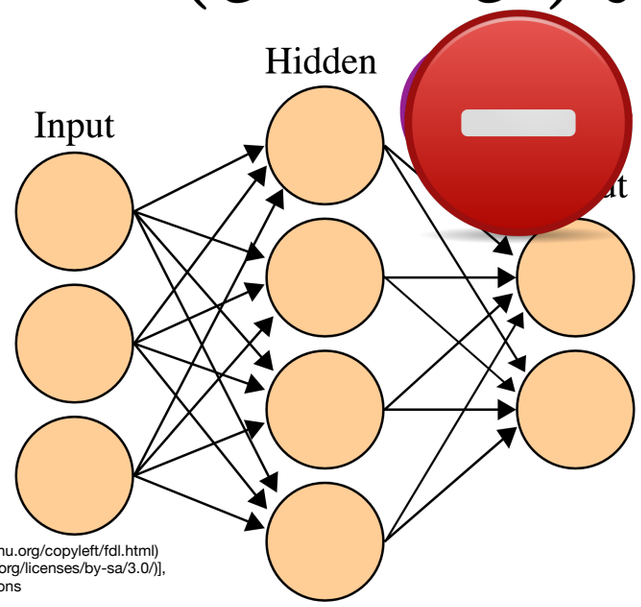
$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

So if we now back propagate this error we use the weight to determine how much of this error should be back propagated

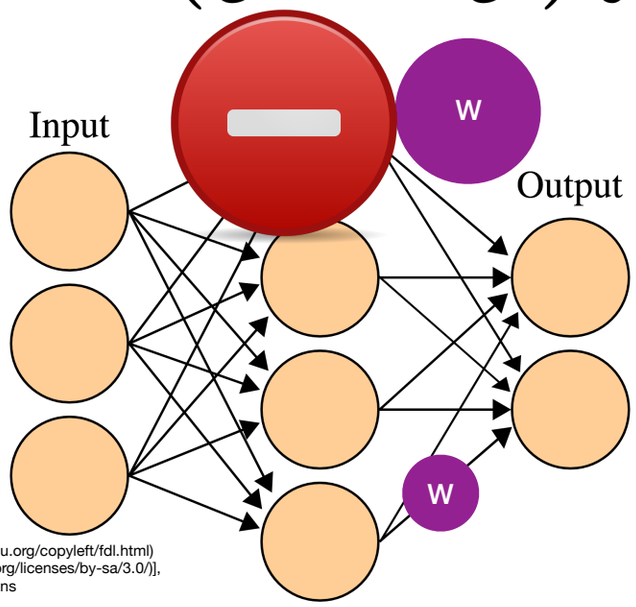
$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

Since the weight is high we also increase the error…

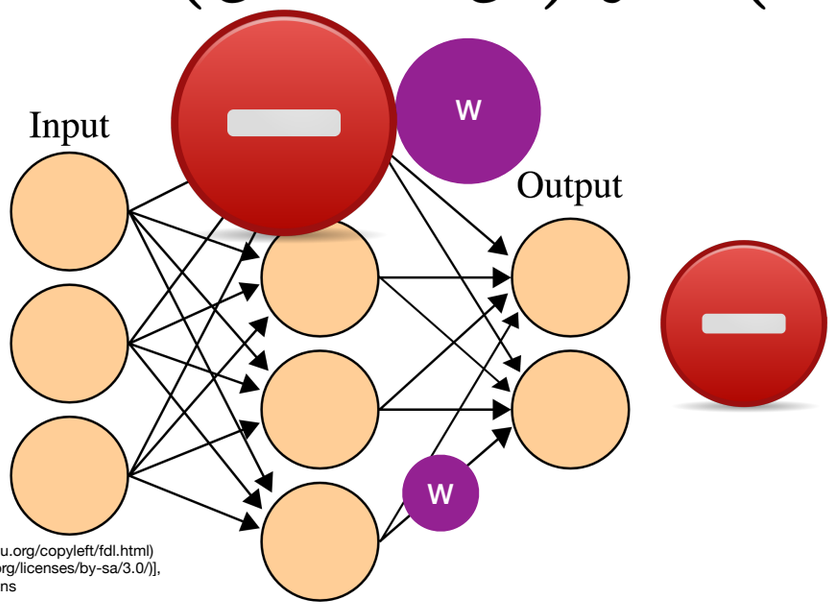
$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

…which then gets backpropagated

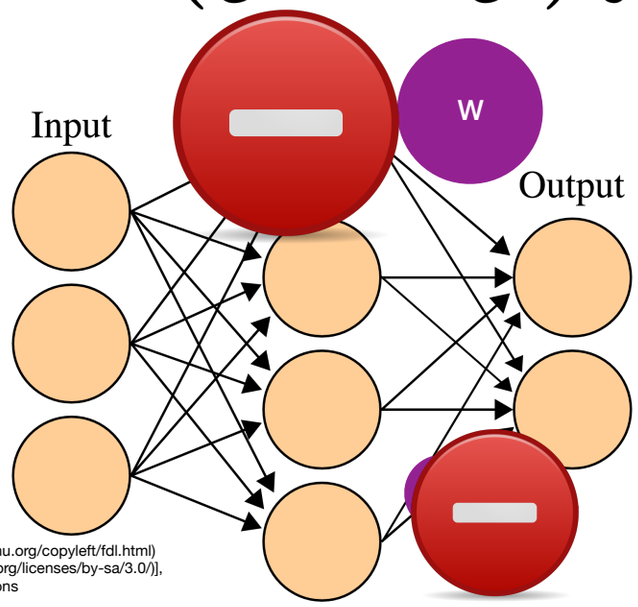
$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

Now if the weight is small

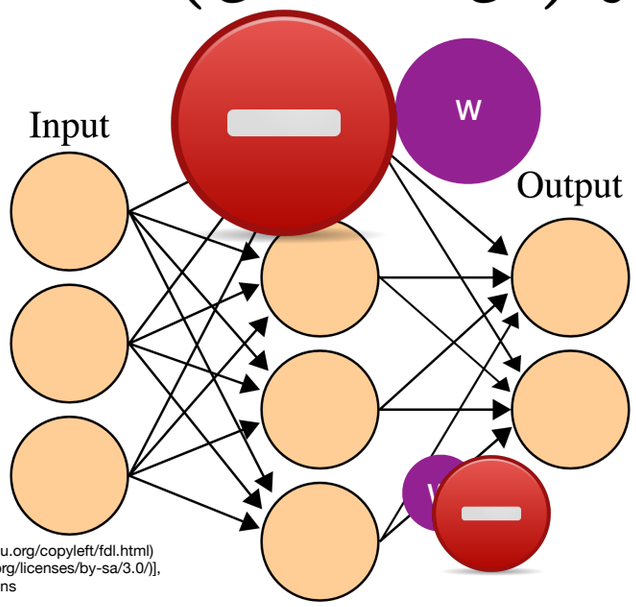
$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

..the error also gets reduced

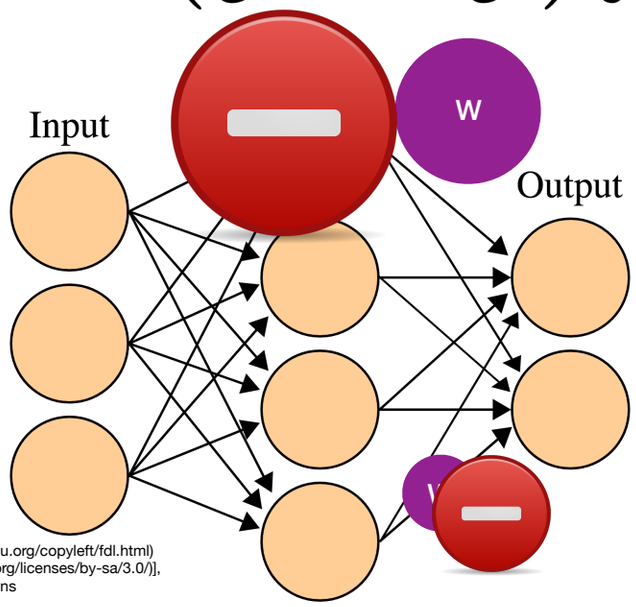
$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

and the amount of error for that particular backpropagation…

$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$



By en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons

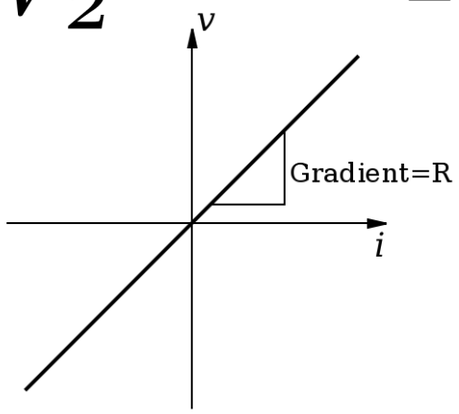
to the upstream layer neutron is less. This is the desired behaviour since big weights are contributing more to the overall error than small weights.

$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$
$$\frac{\partial J}{\partial W_2} = a_2^T \sigma_3$$

So this is nice but what we actually want is not the error or cost on each neuron but

$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$

$$\frac{\partial J}{\partial W_2} = a_2^T \sigma_3$$



By inductiveload (From scratch in Inkscape) [Public domain], via Wikimedia Commons

the gradient of each error. If we know the gradient we know in which direction we have to update each weight on order to go downhill the cost hyper surface. Therefore we compute the partial derivative with respect to double u two first. I'll skip the linear algebra and calculus here but it turns out that the derivative is a two transpose times delta three, which is based on the reverse computation which is also called back propagation. So note that f prime is the first derivative of the activation function f.

$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$

$$\frac{\partial J}{\partial W_2} = a_2^T \sigma_3$$

$$\sigma_2 = \sigma_3 W_2^T f\prime(z_2)$$

Now we continue in back propagating the error and taking the first derivative of that function in parallel to obtain delta two

$$\sigma_3 = -(y - \hat{y}) f\prime(z_3)$$

$$\frac{\partial J}{\partial W_2} = a_2^T \sigma_3$$

$$\sigma_2 = \sigma_3 W_2^T f\prime(z_2)$$

$$\frac{\partial J}{\partial W_1} = X^T \sigma_2$$

And finally we get the partial derivative with respect to double u one. And again the mathematical reason why this is computed using the transpose of X times delta two is beyond the scope of this course and due to the chain rule in calculus.

Summary

So as you have seen, training of neural networks can get quite complicated. But it’s ok if you didn’t understand all the details. Just note that gradient descent doesn’t make any guarantees that it converges to the global optimum unless the cost function is convex, which is not the case in neural networks. So it might get stuck in local optimas and even guarantees to converge at all. Therefore a lot of hyper parameter tuning might be necessary.

Introduction to TensorFlow

 Romeo Kienzler

In the next module we will cover TensorFlow, one of the hottest DeepLearning frameworks out now.