Deep Neural Network from scratch

Florian Courtial

18-04-2017

Chapter 1

Introduction

In this post we will learn how a deep neural network works, then implement one in Python, then using TensorFlow. As a toy example, we will try to predict the price of a car using the following features: number of kilometers travelled, its age and its type of fuel. We will predict the price of only one model of car because our features does not have data about the brand or the model of the car. Let's say BMW Serie 1. Our data will come from leboncoin.fr.

Our model will certainly not work very well because we are missing important car attributes impacting the price, but the goal is to use real data while keeping things simple.

Chapter 2

Data Preprocessing

The first step is to normalize our data (known as feature scaling). We have:

Number of kilometers: quantitative, number between 0 and 350k. Type of fuel: binary data diesel/gasoline. Age: quantitative, number between 0 and 40. Price: quantitative, number between 0 and 40k. The number of kilometers and age (both quantitative) will be normalized using the means and standard deviation. The goal is to bring all the data to the same scale, generally between $[-6,6]$. Else the number of kilometers will have a value up to 350k and the age up to 40 years, and a change in a weight value will not impact the age or number of kilometers in the same way. The formula for the normalization is: $x' = \frac{x-\bar{x}}{\sigma}$ Where x is the original feature dataset (all the cars), $\bar{x}$ is the mean of that feature dataset, and σ is the standard deviation of that feature dataset. We will do it in python later.

The type of fuel (binary) will be encoded with -1 for one value and 1 for the other. There is no categorical data here in order to keep the number of features light because each categorical feature would become a sparse matrix of the size of the number of classes. If you have categorical data in your features (i.e: red, green, pink and blue for a feature) you must use effect encoding or dummy encoding. Our number of features will not change, only the values after the encoding.

We will normalize the price so that all the values are between $[0,1]$. This is necessary because our neural network will produce results between $[0,1]$. If the neural network guesses a price of 0.45 for a car whereas our initial price is 17k, the error between the guess and the real price will be high, 16 999.55€. As the guess will be at most 1, the error will be at best 16 999€ and will never improve. We will normalize the price using the formula:

$$\frac{x_i - \min(x)}{\max(x) - \min(x)}$$

x_i is one car price whereas x is the whole price dataset. It means that each car price will be processed using the maximum available price and the minimum available price.

Our 17K€ price will become something like 0.43 and if our network predict 0.45 it will be a good prediction. We will do the exact opposite from the above formula to find the price back from 0.45.

Chapter 3

Forward Propagation

So we have three inputs (features), one binary, two quantitatives and one quantitative output. As we will predict a quantitative variable and we will use past data to train our network, it is called a supervised regression problem.

One input is a car.

Number of kms	Fuel	Age	Price
38000	Gasoline	3	17 000

Once the data has been normalized and encoded as previously described, we will have.

Number of kms	Fuel	Age	Price
1.4	-1	0.4	0.45

As we will have more than one car, our array will have more lines.

Number of kms	Fuel	Age	Price
1.4	-1	0.4	0.45
0.4	-1	0.1	0.52
5.4	-1	4	0.25
1.5	-1	1	0.31
...	..	...	...

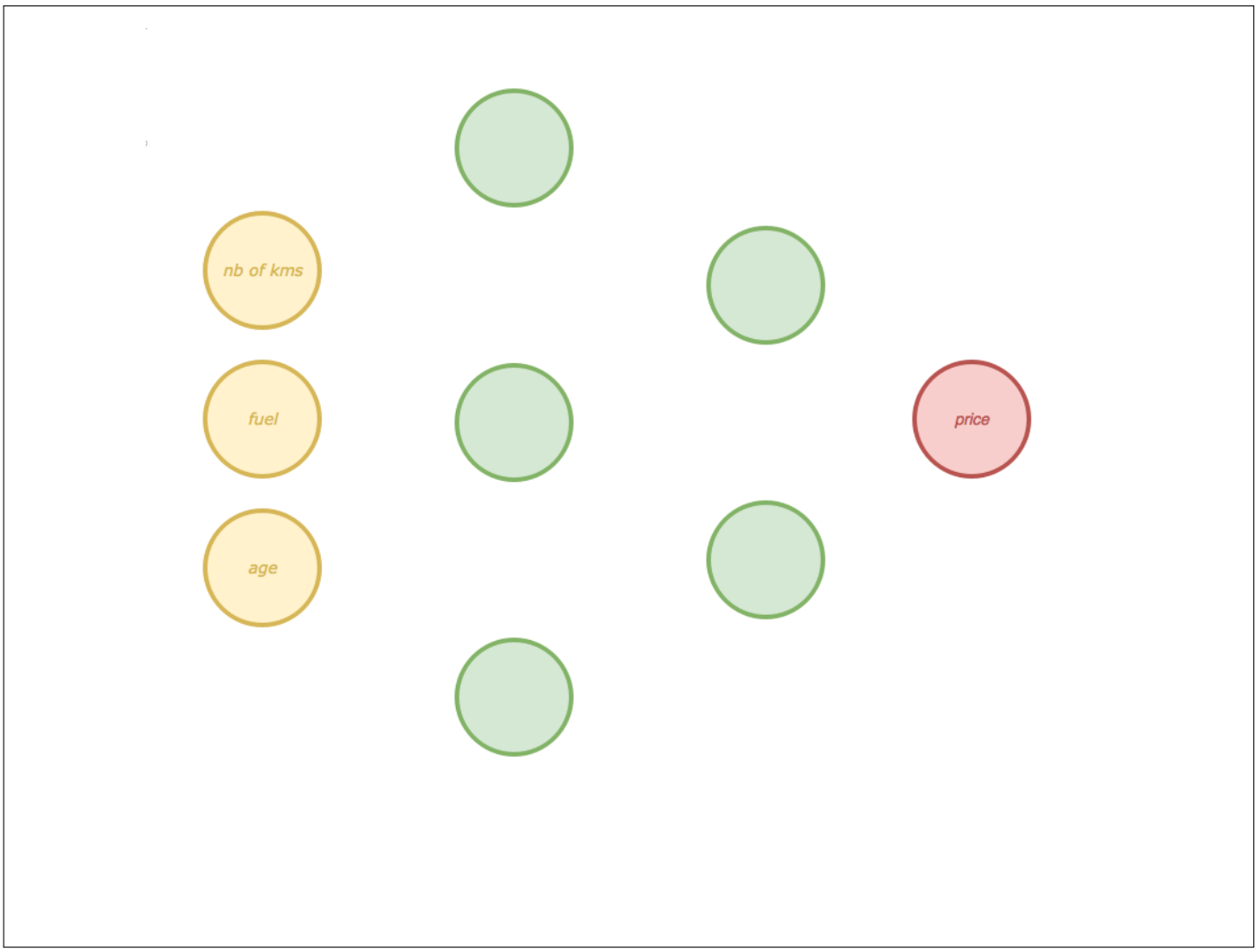
We can see this array as matrix.

$$X = \begin{bmatrix} 1.4 & -1 & 0.4 & 0.45 \\ 0.4 & -1 & 0.1 & 0.52 \\ 5.4 & -1 & 4 & 0.25 \\ 1.5 & -1 & 1 & 0.31 \\ ... & ... & ... & ... \end{bmatrix}$$

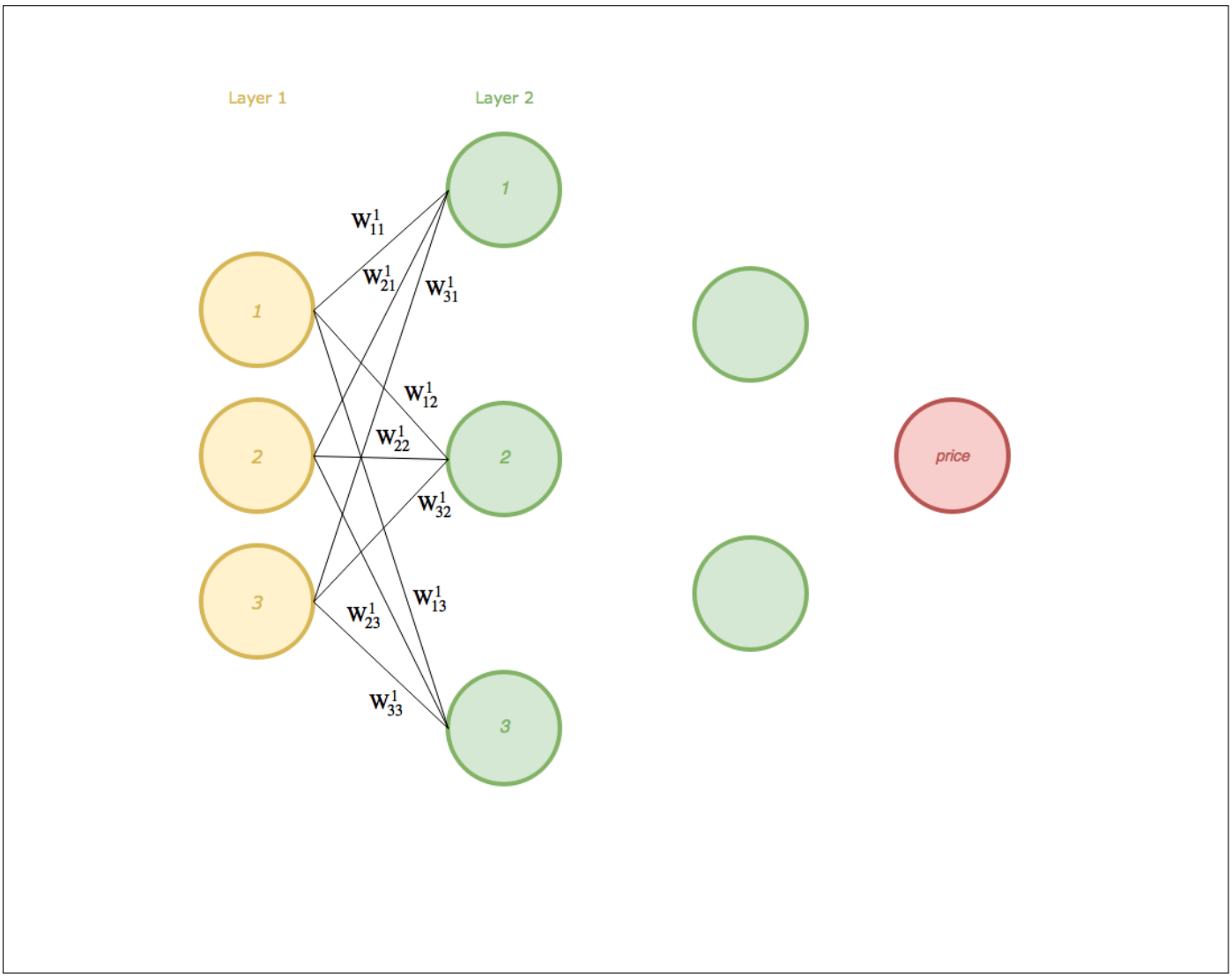
The price is not a feature and will not be available for a new car. We separate inputs (features) and outputs (prediction).

$$X = \begin{bmatrix} 1.4 & -1 & 0.4 \\ 0.4 & -1 & 0.1 \\ 5.4 & -1 & 4 \\ 1.5 & -1 & 1 \\ ... & ... & ... \end{bmatrix} \quad y = \begin{bmatrix} 0.45 \\ 0.52 \\ 0.25 \\ 0.31 \\ ... \end{bmatrix}$$

For now we have two matrices X and y, it's time to complete our network. We choose to have two hidden layers between our inputs and our output, the first layer will have three neurons and the second one two. The number of hidden layers and the number of neurons by layer is up to you. These are two hyper parameters that you have to define before running your neural network.



Each of our input neuron will reach all the neurons in the next layer because we are using a fully connected network. Each link from one neuron to another is called a synapse and comes with a weight. A weight on a synapse is of the form W_{jk}^{l} where l denotes the number of the layer, j the number of the neuron from the l^{th} layer and k the number of the neuron from the $(l+1)^{th}$ layer.



We can view the weights as a matrix.

$$W^1 = \begin{bmatrix} W_{11}^1 & W_{12}^1 & W_{13}^1 \\ W_{21}^1 & W_{22}^1 & W_{23}^1 \\ W_{31}^1 & W_{32}^1 & W_{33}^1 \end{bmatrix}$$

The number of rows equals the number of features (inputs) and the number of columns equals the number of neurons in the layer number 2. Here we have three features and the first layer has three neurons so our W^1 matrix is of the size 3×3. For now I don't talk about the bias unit to keep things simple.

Now we compute the values of the neurons of the first hidden layer (the first column of green circles). For each car defined by three features, we will have three neurons computed for the first hidden layer. We do that using matrix calculus.

Let's say that we have only one car in our dataset. We don't need y for now.

$$X = \begin{bmatrix} 1.4 & -1 & 0.4 \end{bmatrix}$$

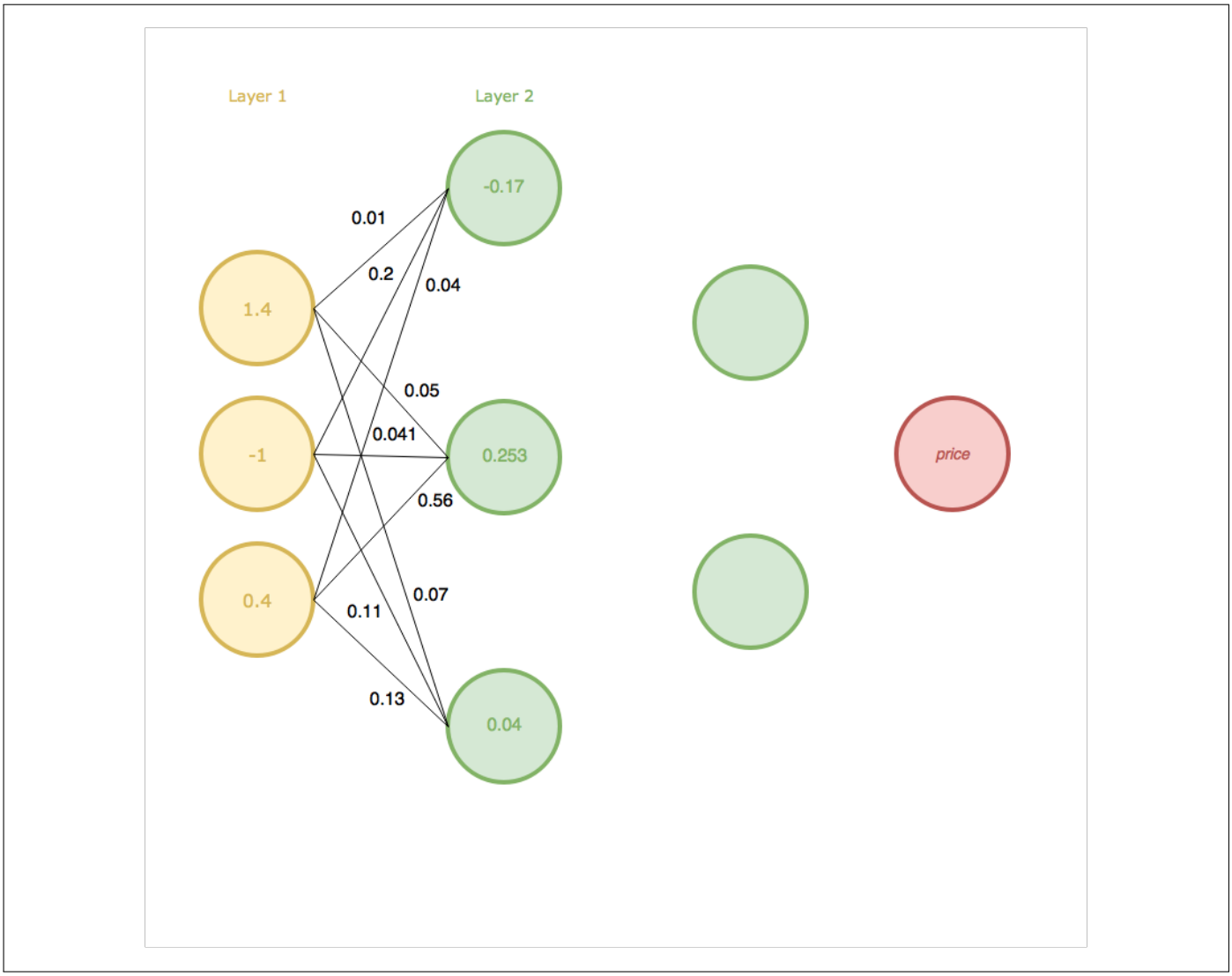
Our W^1 matrix is randomly initialized the first time so we can use random values for the weights.

$$W^1 = \begin{bmatrix} 0.01 & 0.05 & 0.07 \\ 0.20 & 0.041 & 0.11 \\ 0.04 & 0.56 & 0.13 \end{bmatrix}$$

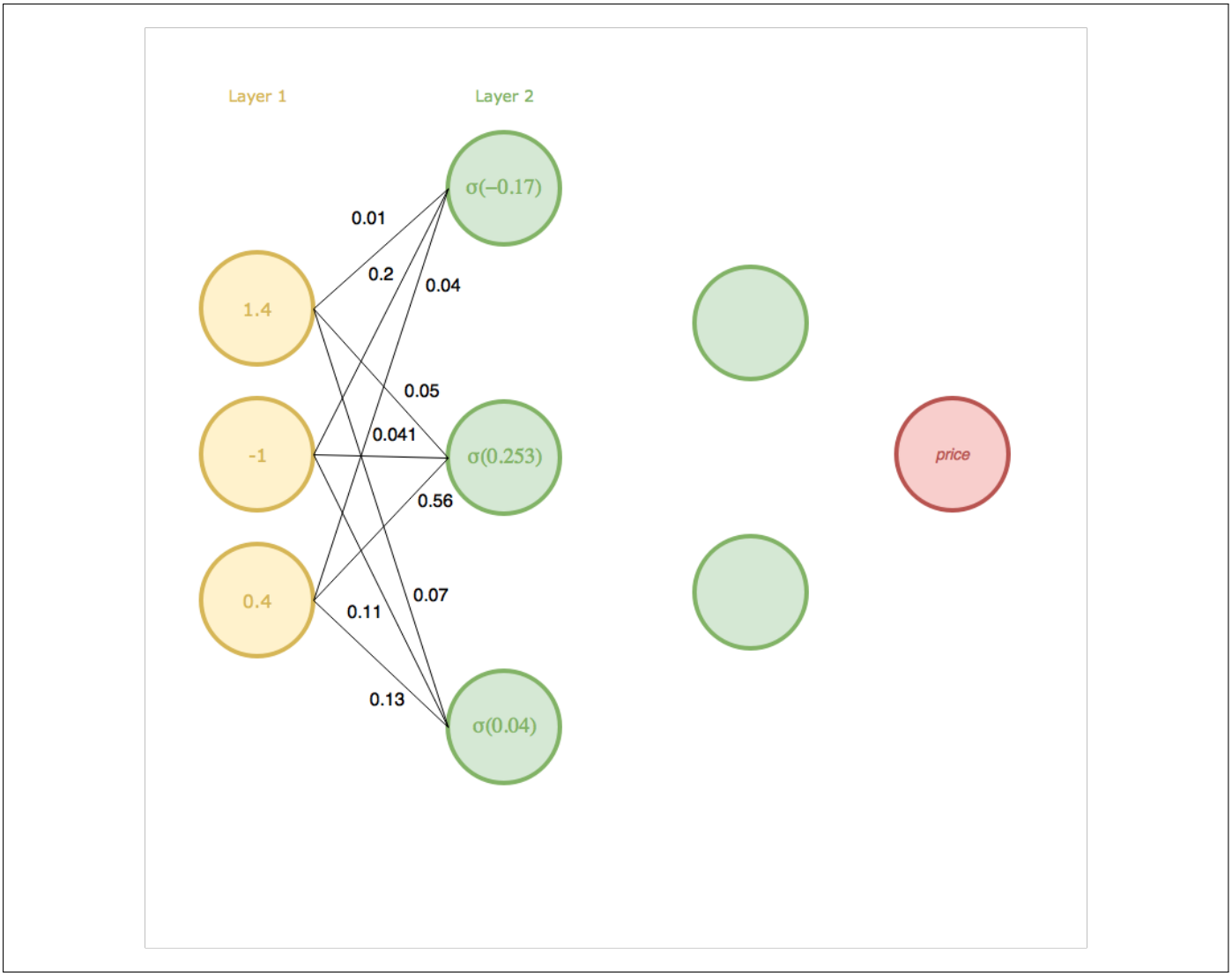
The values of the neurons of the first hidden layer are called the activities of the first hidden layer. They will be saved into a matrix called $a^{(2)}$. We begin by computing $Z^{(2)}$ using the formula: $Z^{(2)} = X.W^1$

$$Z^{(2)} = \begin{bmatrix} 1.4 & -1 & 0.4 \end{bmatrix} . \begin{bmatrix} 0.01 & 0.05 & 0.07 \\ 0.20 & 0.041 & 0.11 \\ 0.04 & 0.56 & 0.13 \end{bmatrix} \quad Z^{(2)} = \begin{bmatrix} -0.17 & 0.253 & 0.04 \end{bmatrix}$$

We can view it on our network.



We are almost done with the first hidden layer, we just have to apply an activation function on $Z^{(2)}$. We can view it like that.



Once we apply an activation function element wise to $Z^{(2)}$ we obtain $a^{(2)}$, the activities of our second layer.

$a^{(2)} = \sigma(Z^{(2)})$

Where $\sigma(x)$ is our activation function. The activation function is applied element wise and is non linear, allowing the network to compute complicated problems using only a small number of nodes. The common ones are Sigmoid, Tanh, ReLU, Leaky ReLU, Maxout. The list is actually bigger than that. Which one to choose? Andrej Karpathy says the following:

TLDR: "What neuron type should I use?" Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of "dead" units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

So we should use ReLU as an activation function, it is short for rectified linear unit and is actually quite simple. If x is greater than 0 we take it, else we take 0. $\sigma(x) = \max(0, x)$ Nonetheless we have very little neurons and one or two dying neuron will undermine the results of our network (when the value of the neuron becomes zero). Instead we will use the Tanh as an activation function.

$\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

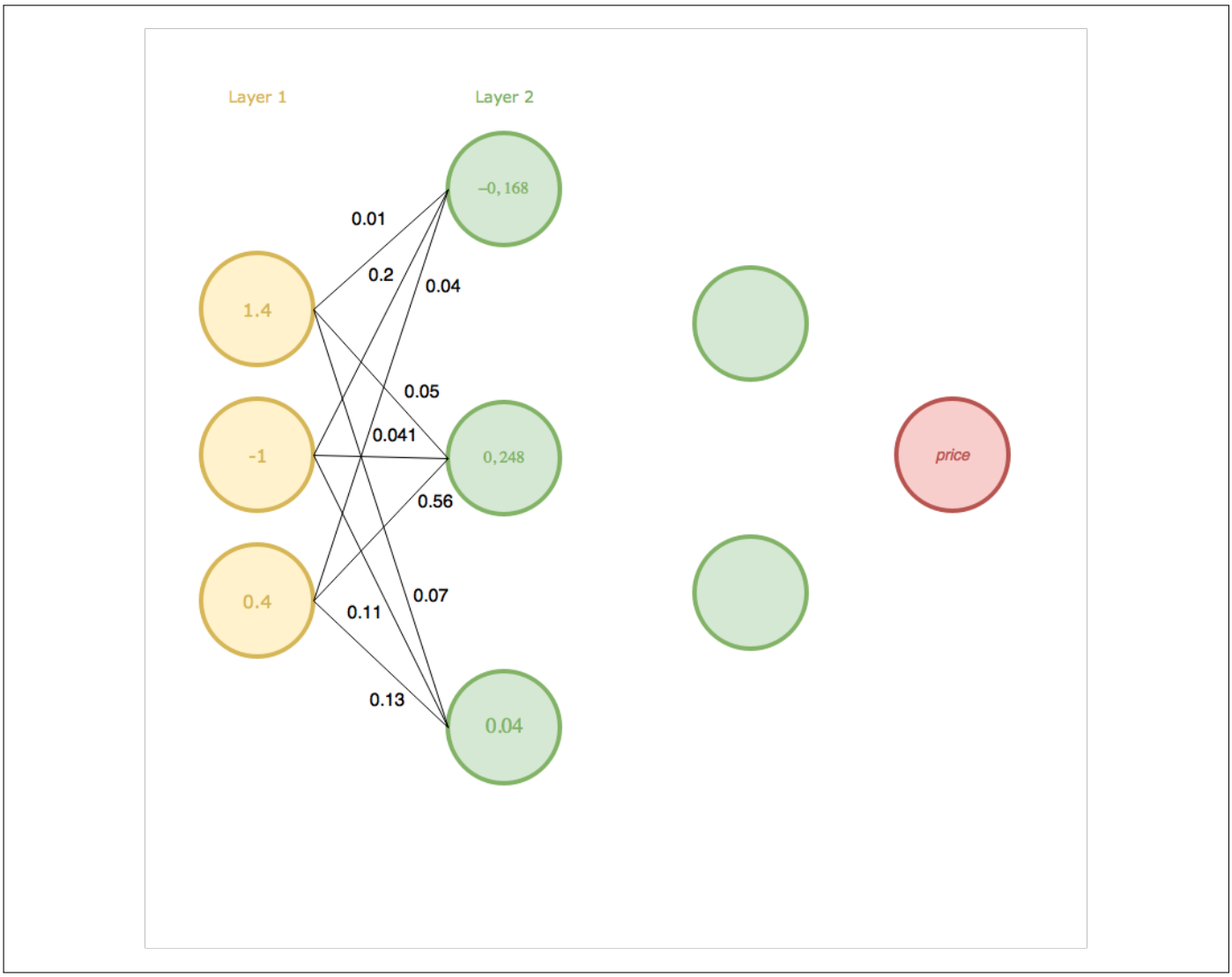
So the first neuron of our first hidden layer will become:

$\sigma(-0.17) = tanh(-0.17) = \frac{e^{-0.17} - e^{0.17}}{e^{-0.17} + e^{0.17}} = -0.168$

The second one:

$$\sigma(0.253) = tanh(0.253) = \frac{e^{0.253} - e^{-0.253}}{e^{0.253} + e^{-0.253}} = 0,248$$

So our first hidden layer with only one car looks like:



Thanks to the matrix calculus property, each neuron of the first hidden layer will receive a weighted sum of the inputs. We have:

$$Z^{(2)} = \begin{bmatrix} 1.4 & -1 & 0.4 \end{bmatrix} . \begin{bmatrix} 0.01 & 0.05 & 0.07 \\ 0.20 & 0.041 & 0.11 \\ 0.04 & 0.56 & 0.13 \end{bmatrix}$$

$$Z^{(2)} = \begin{bmatrix} -0.17 & 0.253 & 0.04 \end{bmatrix}$$

$Z_1^{(2)}$ the first neuron of the layer two, for which we found -0.17 is the result of $1.4 \times 0.01 + -1 \times 0.2 + 0.4 \times 0.04$ (cf Matrix Multiplication). If you compare this to the neural network drawing, you see that in fact the first neuron of the layer two is the input 1 (number of kms) times the weight on the synapse plus the input 2 (type of fuel) times the weight on the synapse plus the input 3 (age) times the weight on the synapse. It is exactly what matrix calculus does for us.

For our example we assumed that only one car was in our dataset, that's why $X = \begin{bmatrix} 1.4 & -1 & 0.4 \end{bmatrix}$. In reality we will have several cars, let's say five. So our inputs are:

$$X = \begin{bmatrix} 1.4 & -1 & 0.4 \\ 0.4 & -1 & 0.1 \\ 5.4 & -1 & 4 \\ 1.5 & -1 & 1 \\ 1.8 & 1 & 1 \end{bmatrix}$$

And our $Z^{(2)}$ calculation will be:

$$Z^{(2)} = \begin{bmatrix} 1.4 & -1 & 0.4 \\ 0.4 & -1 & 0.1 \\ 5.4 & -1 & 4 \\ 1.5 & -1 & 1 \\ 1.8 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0.01 & 0.05 & 0.07 \\ 0.20 & 0.041 & 0.11 \\ 0.04 & 0.56 & 0.13 \end{bmatrix}$$

$$Z^{(2)} = \begin{bmatrix} -0.17 & 0.253 & 0.04 \\ -0.192 & 0.035 & -0.069 \\ 0.014 & 2.469 & 0.788 \\ -0.145 & 0.594 & 0.125 \\ 0.258 & 0.691 & 0.366 \end{bmatrix}$$

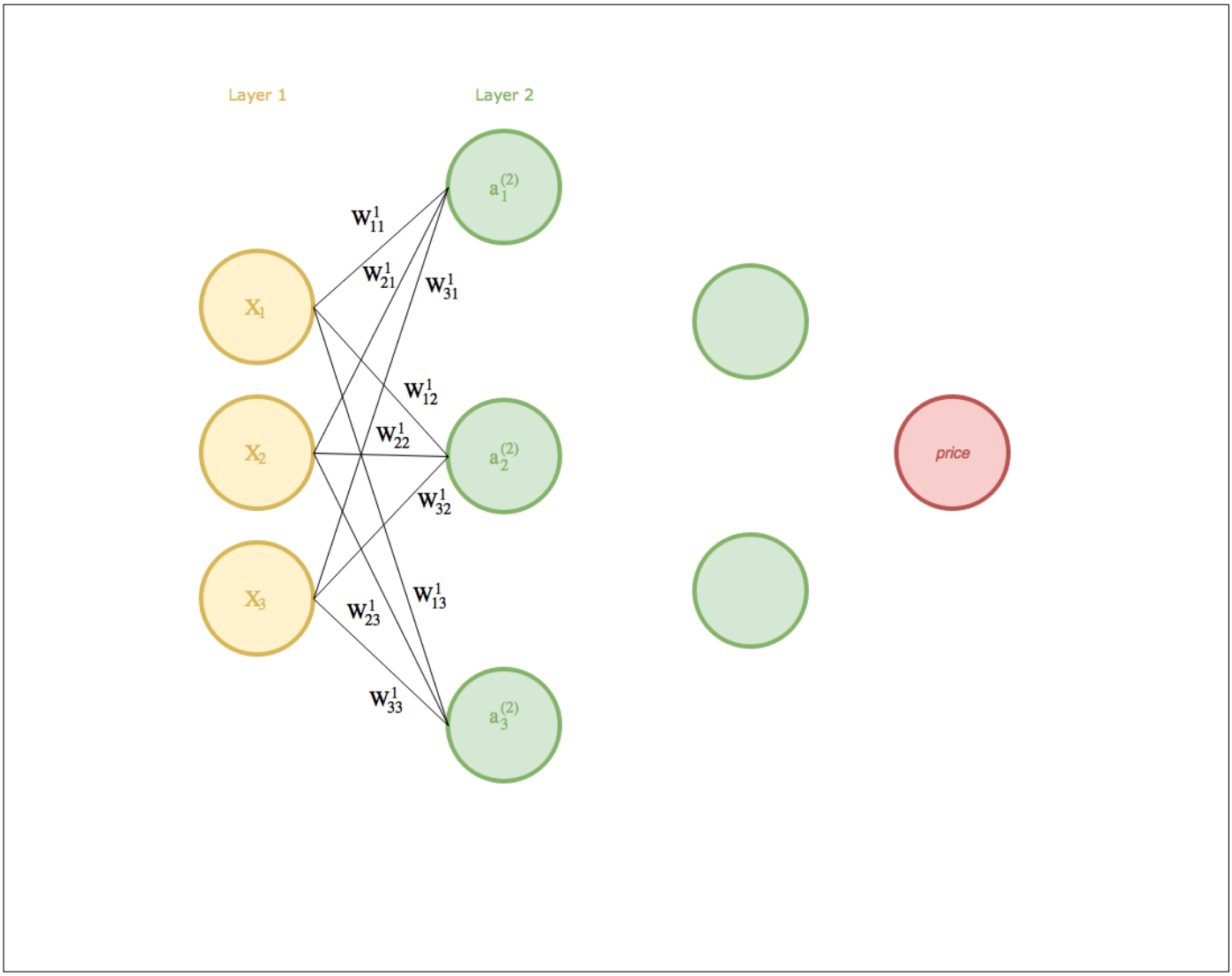
With the tanh applied element wise:

$$a^{(2)} = \sigma(Z^{(2)}) = tanh(Z^{(2)}) = \begin{bmatrix} tanh(-0.17) & tanh(0.253) & tanh(0.04) \\ tanh(-0.192) & tanh(0.035) & tanh(-0.069) \\ tanh(0.014) & tanh(2.469) & tanh(0.788) \\ tanh(-0.145) & tanh(0.594) & tanh(0.125) \\ tanh(0.258) & tanh(0.691) & tanh(0.366) \end{bmatrix}$$

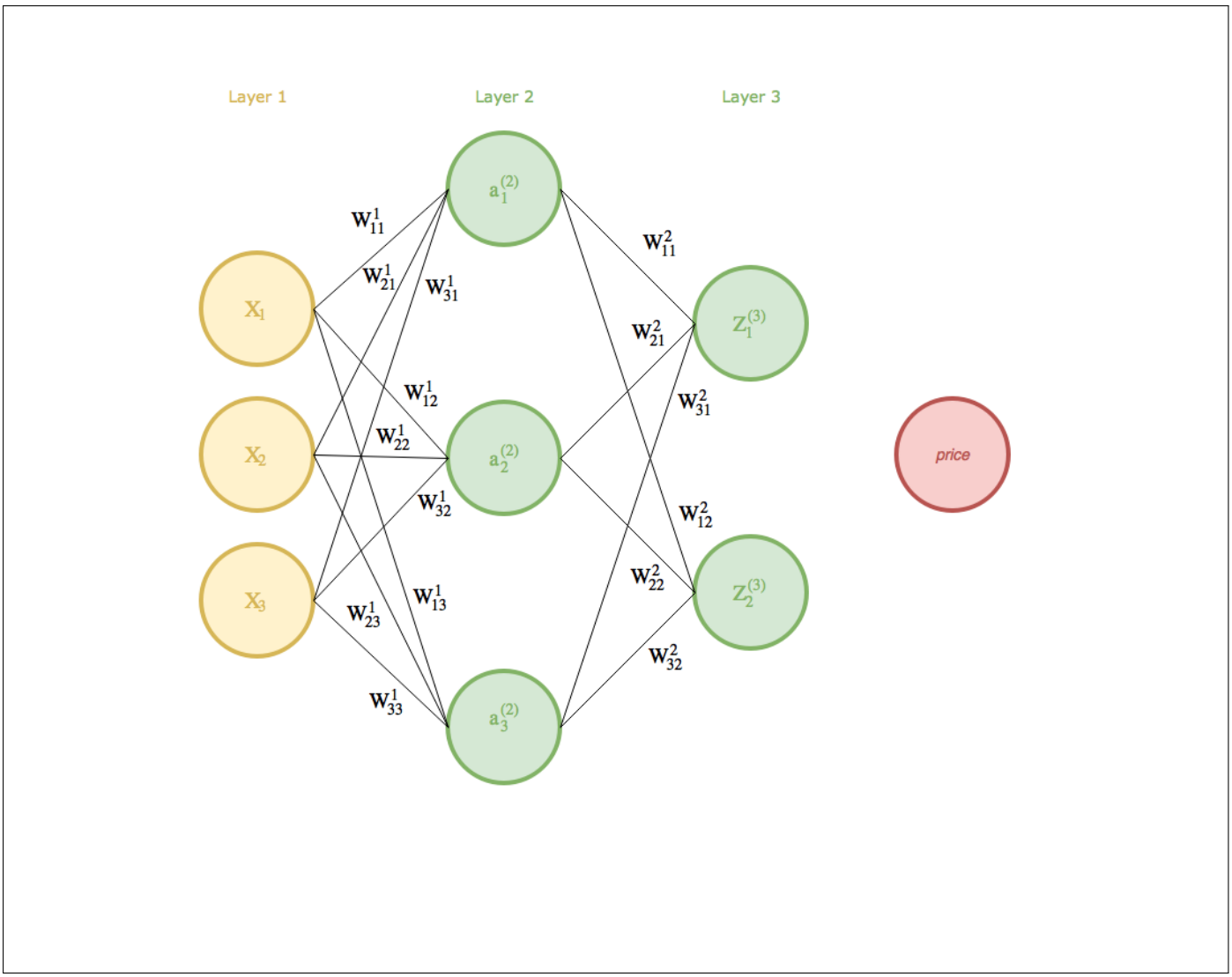
$$a^{(2)} = \begin{bmatrix} -0.16838105 & 0.24773663 & 0.03997868 \\ -0.18967498 & 0.03498572 & -0.06889071 \\ 0.01399909 & 0.98576421 & 0.65727455 \\ -0.14399227 & 0.53276635 & 0.124353 \\ 0.25242392 & 0.59862403 & 0.35048801 \end{bmatrix}$$

$Z^{(2)}$ and $a^{(2)}$ are of size 5×3, one row for each car and one column for each hidden unit (neuron) of the layer 2.

To summarise, for now we have three matrices, our input X, our weights W^1 between the layer 1 and 2 and our first hidden layer $a^{(2)} = tanh(X.W^1)$ ($Z^{(2)}$ is an intermediary matrix that holds the value of $X.W^1$).



We will repeat the exact same steps but instead of using our matrix X as inputs, we will now use $a^{(2)}$. We add synapses from the layer 2 to the layer 3.



We also view the weights as a matrix.

$$W^2 = \begin{bmatrix} W_{11}^2 & W_{12}^2 \\ W_{21}^2 & W_{22}^2 \\ W_{31}^2 & W_{32}^2 \end{bmatrix}$$

The number of rows equals the number of neurons in the layer 2 and the number of columns equals the number of neurons in the layer 3. We will compute the values of our second hidden layer into the matrix $Z^{(3)}$ as we did previously. For a given layer, the input is always the output of the previous layer. For the layer 2, the output of the previous layer are the data from the layer 1, meaning X, for the layer 3 the output of the previous layer are the data from the layer 2, meaning $a^{(2)}$.

$$Z^{(3)} = a^{(2)}.W^2$$

And then we apply the activation function, we are keeping $tanh(x)$ as activation function because it is unusual to use different activation functions for each layer.

$$a^{(3)} = tanh(Z^{(3)})$$

Previously we found that:

$$a^{(2)} = \begin{bmatrix} -0.16838105 & 0.24773663 & 0.03997868 \\ -0.18967498 & 0.03498572 & -0.06889071 \\ 0.01399909 & 0.98576421 & 0.65727455 \\ -0.14399227 & 0.53276635 & 0.124353 \\ 0.25242392 & 0.59862403 & 0.35048801 \end{bmatrix}$$

Our W^2 matrix is randomly initialized the first time so we can use random values for the weights.

$$W^2 = \begin{bmatrix} 0.04 & 0.78 \\ 0.40 & 0.45 \\ 0.65 & 0.23 \end{bmatrix}$$

We calculate $Z^{(3)}$:

$$Z^{(3)} = a^{(2)}.W^2$$

$$Z^{(3)} = \begin{bmatrix} -0.16838105 & 0.24773663 & 0.03997868 \\ -0.18967498 & 0.03498572 & -0.06889071 \\ 0.01399909 & 0.98576421 & 0.65727455 \\ -0.14399227 & 0.53276635 & 0.124353 \\ 0.25242392 & 0.59862403 & 0.35048801 \end{bmatrix} . \begin{bmatrix} 0.04 & 0.78 \\ 0.40 & 0.45 \\ 0.65 & 0.23 \end{bmatrix}$$

$$Z^{(3)} = \begin{bmatrix} 0.11834555 & -0.01066064 \\ -0.03837167 & -0.14804778 \\ 0.8220941 & 0.60568633 \\ 0.2881763 & 0.15603208 \\ 0.47736378 & 0.54688371 \end{bmatrix}$$

We then apply our activation function:

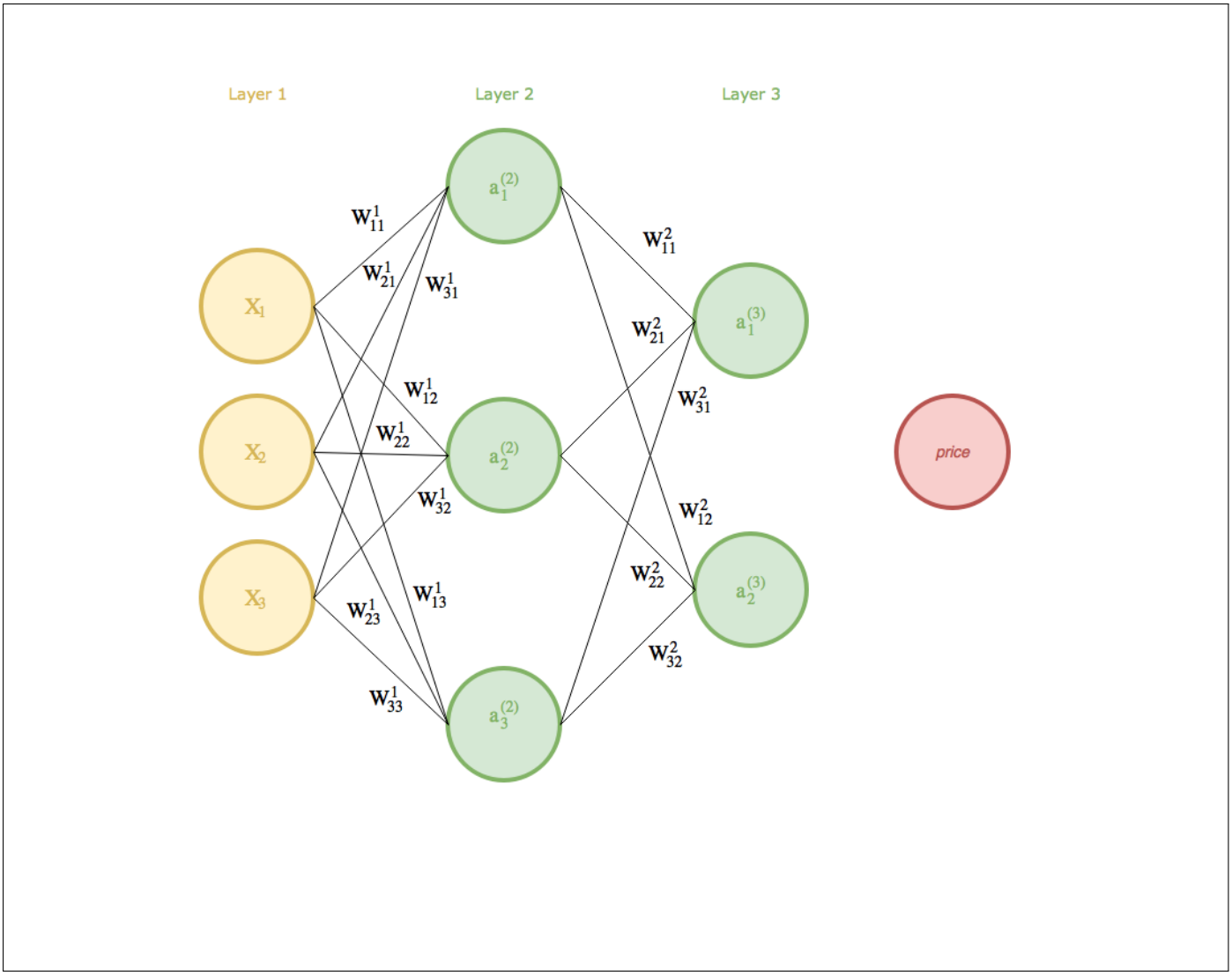
$$a^{(3)} = tanh(Z^{(3)})$$

$$a^{(3)} = \begin{bmatrix} tanh(0.11834555) & tanh(-0.01066064) \\ tanh(-0.03837167) & tanh(-0.14804778) \\ tanh(0.8220941) & tanh(0.60568633) \\ tanh(0.2881763) & tanh(0.15603208) \\ tanh(0.47736378) & tanh(0.54688371) \end{bmatrix}$$

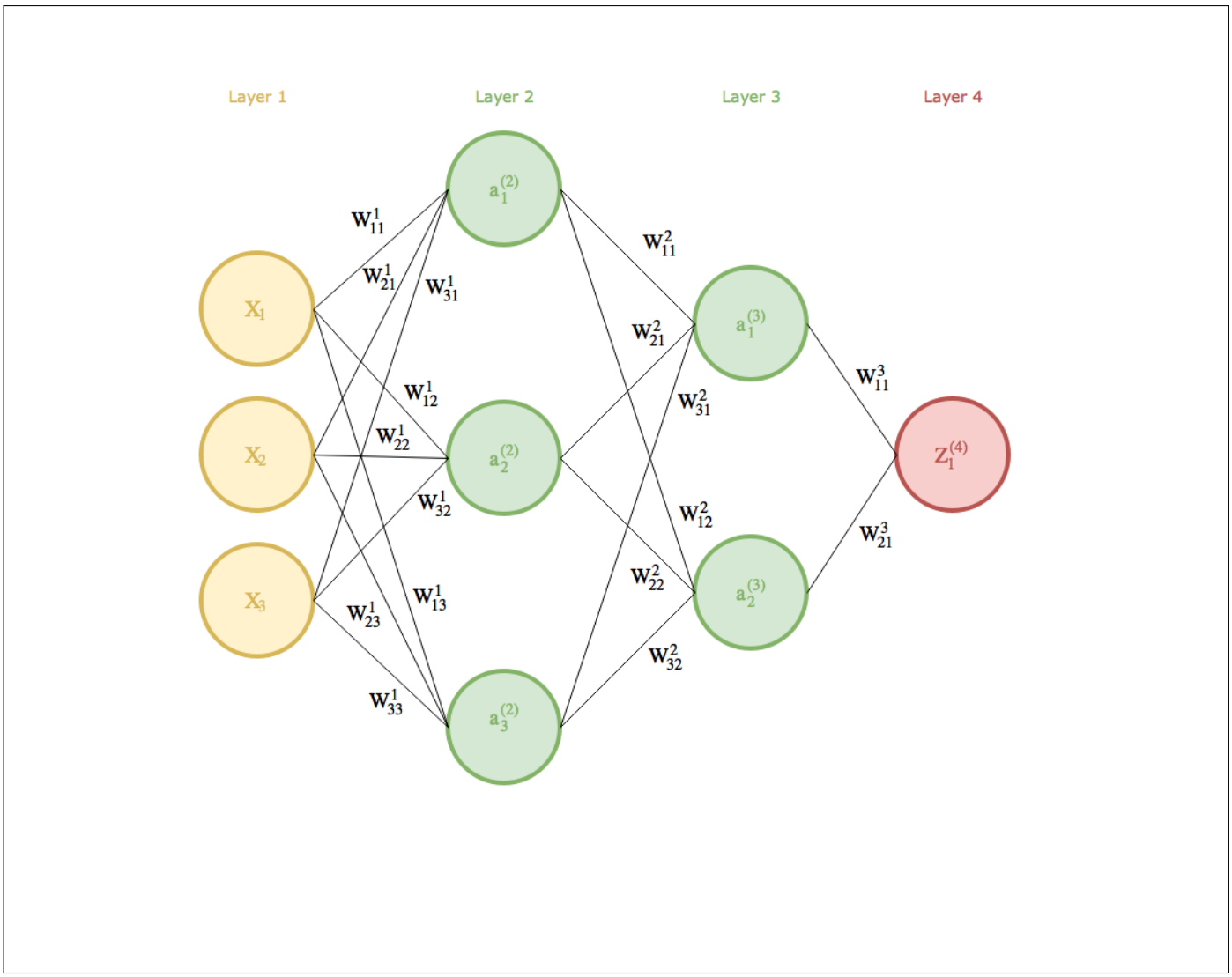
$$a^{(3)} = \begin{bmatrix} 0.11779613 & -0.01066023 \\ -0.03835285 & -0.14697553 \\ 0.67620804 & 0.54108347 \\ 0.28045542 & 0.15477804 \\ 0.44412987 & 0.49818098 \end{bmatrix}$$

$Z^{(3)}$ and $a^{(3)}$ are of size 5×2, one row for each car and one column for each hidden unit (neuron) of the layer 3.

Our network looks like:



We will now connect the last layer, using the same technique as previously seen.



As for the two previous layers, we have:

$Z^{(4)} = a^{(3)}.W^3 \; a^{(4)} = tanh(Z^{(4)})$

Where:

$$W^3 = \begin{bmatrix} W^3_{11} \\ W^3_{21} \end{bmatrix}$$

The number of rows equals the number of neurons in the layer 3 and the number of columns equals the number of neurons in the layer 4.

Our W^3 matrix is randomly initialized the first time so we can use random values for the weights. For instance:

$$W^3 = \begin{bmatrix} 0.04 \\ 0.41 \end{bmatrix}$$

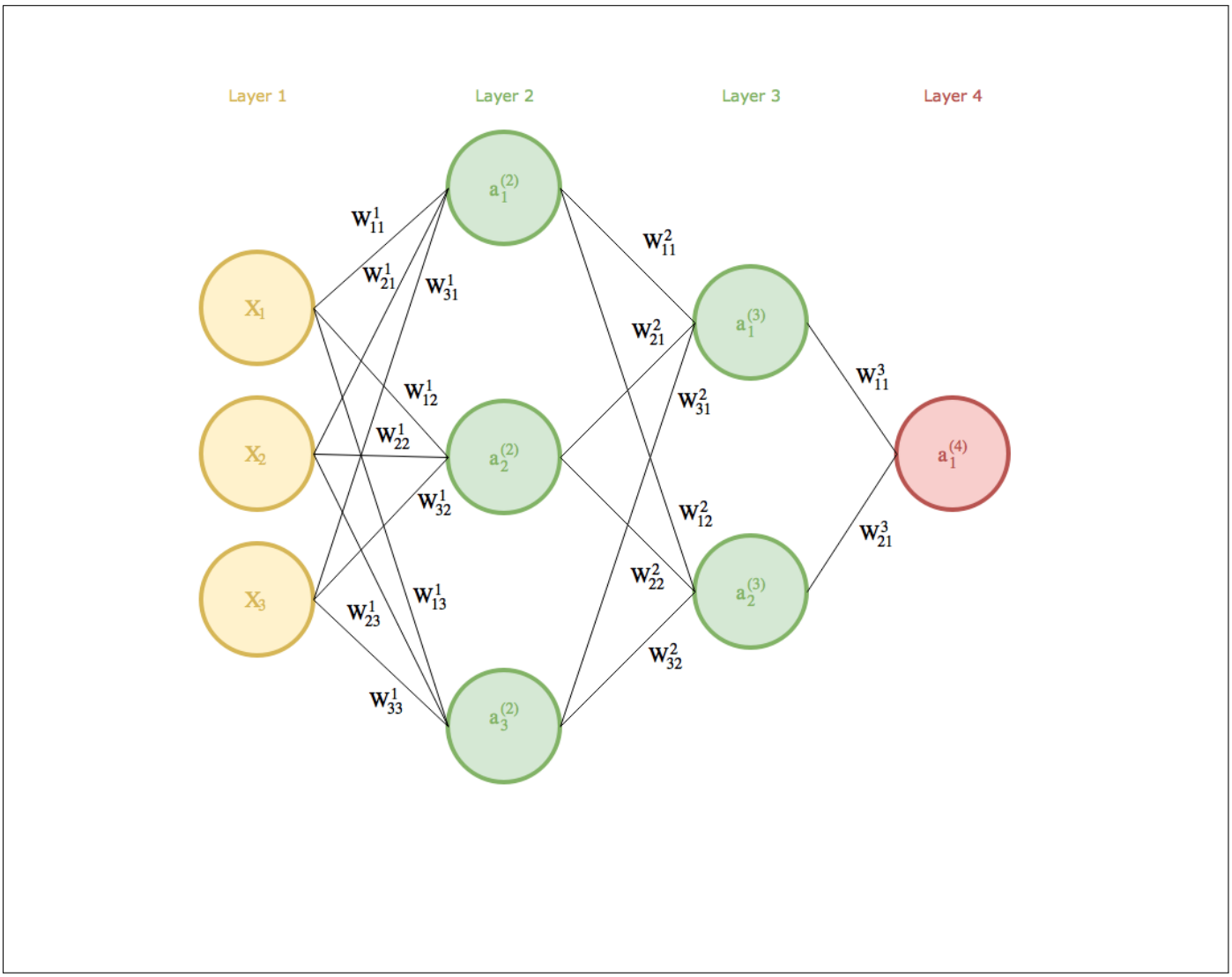
We then calculate $Z^{(4)}$ using $a^{(3)}$ which is the result of the previous layer.

$$Z^{(4)} = a^{(3)}.W^3 \quad Z^{(4)} = \begin{bmatrix} 0.11779613 & -0.01066023 \\ -0.03835285 & -0.14697553 \\ 0.67620804 & 0.54108347 \\ 0.28045542 & 0.15477804 \\ 0.44412987 & 0.49818098 \end{bmatrix} . \begin{bmatrix} 0.04 \\ 0.41 \end{bmatrix} \quad Z^{(4)} = \begin{bmatrix} 0.00034115 \\ -0.06179408 \\ 0.24889254 \\ 0.07467721 \\ 0.22201939 \end{bmatrix}$$

We then apply the activation function:

$$a^{(4)} = tanh(Z^{(4)}) = \begin{bmatrix} tanh(0.00034115) \\ tanh(-0.06179408) \\ tanh(0.24889254) \\ tanh(0.07467721) \\ tanh(0.22201939) \end{bmatrix} \quad a^{(4)} = \begin{bmatrix} 0.000341156 \\ -0.0617156 \\ 0.243877 \\ 0.0745387 \\ 0.218442 \end{bmatrix}$$

Finally our network looks like the following:



What we did is called the forward propagation, we have an input and we propagate it through the network. Each time we have an hidden layer, we compute the values of its neurons using:

$$a^{(l)} = \sigma(a^{(l-1)} \times W^{(l-1)})$$

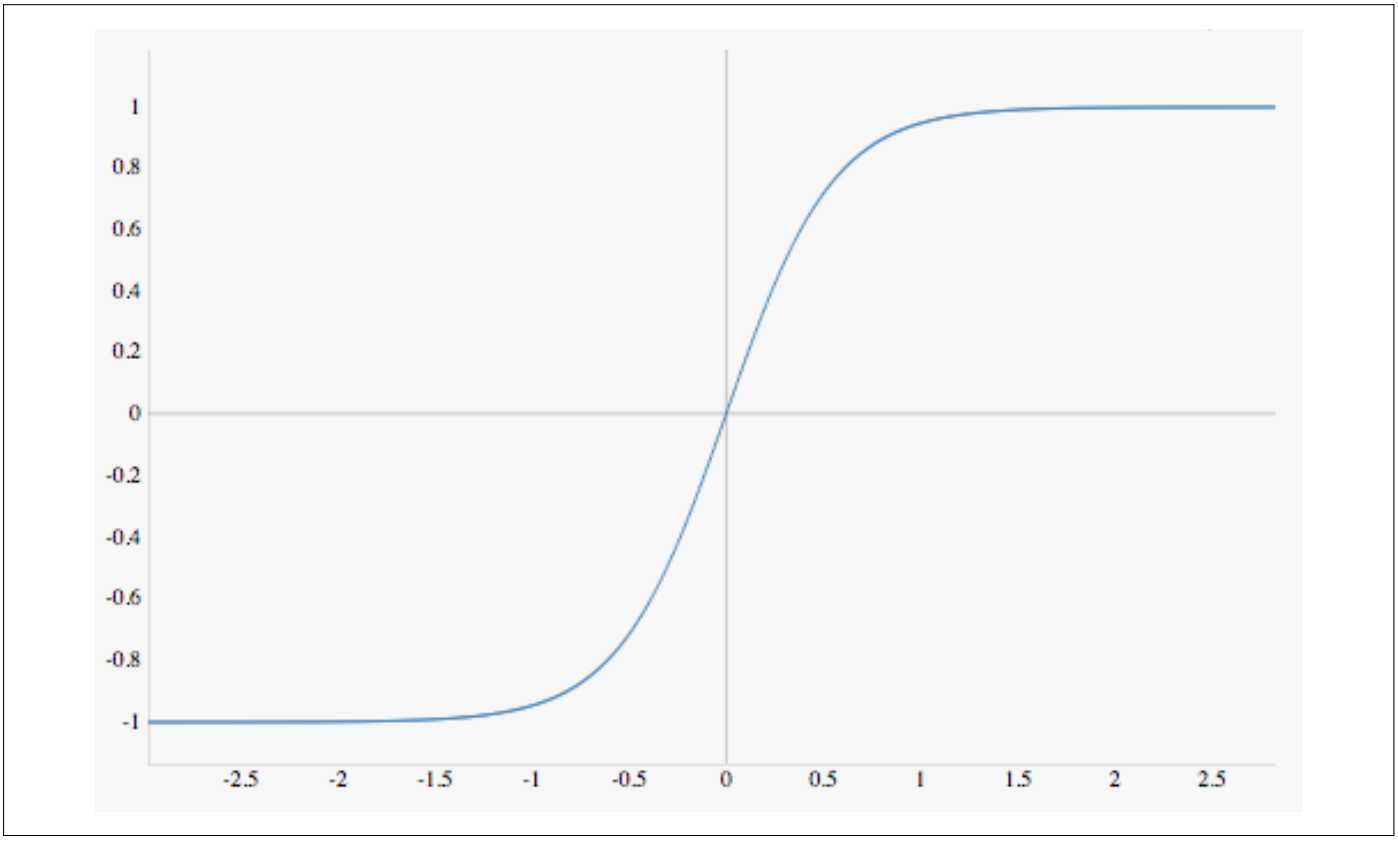
If $l = 1$, $a^{(1)} = X$ (our inputs), $\sigma(x)$ in our case is $tanh(x)$. Each time you see the above formula, you should think "neural network layer".

Chapter 4

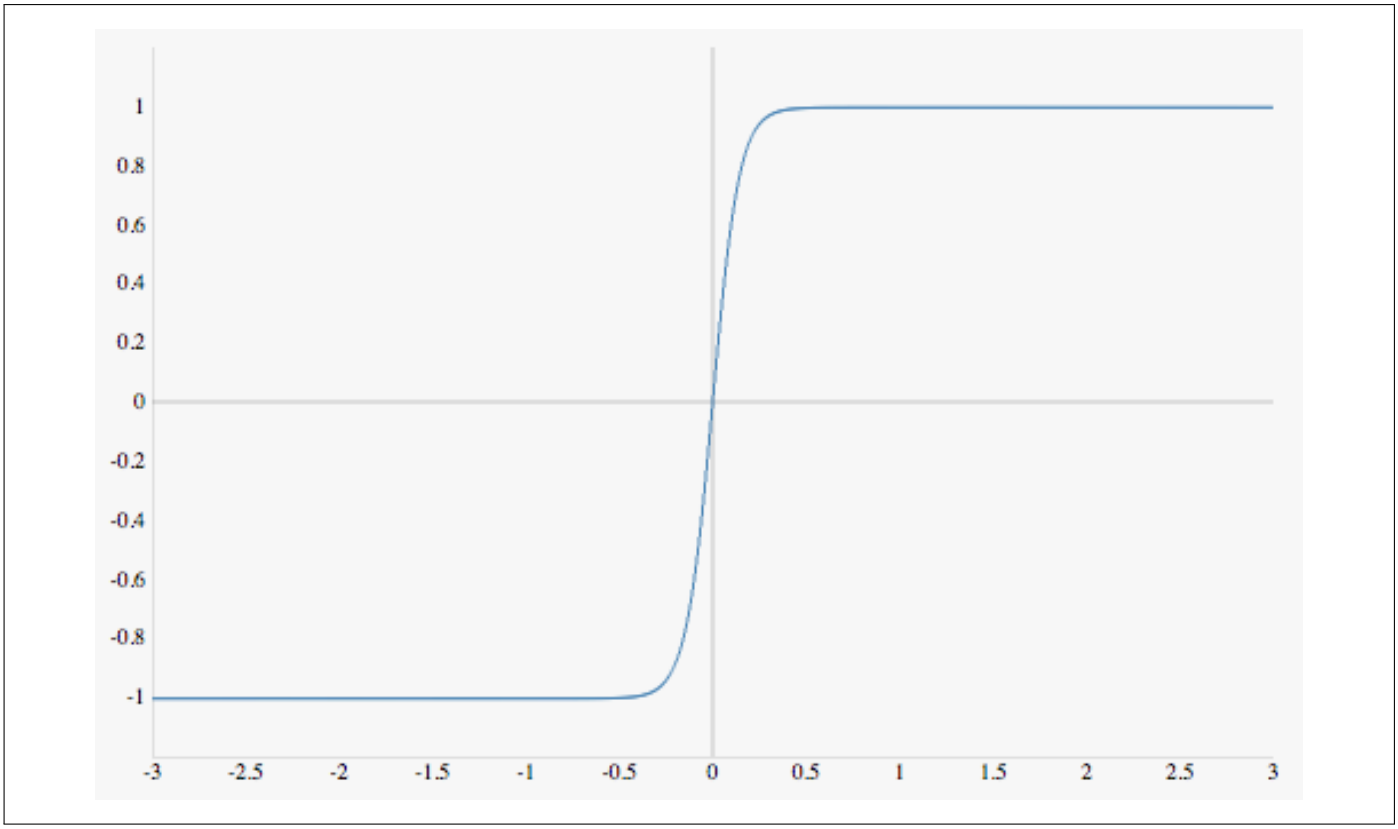
Bias

We didn't used a bias unit to keep things simple, we will add it now. Why do we need a bias? On the above picture we saw that $Z_1^{(2)} = X_1 \times W_{11}^1 + X_2 \times W_{21}^1 + X_3 \times W_{31}^1$ where X_1, X_2 and X_3 are the attributes of our car. This calculation will produce a number an then our activation function $tanh(Z_1^{(2)})$ will be applied. We have a 3D input (X_1, X_2, X_3), to explain the bias usefulness, we will keep only one feature, for instance the number of kilometers and assume that only the number of kilometers of a car describes its price. We now have $a_1^{(2)} = X_1 \times W_{11}^1$ where X_1 is the car's number of kilometers. And once the activation function is applied, we have $a_1^{(2)} = tanh(Z_1^{(2)}) = tanh(X_1 \times W_{11}^1)$.

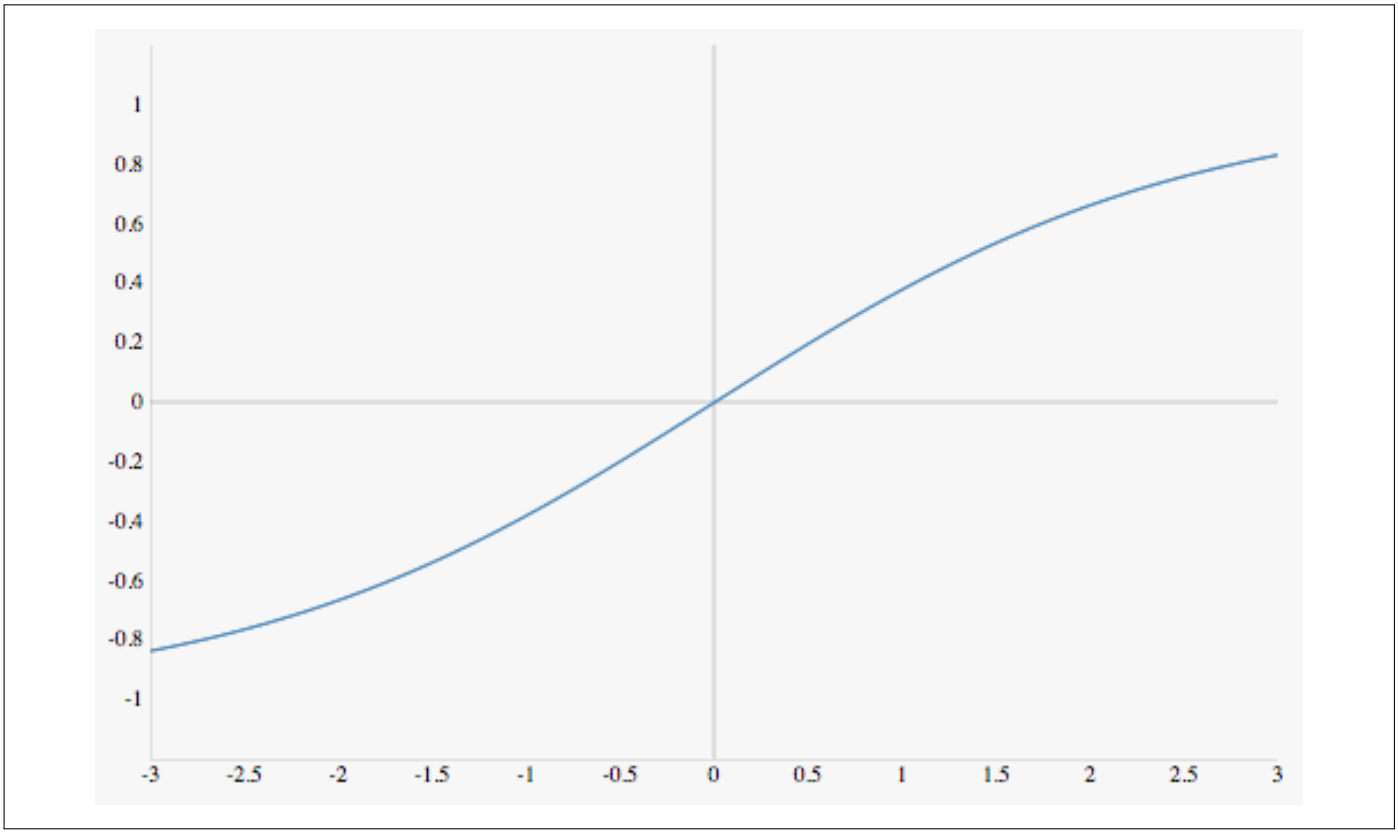
After the training part described later, our W_{11}^1 will be a learned value. For instance 1.8. Our X_1 is unknown and will be different for each car, $a_1^{(2)}$ will be the result of $tanh(1.8 \times X_1)$. We can draw this function:



As you can see this function is centered in zero, as our W_{11}^1 is a learned value, our network will be able to tweak it. We assumed that our network learned 1.8 but let's draw the graph when W_{11}^1 is larger, let's say 7.



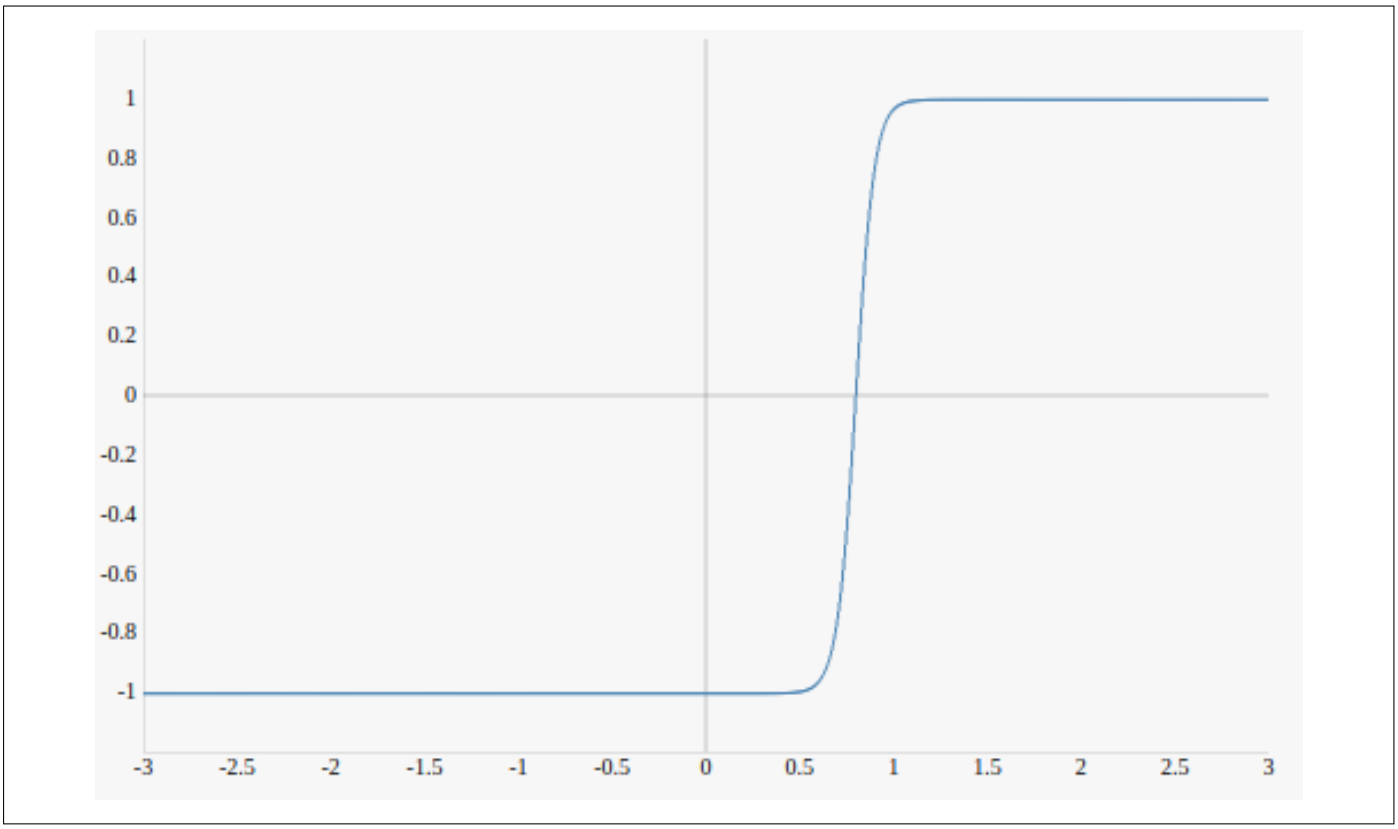
And when $W_{11}^1 = 0.4$:



As we can see, changing W_{11}^1 changes the stiffness of the graph, as our network can only tweak the weights, it will only be able to change the stiffness but will stay centered in zero. Is this a problem ?

Firstly the number of kilometers of a car will always be positive, so all the left part of the graph will be useless, $a_1^{(2)}$ will never be negative, is that a good thing ? We don't really know, but even if it was a good thing, we would have no choice.

Secondly, let's say that we have two cars, one with 30k kilometers and one with 170k kilometers, once normalized, we will have 0.5 and 2.5. Let's say that the impact of the number of kilometers is clearly determined, like if the car is less than 50k kilometers the price is high and if the car is more than 50k kilometers the price is low, the limit is clear. Once normalized 50k will be around 0.7, so we need a steep graph that produce -1 when x $<$ 0.7 and produce 1 when x $>$ 0.7. Like this one:



That would be perfect but unfortunately we can't move the graph to the right, so we are stuck with a graph centered in zero (the first image). We can only make the graph steep using a high value for W_{11}^1. The above picture would be a good solution for us, it is the graph of the function: $tanh(10x-8)$ Where 10 is the value for our W_{11}^1 and 8 is a constant that comes handy, the bias. Instead of $a_1^{(2)} = tanh(X_1 \times W_{11}^1)$ we will have $a_1^{(2)} = tanh(X_1 \times W_{11}^1 + b)$ and this little b will greatly improve our network performances because it will move the graph of our activation function to the left or the right and the result produced will be more representative of our problem. In our example (the above picture) $b = -8$.

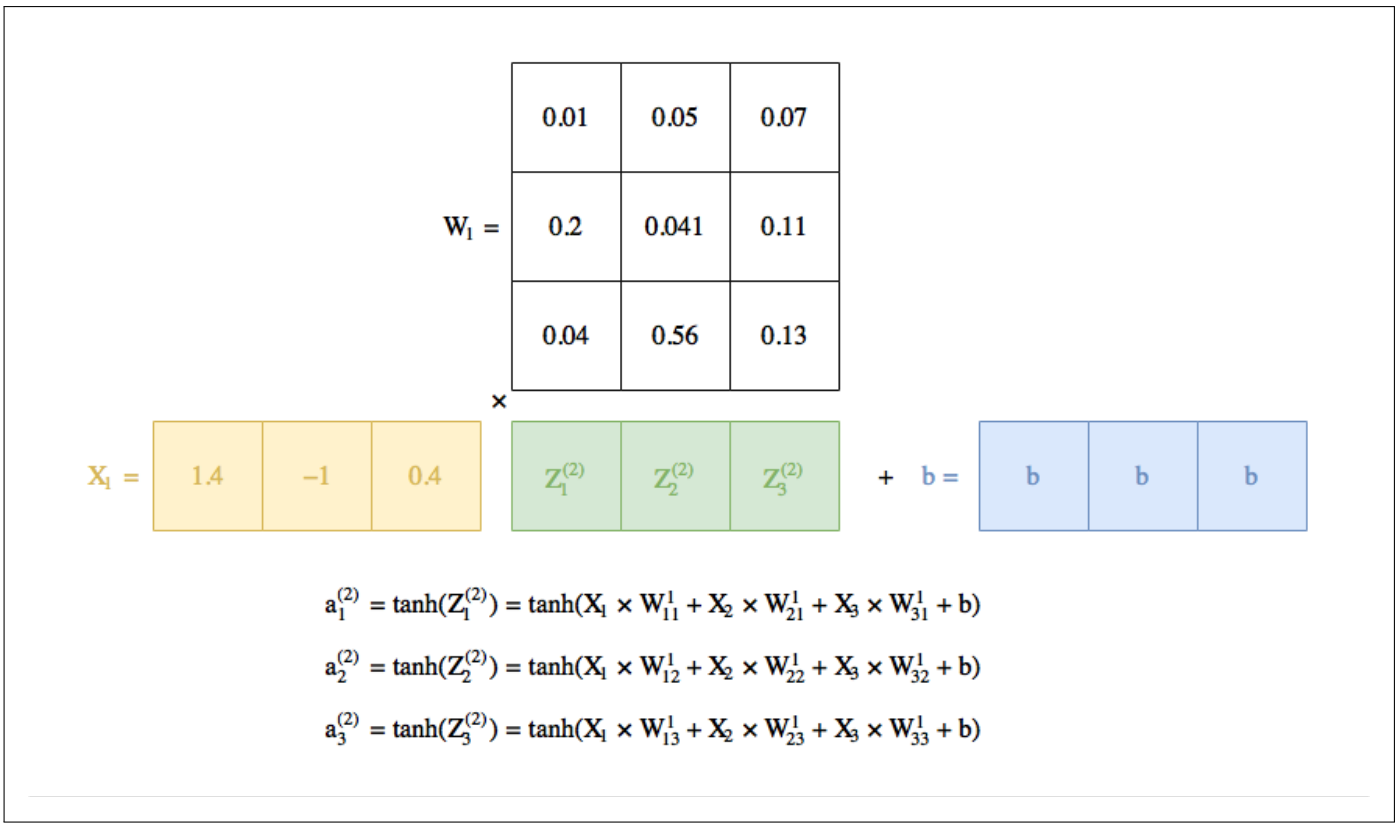
We used only one feature to easily draw graphs and display the impact of the bias but it is the same thing with more dimensions, our original example with a bias would be $a_1^{(2)} = tanh(X_1 \times W_{11}^1 + X_2 \times W_{21}^1 + X_3 \times W_{31}^1 + b)$.

This value b also needs to be learned, because regarding the problem, the bias will be different. The same bias must be added to each car. We saw previously the origin of the weights and their matrix representation, how can we add a bias?

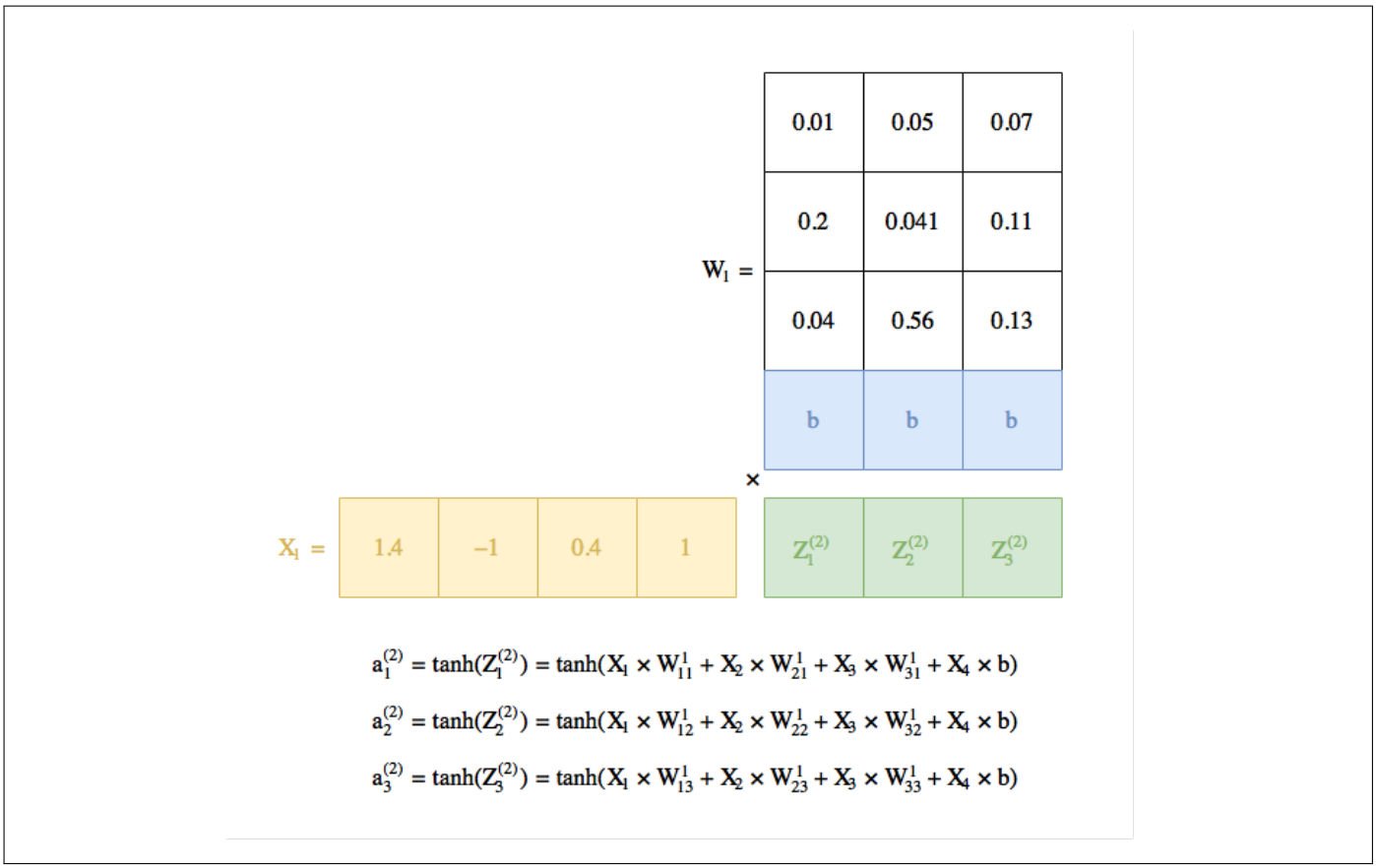
We will have a bias for each neuron. For instance for the first neuron of the first hidden layer we have:

$$a_1^{(2)} = tanh(X_1 \times W_{11}^1 + X_2 \times W_{21}^1 + X_3 \times W_{31}^1 + b)$$

As we have three neurons in the first hidden layer, we will need three biases, we could use a bias matrix.



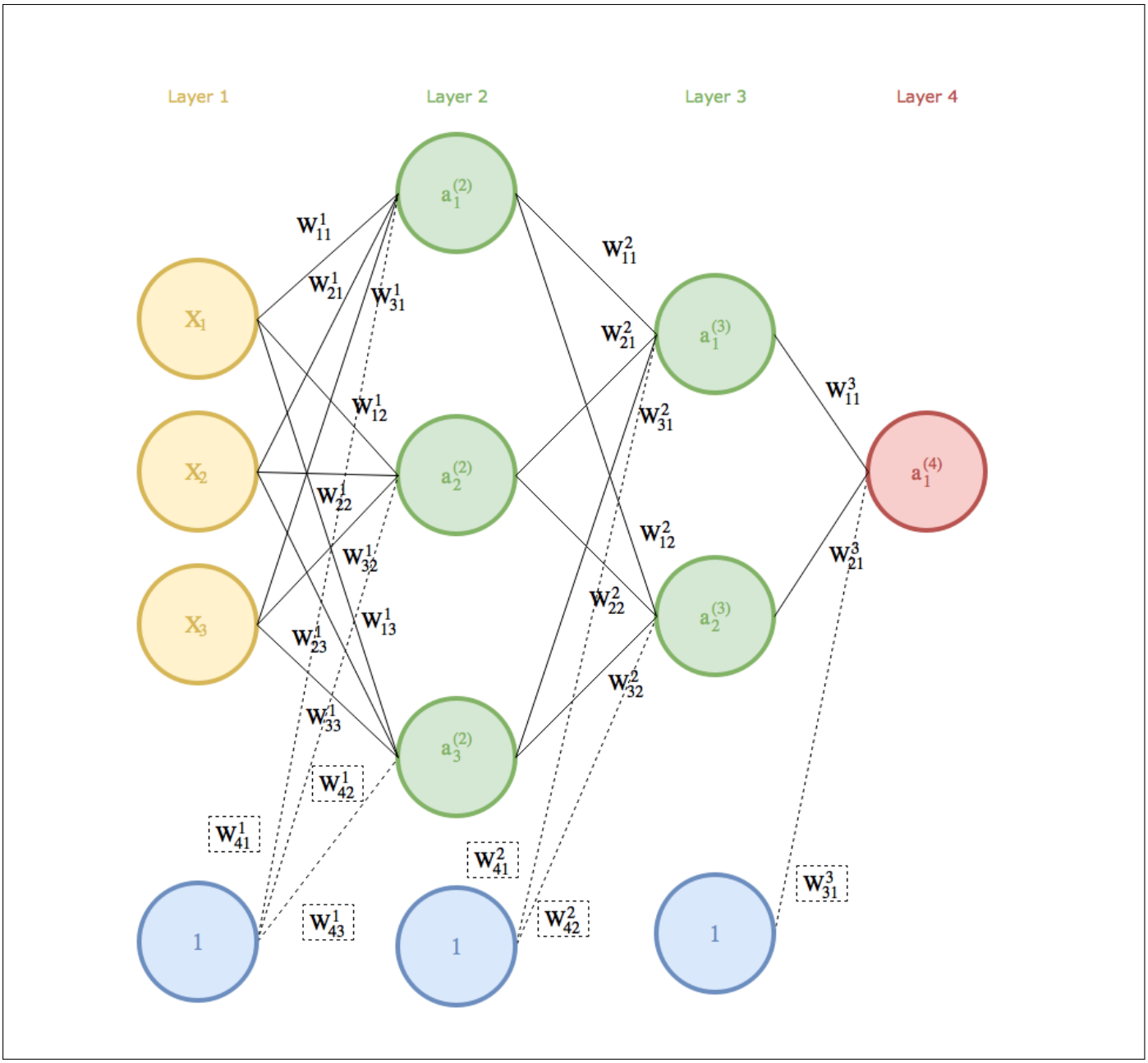
Nonetheless managing the biases and the weights in separate matrices can be cumbersome. As the biases are learned, there are not different from the weights nonetheless they should not depend of a car attributes because all the car will have different attributes whereas the biases should be the same for all the cars. The solution is to add an additional feature to each car with a value 1. So that this feature when multiplied with the bias will not change its value.



Note that X_4 is equal to 1 so the calculations of the two previous pictures are exactly the same but using

the bias trick we only have one weights matrix.

Adding a bias means adding a 1 feature to all our inputs. We add biases to our input layer and each one of our hidden layers. Now our network looks like.



The link between the bias and the neurons are dotted to keep the network readable but they are normal weights. Our cars have a new feature with the value 1. Each hidden layer has also a new neuron 1 because the problem that the bias solve appears each time we use the activation function (so each layer). There is no link between the bias and the previous layer because the bias is added after the calculations, it is not the result of a matrix multiplication.

We will do again all the calculus with the biases. We began by adding the bias unit to our input data, this means adding a new column of 1.

$$X = \begin{bmatrix} 1.4 & -1 & 0.4 & 1 \\ 0.4 & -1 & 0.1 & 1 \\ 5.4 & -1 & 4 & 1 \\ 1.5 & -1 & 1 & 1 \\ 1.8 & 1 & 1 & 1 \end{bmatrix}$$

And our W_1 matrix has a new row $[W_{41}^1 \quad W_{42}^1 \quad W_{43}^1]$, because adding 1 to our input created new links. This row is the biases, we init them at 0.1 so that we will see the differences with the previous values (they should be initialised at 0 or around 0).

$$W^1 = \begin{bmatrix} 0.01 & 0.05 & 0.07 \\ 0.20 & 0.041 & 0.11 \\ 0.04 & 0.56 & 0.13 \\ 0.1 & 0.1 & 0.1 \end{bmatrix}$$

Then we are doing the same calculations as before.

$$Z^{(2)} = \begin{bmatrix} 1.4 & -1 & 0.4 & 1 \\ 0.4 & -1 & 0.1 & 1 \\ 5.4 & -1 & 4 & 1 \\ 1.5 & -1 & 1 & 1 \\ 1.8 & 1 & 1 & 1 \end{bmatrix} . \begin{bmatrix} 0.01 & 0.05 & 0.07 \\ 0.20 & 0.041 & 0.11 \\ 0.04 & 0.56 & 0.13 \\ 0.1 & 0.1 & 0.1 \end{bmatrix}$$

$$Z^{(2)} = \begin{bmatrix} -0.07 & 0.353 & 0.14 \\ -0.092 & 0.135 & 0.031 \\ 0.114 & 2.569 & 0.888 \\ -0.045 & 0.694 & 0.225 \\ 0.358 & 0.791 & 0.466 \end{bmatrix}$$

If you compare these $Z^{(2)}$ results to the previous ones where we were not using biases, you see that each $Z^{(2)}$ has $+0.1$. As before we then apply the activation function.

$$a^{(2)} = tanh(Z^{(2)}) \; a^{(2)} = \begin{bmatrix} -0.06988589 & 0.33903341 & 0.13909245 \\ -0.09174131 & 0.13418581 & 0.03099007 \\ 0.11350871 & 0.98832966 & 0.71040449 \\ -0.04496965 & 0.60054553 & 0.22127847 \\ 0.34345116 & 0.65897516 & 0.43496173 \end{bmatrix}$$

We also add a bias to our first hidden layer. As we did with the inputs we must add 1 as a fourth neuron to $a^{(2)}$. As a reminder, each row contains data for one car.

$$a^{(2)} = \begin{bmatrix} -0.06988589 & 0.33903341 & 0.13909245 & 1 \\ -0.09174131 & 0.13418581 & 0.03099007 & 1 \\ 0.11350871 & 0.98832966 & 0.71040449 & 1 \\ -0.04496965 & 0.60054553 & 0.22127847 & 1 \\ 0.34345116 & 0.65897516 & 0.43496173 & 1 \end{bmatrix}$$

As we added a 1 neuron to $a^{(2)}$ new links are created between the layer 2 and 3, as for W^1, W^2 will gain a row of biases that we init at 0.1.

$$W^2 = \begin{bmatrix} 0.04 & 0.78 \\ 0.40 & 0.45 \\ 0.65 & 0.23 \\ 0.1 & 0.1 \end{bmatrix}$$

We do the same calculations as before:

$$Z^{(3)} = a^{(2)}.W^2$$

$$Z^{(3)} = \begin{bmatrix} -0.06988589 & 0.33903341 & 0.13909245 & 1 \\ -0.09174131 & 0.13418581 & 0.03099007 & 1 \\ 0.11350871 & 0.98832966 & 0.71040449 & 1 \\ -0.04496965 & 0.60054553 & 0.22127847 & 1 \\ 0.34345116 & 0.65897516 & 0.43496173 & 1 \end{bmatrix} . \begin{bmatrix} 0.04 & 0.78 \\ 0.40 & 0.45 \\ 0.65 & 0.23 \\ 0.1 & 0.1 \end{bmatrix}$$

$$Z^{(3)} = \begin{bmatrix} 0.32322802 & 0.2300453 \\ 0.17014822 & 0.09595311 \\ 0.96163513 & 0.79667817 \\ 0.48225043 & 0.38606321 \\ 0.66005324 & 0.76447193 \end{bmatrix}$$

$$a^{(3)} = tanh(Z^{(3)})$$

$$a^{(3)} = \begin{bmatrix} 0.31242279 & 0.22607134 \\ 0.16852506 & 0.09565971 \\ 0.74500533 & 0.66217559 \\ 0.44804409 & 0.3679614 \\ 0.57839884 & 0.64370347 \end{bmatrix}$$

Finally we also add a bias neuron to $a^{(3)}$, as before new links are created between the layer 3 and 4, as for W^2, W^3 will gain a row of biases that we init at 0.1.

$$Z^{(4)} = a^{(3)}.W^3 \; Z^{(4)} = \begin{bmatrix} 0.31242279 & 0.22607134 & 1 \\ 0.16852506 & 0.09565971 & 1 \\ 0.74500533 & 0.66217559 & 1 \\ 0.44804409 & 0.3679614 & 1 \\ 0.57839884 & 0.64370347 & 1 \end{bmatrix} . \begin{bmatrix} 0.04 \\ 0.41 \\ 0.1 \end{bmatrix} \; Z^{(4)} = \begin{bmatrix} 0.20518616 \\ 0.14596148 \\ 0.4012922 \\ 0.26878594 \\ 0.38705438 \end{bmatrix}$$

Then we apply the activation function.

$$a^{(4)} = tanh(Z^{(4)}) = \begin{bmatrix} 0.2023543 \\ 0.14493368 \\ 0.38105408 \\ 0.26249479 \\ 0.36881806 \end{bmatrix}$$

The values obtained on the output layer are our network predictions. We will save them into the matrix $\hat{y} = a^{(4)}$. It is the price of the cars predicted by our network whereas y is the real price of the cars (given by the dataset). For instance our network thinks that the car number one is less expensive than the car number five.

Adding the biases is really simple (just add a neuron 1) and will greatly improve our results. The whole calculation that we did, from X to $a^{(4)}$ is called forward propagation. This is how we will infer the price of a car in the future. We will take the car attributes, make the same calculations and get a price in the $a^{(4)}$ matrix.

Right now our results are pretty bad because we randomly initialised our weights. We will train our network step by step in order to tweak the weights until it outputs good predictions.

In our original dataset we have the car attributes and the car price. The features (attributes) of the first car with the bias was $X = \begin{bmatrix} 1.4 & -1 & 0.4 & 1 \end{bmatrix}$ and its price was $y = 0.45$. Our network output was $\hat{y} = 0.2023543$ given in $a_1^{(4)}$. We clearly see that we are missing around 0.25 by doing $y - \hat{y}$. This is called the cost, how

bad our network predicted the price compared to the actual price. Actually we will define a cost function in order to mesure how bad our predictions are.

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

This gives us the cost for one example. For instance for our first car, its real price is 0.45, our network outputted 0.2023543, using the above formula we have an error of $J(W) = \frac{1}{2}(0.45 - 0.2023543)^2 = 0,031$

This gives us the error for the first car, we will do it for all the cars then sum up the errors, so our formula become:

$J(W) = \sum_1^n \frac{1}{2}(y - \hat{y})^2$

Where n is the number of cars. We squared the error to get its absolute value but also because the quadratic function x^2 is convex, it means that the function has only one minimum. We introduced $\frac{1}{2}$ for later convenience.

Chapter 5

Gradient Descent

The function $J(W)$ gives us the error of our network regarding our inputs X and the weights of our network. If we replace $\hat{y}$ by its calculations, our function is:

$J(W) = \sum_1^n \frac{1}{2}(y - \tanh(\tanh(\tanh(X.W_1).W_2).W_3))^2$

$J(W)$ is a function that gives us the cost regarding our examples (the cars) and the weights (W_1, W_2 and W_3). The minimum the cost is, the better our network predicts. Our goal is to minimize the function $J(W)$, i.e: find its minimum. This is an optimization problem. We can't touch our examples X so we will minimize our function $J(W)$ by tweaking the weights. We will use the batch gradient descent algorithm (with a non convex cost function it is better to use the stochastic gradient descent). We choose the gradient descent as the optimization algorithm but other alternatives could be used. Let's see what a gradient is.

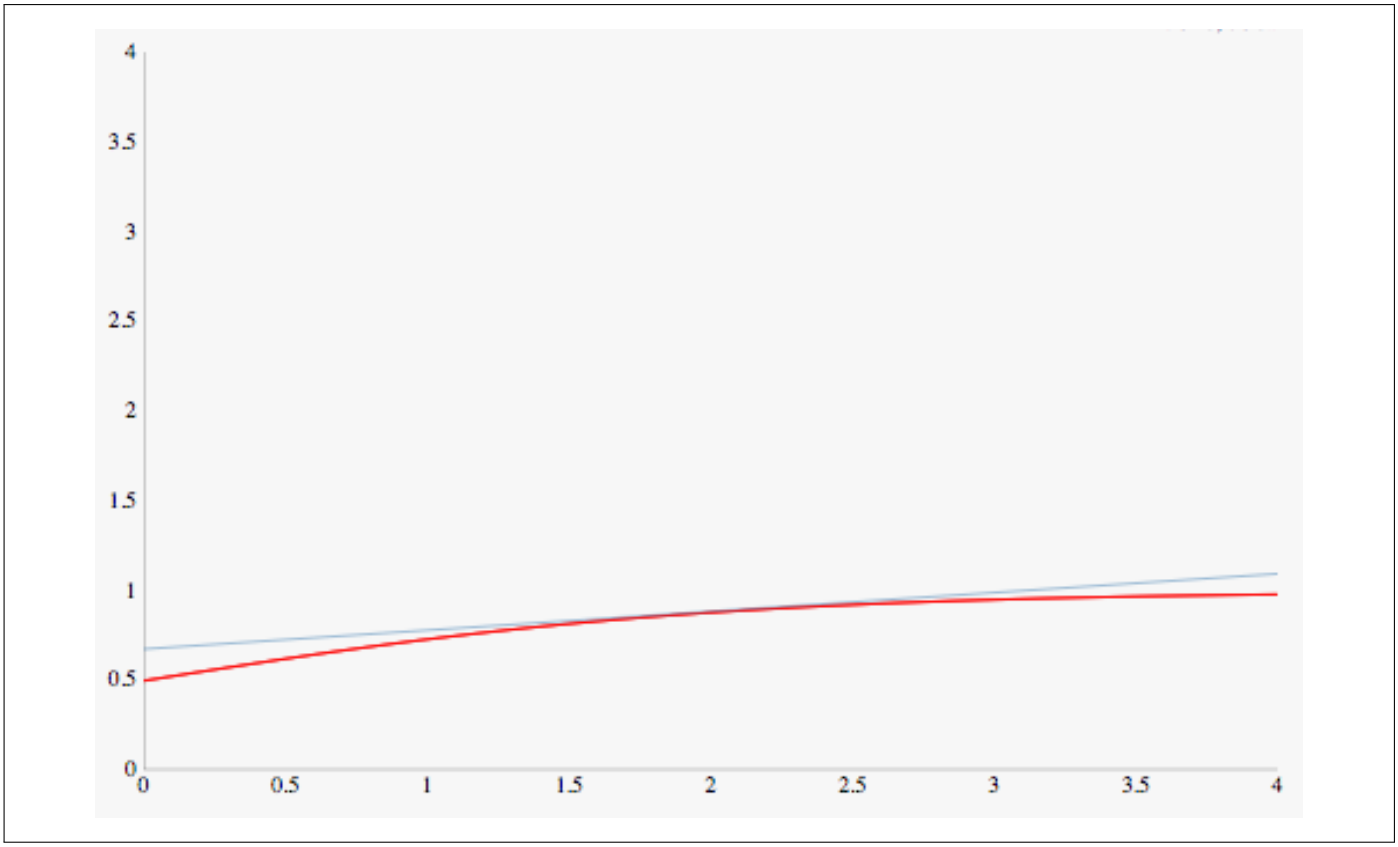
In mathematics, a function is a rule explaining how to process an input to give an output. The input is noted as x and the output as y, the function is generally written as $y = f(x)$. It is possible to have multiple inputs and outputs, multiple inputs is common and looks like $z = f(x, y)$, multiple outputs is a vector valued function that produces a vector instead of only y.

Strictly speaking the inputs are called independent variables and the outputs dependent variables. The function explains how the dependent variables depend on the independent variables.

The derivative of a function is a key tool in machine learning, it is leveraged among others by the gradient descent algorithm. The derivative measures how a change in the independent variables impact the dependent variables (how changing x impacts y). The process of finding the derivative is called differentiation.

There are two cases useful to us.

The first case is when there is only one independent variable and one dependent variable ($y = f(x)$ where $x \in \mathbb{R}$ and $y \in \mathbb{R}$). In that case the derivative of a function at a given input is the slope of the tangent line to the graph of the function at the given input. It means that at the given input, the function is well-approximated by a straight line, and the derivative is the slope of this straight line. Let's take an example. For instance if we take the sigmoïd function: $\sigma(x) = \frac{1}{1+e^{-x}}$ The derivative of this function is: $\frac{e^x}{(e^x+1)^2}$ You can compute the derivative detail here. Let's use this derivative to calculate the slope of the tangent at $x = 2$, we have: $\frac{e^2}{(e^2+1)^2} \approx 0,105$ If we draw the sigmoid function with its tangent at $x = 2$ we see that the slope of the tangent is $\approx 0,105$:



Our function does not looks like the sigmoid function because we used the same scale for the x and y, it is easier to see that the slope of the tangent, the blue line, is indeed $\approx 0,105$.

The tangent is the best linear approximation of a function at a given value, it shows how the independent variable impacts the dependent variable (small or big slope).

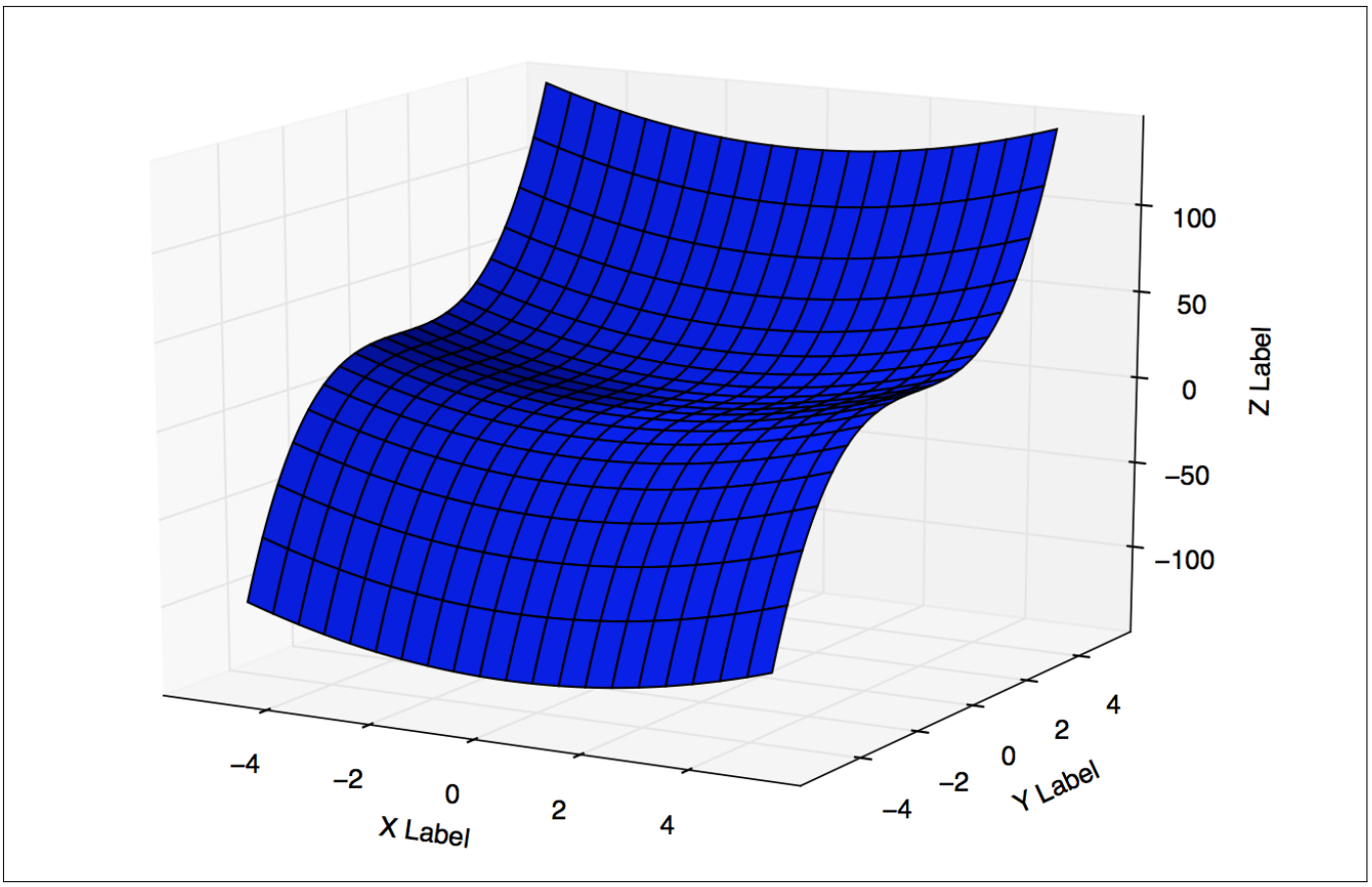
The second case is when there are several independent variables ($y = f(x)$ where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$). As the function takes as input several variables, we will now compute the partial derivative of the function with respect to each input variable.

The partial derivative of a function w.r.t (with respect to) one of the input variable is the derivative of the function where others variables held constant. It indicates the rate of change of a function with respect to that variable surrounding an infinitesimally small region near a particular point.

If we take for example, $z = x^2 + y^3$, we have two inputs x and y so the derivative of the function will be a vector containing two partial derivatives: $F = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$ To calculate the partial derivative $\frac{\partial f}{\partial x}$ we held y^3 as a constant so we have: $\frac{\partial f}{\partial x} = 2x + 0 = 2x$ To calculate the partial derivative $\frac{\partial f}{\partial y}$ we held x^2 as a constant so we have: $\frac{\partial f}{\partial y} = 3y^2 + 0 = 3y^2$ So the derivative of our initial function is: $F = [2x, 3y^2]$ This vector of partial derivatives is the gradient. It represents the slope of the tangent of the graph of the function, it means that the gradient points in the direction of the greatest rate of increase of the function and its magnitude is the slope of the graph in that direction.

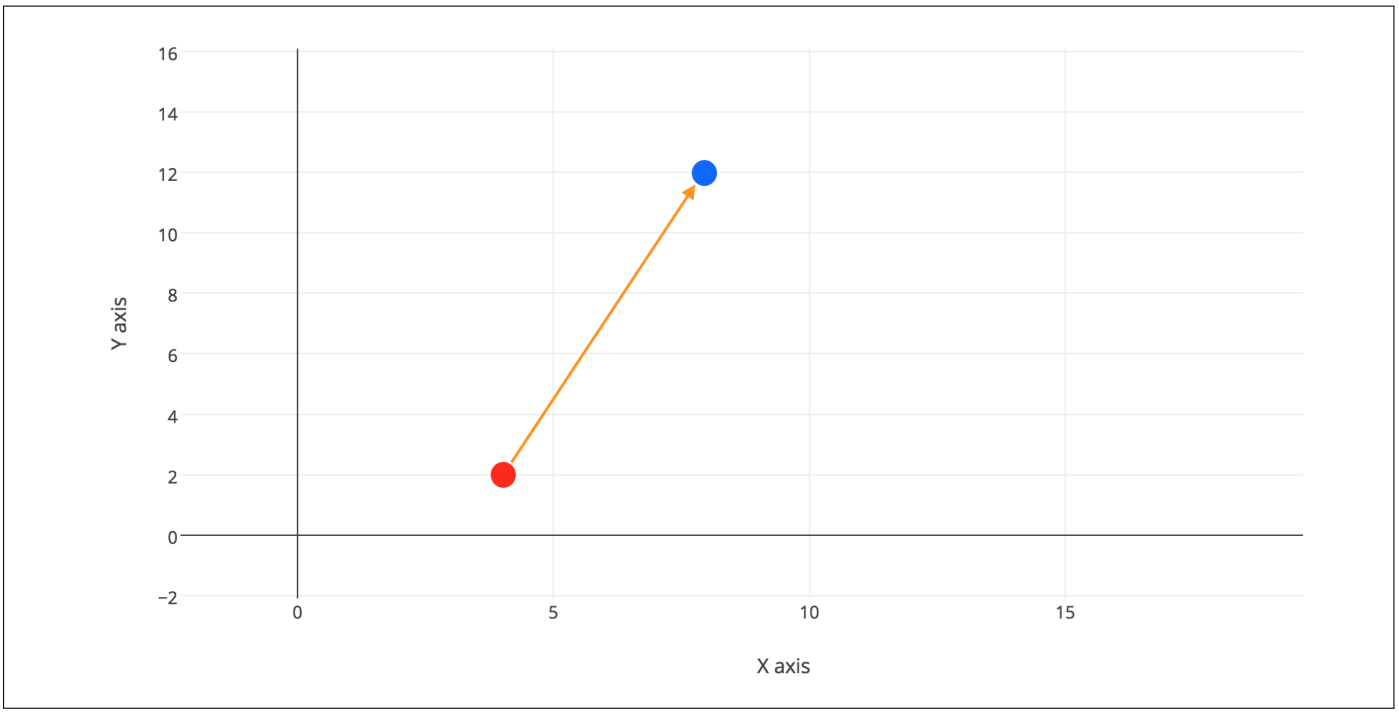
For instance, for the previous function, if we are at $x = 4$ and $y = 2$, the corresponding gradient is $(8, 12)$ (we are using $(2x, 3y^2)$). It means that the function increases more in the direction of y than x. Using the two informations, it gives us a vector (a direction) for which Z will increase the most.

If we draw the previous function in a 3D space, it looks like the following:



If we forget the z (because we can't tune its value) we have a 2D surface on which we can move on the x and y axis, it is like watching the previous 3D drawing from above, the z axis does not appear.

We can place our original point ($x = 4$ and $y = 2$) in red, its derivative $(8, 12)$ in blue and draw an arrow between them (orange).



The orange arrow is the gradient, it gives us the direction of the steepest increase of z and the length of the arrow gives us how z changes, a long arrow means a big slope. If it is always unclear, here is a 5 minutes video that can help you understand.

Gradient descent is an optimization algorithm, it allows to find a local minimum of a function.

The algorithm comes from the observation that if one goes in the direction of the negative gradient of a function, the function decreases faster.

As we saw previously, the gradient gives us the direction of the maximum increase of the function, we use the negation of this gradient to update the coordinates of our position in the space.

By doing it repeatedly we are sure to find a local minimum of the function.

Let's say that our function is $z = x^2 + y^3$, its gradient is $F = [2x, 3y^2]$ so if we are at $x = 4$ and $y = 2$, the corresponding gradient is $(8, 12)$.

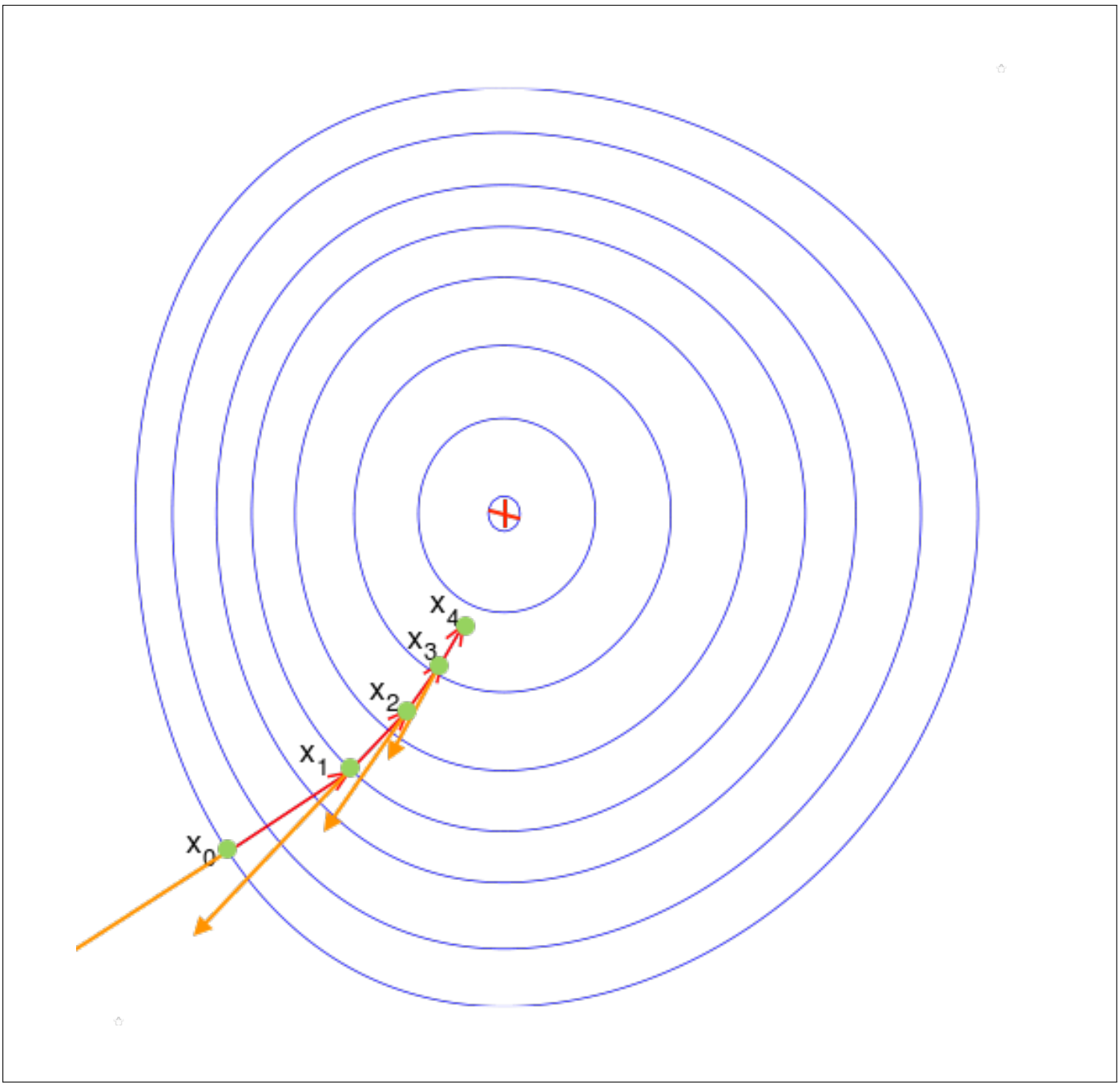
The basic idea is to substract this gradient from our original coordinates, but directly substracting the gradient would produce a big jump from our original coordinates. For our example we are at $(4, 2)$, if we directly substract the gradient, we will be at $(4-8, 2-12) = (-4, -10)$. What a big jump. So before to substract it we multiply it by a coefficient so that its value will be reduced. We have $(4-0.01*8, 2-0.01*12) = (3.92, 1.88)$. We did a small move but at least we will not miss the minima.

This coefficient 0.01 is called the learning rate, how much the gradient will impact our current position. A big learning rate means bigger steps but we could jump over the minimum of the function. A small learning rate means a little but precise step and more steps needed to find the minimum. This value must be small and the perfect value is determined by a try/fail process.

So if we summarize, we are at a point in our space and we want to find the minimum of a function. If we find the minimum we will have found the weights that give us the lowest error. So for the given coordinates, we compute the gradient, we multiply the gradient by a tiny coefficient and then we substract this updated gradient from our coordinates. This gives us our new coordinates and we start again.

This repeated process is the gradient descent algorithm. If the function that we are trying to minimize is convex, all local minimas are also global minimas so gradient descent descent will give us the global solution (but it can take an eternity). A convex cost function is not mandatory.

If we complete the image showing several iterations from wikipedia, we have:



The red cross is our minima, the red arrow is the negative gradient that we substract from our coordinates and the orange arrows are the actual gradients. As you can see the red arrow is smaller than the orange arrow because the gradient has been multiplied by the learning rate (so reduced). The green dots are the new coordinates produced after each iteration. If X_0 is our original point, we see four iterations on the picture.

Chapter 6

Back Propagation

Our goal is therefore to find the gradients of our function $J(W)$ and use them to update the weights of our network. Our cost function computes three inputs that are the network weights. We have to find the partial derivatives (gradients) with regards to its weights.

$$\nabla(J(W)) = \left[\frac{\partial J(W)}{\partial W_1}, \frac{\partial J(W)}{\partial W_2}, \frac{\partial J(W)}{\partial W_3}\right]$$

Then we will update the weights using the gradients, for instance W_1 will be updated using the following rule:

$$W_1 = W_1 - \alpha \frac{1}{n} \frac{\partial J(W)}{\partial W_1}$$

Where α is our learning rate, we divide by n, which is the number of inputs (our cars) because the gradients for each car will be summed and we are using the averaged gradient to update our weights W_1.

We could directly apply the backpropagation algorithm to find the gradients of $J(W)$ but we will compute the derivative of our cost function to get a better understanding of the algorithm.

As a reminder:

$$J(W) = \sum_1^n \frac{1}{2}(y - \tanh(\tanh(\tanh(X.W_1).W_2).W_3))^2$$

Our cost function is a composition of several functions, you can see a $tanh$ into a $tanh$ into a $tanh$. To derive it we will use the chain rule. It is a formula for computing the derivative of the composition of two or more functions.

The usual formula is:

$$(f \circ g)' = (f' \circ g).g'$$

For instance if we take the function $f(x) = (2x^2 + 8)^3$ we see a composition. The result of the first function $g(x) = 2x^2 + 8$ is used by the second function $f(g(x)) = (g(x))^3$. The derivative of $g(x)$ is $g'(x) = 4x$ and the derivative of $f(g(x))$ is $f'(g(x)) = 3g(x)^2$. We apply the above formula:

$$f'(x) = f'(g(x)).g'(x) = 3(2x^2 + 8)^2 \cdot 4x$$

The chain rule can also be written in the following way:

$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$

Meaning that if y depends on x and z depends on y, z also depends on x. If we use our previous example. We want $\frac{\partial f}{\partial x}$. We know that f depends on g and g depends on x because $f(g(x)) = g(x)^3$ and $g(x) = 2x^2 + 8$ so we can write:

$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$

Then we compute the derivative:

$\frac{\partial f}{\partial g} = 3g(x)^2$

$\frac{\partial g}{\partial x} = 4x$

$\frac{\partial f}{\partial x} = 3(2x^2 + 8)^2 \cdot 4x$

We will use the same way to compute the gradients of our cost function. There is a sum in our cost function, meaning that we have to add all together the cost of each input (our cars) in order to obtain the overall cost. We will forget it for now and talk about it later, the sum rule allows us to ignore it for now. The derivative of the sums equals the sum of the derivatives. At the end, once we have the derivative for one example we will just sum up the derivatives of all examples. So we have:

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

And we have to find:

$\nabla(J(W)) = \left[\frac{\partial J(W)}{\partial W_1}, \frac{\partial J(W)}{\partial W_2}, \frac{\partial J(W)}{\partial W_3} \right]$

We begin from the output of our network, so let's find $\frac{\partial J(W)}{\partial W_3}$ first. We want the derivative of $J(W)$ with regards to W_3, it means that other variables, meaning W_1 and W_2 are held constants.

We have:

$\frac{\partial J(W)}{\partial W_3} = \frac{1}{2}(y - \hat{y})^2$

We can see a first composition. $J(W) = \frac{1}{2}(g(x))^2$ where $g(x) = y - \hat{y}$ so we have:

$\frac{\partial J(W)}{\partial W_3} = \frac{\partial J(W)}{\partial g} \cdot \frac{\partial g}{\partial W_3}$

Using the power rule, we have:

$\frac{\partial J(W)}{\partial g} = 2 \times \frac{1}{2}(g(x)) = g(x)$

Thats why we put $\frac{1}{2}$ in our cost function, so that when differentiating things go smoothly and the term $2 \times \frac{1}{2}$ disappear. So:

$\frac{\partial J(W)}{\partial W_3} = g(x) \cdot \frac{\partial g}{\partial W_3} = (y - \hat{y}) \cdot \frac{\partial g}{\partial W_3}$

Now let's find the $\frac{\partial g}{\partial W_3}$ term, as a reminder $g(x) = y - \hat{y}$, we know that y is a constant that does not depend on W_3 whereas $\hat{y}$ does depend on it. We have:

$\frac{\partial g}{\partial W_3} = 0 - \frac{\partial \hat{y}}{\partial W_3}$

And:

$\frac{\partial J(W)}{\partial W_3} = (y - \hat{y}) \cdot - \frac{\partial \hat{y}}{\partial W_3}$

As we said before, $\hat{y}$ is our predictions, meaning the output of our network, meaning $a^{(4)}$. We know that $\hat{y} = a^{(4)} = tanh(Z^{(4)})$ so our $\hat{y}$ depends on $Z^{(4)}$ and $Z^{(4)}$ depends on W_3. So we can write:

$\frac{\partial J(W)}{\partial W_3} = (y - \hat{y}) \cdot - \frac{\partial \hat{y}}{\partial Z^{(4)}} \cdot \frac{\partial Z^{(4)}}{\partial W_3}$

We can find $\frac{\partial \hat{y}}{\partial Z^{(4)}}$ directly, we have: $\hat{y} = tanh(Z^{(4)})$ so our derivative with regards to $Z^{(4)}$ is the following:

$\frac{\partial \hat{y}}{\partial Z^{(4)}} = tanh'(Z^{(4)}) = 1 - \tanh(Z^{(4)})^2$

We can replace its value in our initial formula:

$\frac{\partial J(W)}{\partial W_3} = (y - \hat{y}) \cdot -(1 - \tanh(Z^{(4)})^2) \cdot \frac{\partial Z^{(4)}}{\partial W_3}$

Now we have one final term to compute $\frac{\partial Z^{(4)}}{\partial W_3}$ and we know that $Z^{(4)} = a^{(3)} \cdot W^3$. Finally $Z^{(4)}$ depends on W_3 directly so no more chain rule needed for this first gradient. We keep $a^{(3)}$ as a constant and W^3 becomes one because we are differentiating with regard to W^3. We have:

$\frac{\partial Z^{(4)}}{\partial W_3} = a^{(3)} \times 1 = a^{(3)}$

We can replace it in our initial formula:

$\frac{\partial J(W)}{\partial W_3} = (y - \hat{y}) \cdot -(1 - \tanh(Z^{(4)})^2) \cdot a^{(3)}$

We found our first gradient! We will use it during the training process to update our weights W_3.

We will introduce $\delta^{(4)}$ equals to:

$\delta^{(4)} = (y - \hat{y}) \cdot -(1 - \tanh(Z^{(4)})^2)$

So our previous gradient is in fact:

$\frac{\partial J(W)}{\partial W_3} = \delta^{(4)} \cdot a^{(3)}$

Now we need our second gradient, $\frac{\partial J(W)}{\partial W_2}$, we will use the same steps as before for the beginning. We begin from:

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

Using the exact same steps as for W_3 we will arrive at:

$\frac{\partial J(W)}{\partial W_2} = (y - \hat{y}) \cdot -(1 - \tanh(Z^{(4)})^2) \cdot \frac{\partial Z^{(4)}}{\partial W_2}$

You can see above that we have the same term as before, that's why we introduced $\delta^{(4)}$, we can replace it in our formula:

$\frac{\partial J(W)}{\partial W_2} = \delta^{(4)} \cdot \frac{\partial Z^{(4)}}{\partial W_2}$

Before we were searching the derivative of $Z^{(4)}$ with regards to W_3 so the derivative was $a^{(3)} \times 1$, as a reminder $Z^{(4)} = a^{(3)} \cdot W_3$ but this time we are searching the derivative with regards to W_2. W_3 does not depend on W_2 so it becomes a constant, meanwhile $a^{(3)}$ depends on W_2 so we have to find its derivative with regards to W_2. This gives us:

$\frac{\partial Z^{(4)}}{\partial W_2} = W_3 \cdot \frac{\partial a^{(3)}}{\partial W_2}$

We can replace it in our original formula:

$\frac{\partial J(W)}{\partial W_2} = \delta^{(4)} \cdot W_3 \cdot \frac{\partial a^{(3)}}{\partial W_2}$

Now we have to compute $\frac{\partial a^{(3)}}{\partial W_2}$, we know that $a^{(3)}$ depends on $z^{(3)}$ (because $a^{(3)} = tanh(z^{(3)})$) which itself depends on W_2 (because $z^{(3)} = a^{(2)} \cdot W_2$). Using the chain rule we can write:

$\frac{\partial a^{(3)}}{\partial W_2} = \frac{\partial a^{(3)}}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial W_2}$

We can replace it in our original formula:

$\frac{\partial J(W)}{\partial W_2} = \delta^{(4)} \cdot W_3 \cdot \frac{\partial a^{(3)}}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial W_2}$

We differentiate $\frac{\partial a^{(3)}}{\partial z^{(3)}}$, we have:

$\frac{\partial a^{(3)}}{\partial z^{(3)}} = tanh'(z^{(3)}) = 1 - \tanh(Z^{(3)})^2$

We can replace it in our original formula:

$\frac{\partial J(W)}{\partial W_2} = \delta^{(4)} \cdot W_3 \cdot 1 - \tanh(Z^{(3)})^2 \cdot \frac{\partial z^{(3)}}{\partial W_2}$

And we differentiate the last missing term $\frac{\partial z^{(3)}}{\partial W_2}$:

$\frac{\partial z^{(3)}}{\partial W_2} = a^{(2)} \cdot 1 = a^{(2)}$

We can replace it in our original formula:

$\frac{\partial J(W)}{\partial W_2} = \delta^{(4)} \cdot W_3 \cdot 1 - \tanh(Z^{(3)})^2 \cdot a^{(2)}$

We found the second gradient of our function $J(W)$. As you can see, the more you go toward the beginning of the network, the more the differentiation will be long. That's why we introduced the $\delta^{(l)}$ terms where l is the layer number. So that we don't have to differentiate again the first part of the function but directly use $\delta^{(l)}$. For the second gradient we introduce:

$\delta^{(3)} = \delta^{(4)} \cdot W_3 \cdot 1 - \tanh(Z^{(3)})^2$

We now have to find our last gradient $\frac{\partial J(W)}{\partial W_1}$, we will use the same steps as before for the beginning. We begin from:

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

Using the exact same steps as for W_2 we will arrive at:

$$\frac{\partial J(W)}{\partial W_1} = \delta^{(4)} \cdot W_3 \cdot 1 - \tanh(Z^{(3)})^2 \cdot \frac{\partial z^{(3)}}{\partial W_1}$$

As we introduced $\delta^{(3)}$ we can use it:

$$\frac{\partial J(W)}{\partial W_1} = \delta^{(3)} \cdot \frac{\partial z^{(3)}}{\partial W_1}$$

Before we were searching the derivative of $z^{(3)}$ with regards to W_2 and so the derivative was equal to $a^{(2)}$. As a reminder $z^{(3)} = a^{(2)} \cdot W_2$. This time we are searching the derivative of $z^{(3)}$ with regards to W_1 and so W_2 is only a constant, we have:

$$\frac{\partial z^{(3)}}{\partial W_1} = W_2 \cdot \frac{\partial a^{(2)}}{\partial W_1}$$

We can replace it in our original formula:

$$\frac{\partial J(W)}{\partial W_1} = \delta^{(3)} \cdot W_2 \cdot \frac{\partial a^{(2)}}{\partial W_1}$$

We now have to find the derivative of $\frac{\partial a^{(2)}}{\partial W_1}$, we know that $a^{(2)}$ depends on $z^{(2)}$ which itself depends on W_1. As before we can use the chain rule here. We have:

$$\frac{\partial a^{(2)}}{\partial W_1} = \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial W_1}$$

Where:

$$\frac{\partial a^{(2)}}{\partial z^{(2)}} = tanh'(z^{(2)}) = 1 - \tanh(Z^{(2)})^2$$

We replace it in our original formula:

$$\frac{\partial J(W)}{\partial W_1} = \delta^{(3)} \cdot W_2 \cdot 1 - \tanh(Z^{(2)})^2 \cdot \frac{\partial z^{(2)}}{\partial W_1}$$

We have one last term to differentiate, if you remember $z^{(2)} = X \cdot W_1$ as we differentiate with regards to W_1 we have:

$$\frac{\partial z^{(2)}}{\partial W_1} = X \cdot W_1 = X$$

So our last gradient is:

$$\frac{\partial J(W)}{\partial W_1} = \delta^{(3)} \cdot W_2 \cdot 1 - \tanh(Z^{(2)})^2 \cdot X$$

We also introduce the term $\delta^{(2)}$, we have:

$$\delta^{(2)} = \delta^{(3)} \cdot W_2 \cdot 1 - \tanh(Z^{(2)})^2$$

And:

$$\frac{\partial J(W)}{\partial W_1} = \delta^{(2)} \cdot X$$

Here we are, we found the gradient of $J(W)$ with regards to its weights. As a reminder we found:

$$\frac{\partial J(W)}{\partial W_1} = \delta^{(2)} \cdot X$$

$$\delta^{(2)} = \delta^{(3)} \cdot W_2 \cdot 1 - \tanh(Z^{(2)})^2$$

$$\frac{\partial J(W)}{\partial W_2} = \delta^{(3)} \cdot a^{(2)}$$

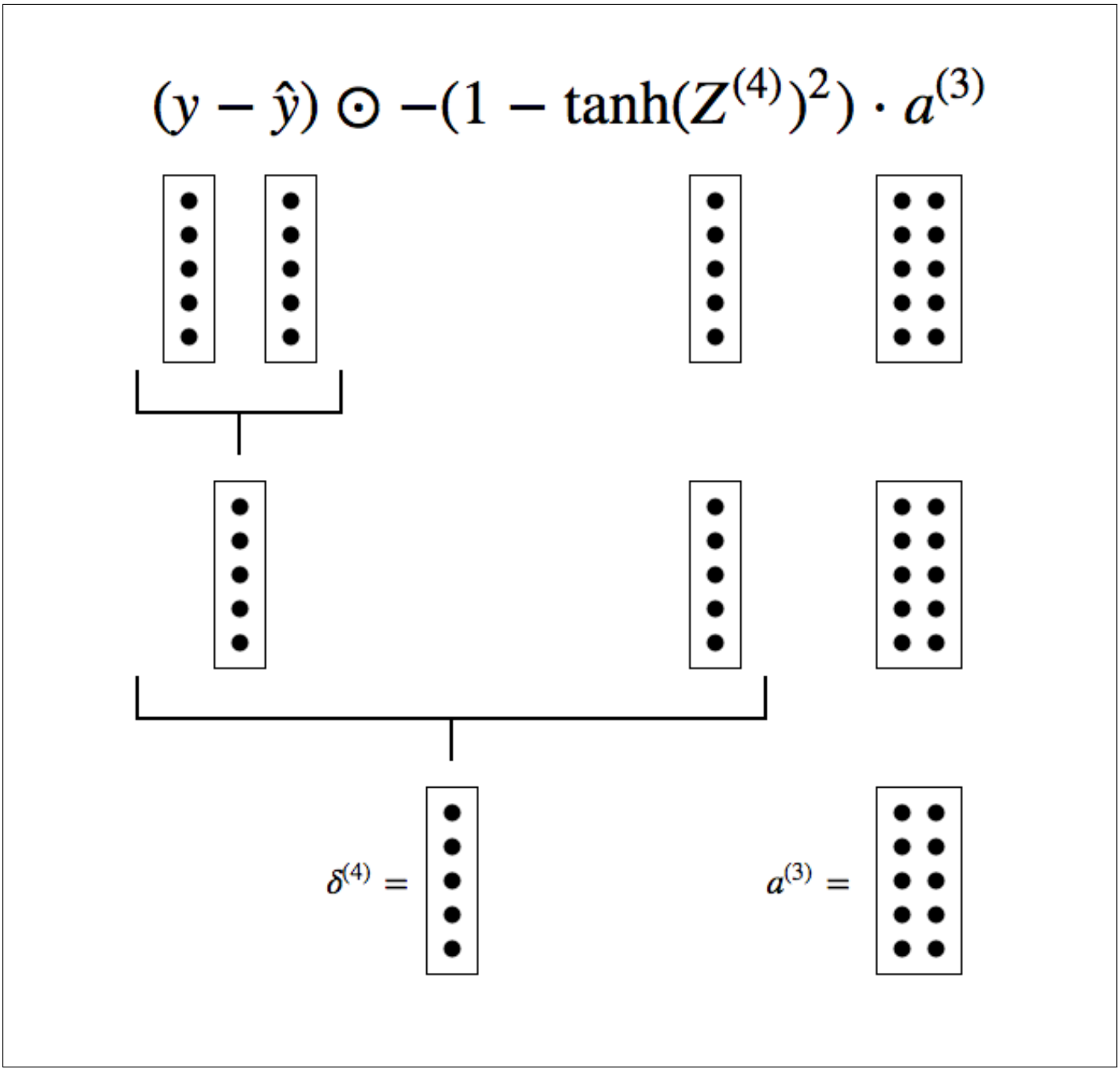
$$\delta^{(3)} = \delta^{(4)} \cdot W_3 \cdot 1 - \tanh(Z^{(3)})^2$$

$$\frac{\partial J(W)}{\partial W_3} = \delta^{(4)} \cdot a^{(3)}$$

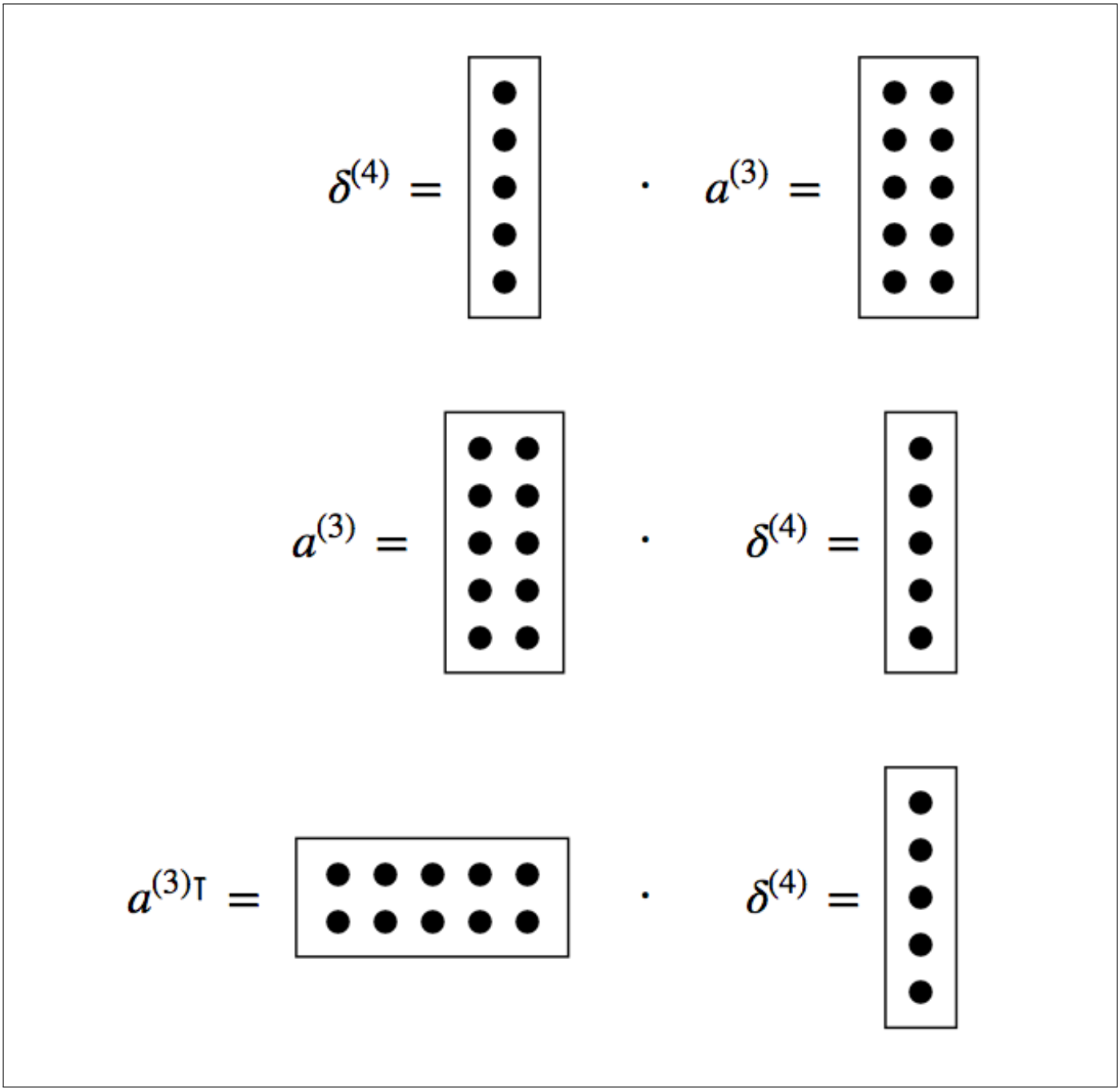
$$\delta^{(4)} = (y - \hat{y}) \cdot -(1 - \tanh(Z^{(4)})^2)$$

If you remember, each gradient will be used to update a weight matrix during one gradient descent iteration. This means that our gradients should have the same size as the weights matrix that will use it. For instance W_3 is a 2×1 matrix so $\frac{\partial J(W)}{\partial W_3}$ must be a 2×1 matrix.

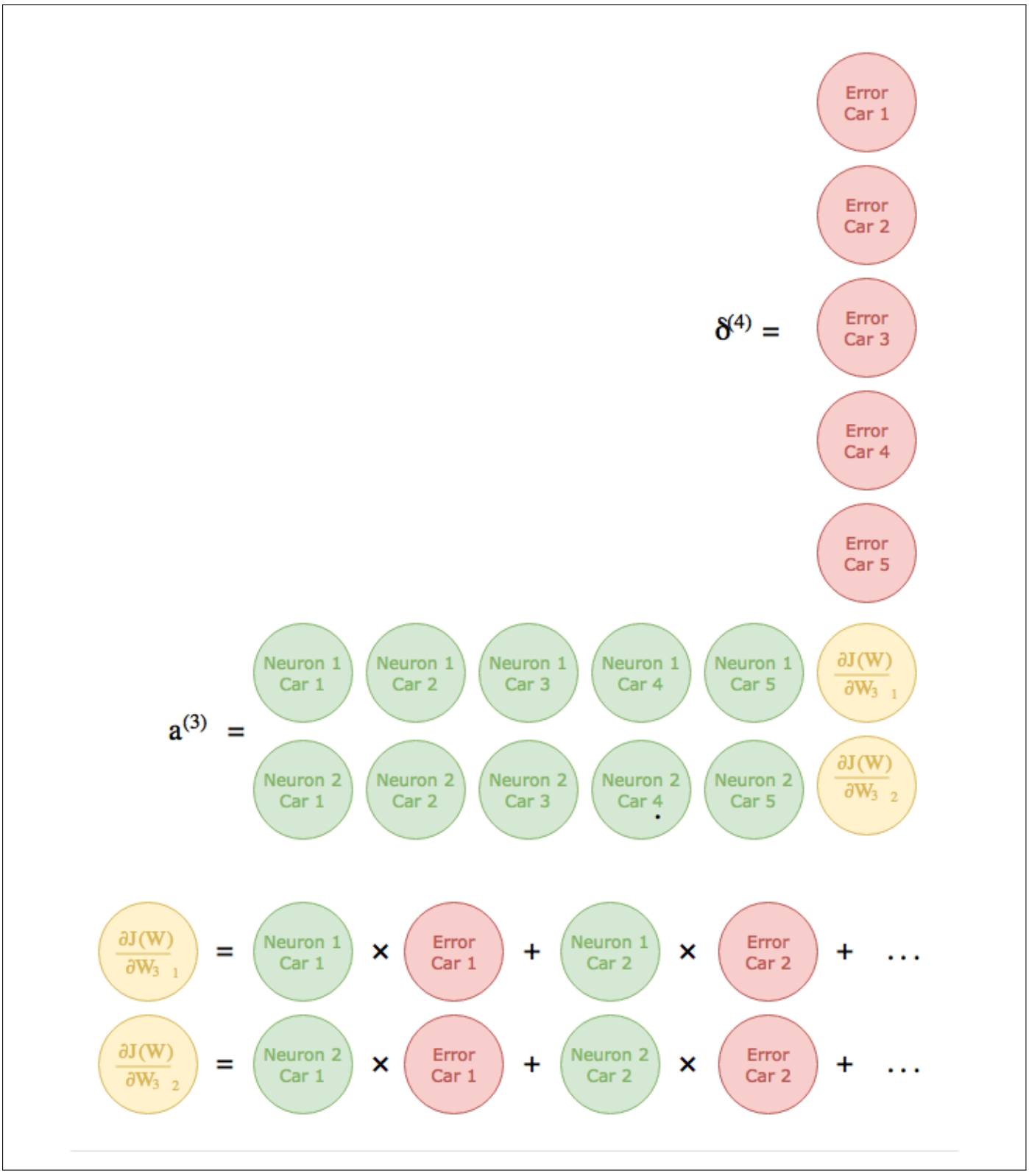
We will detail the dimensions of the matrices used to calculate our gradients so that it appears clearly that the gradients are summed up. It will also be useful to know when were are using element wise or matrix multiplication. Let's say that we have five cars, for $\frac{\partial J(W)}{\partial W_3}$ we found:



The $\odot$ means an element wise multiplication, the $\cdot$ means a matrix multiplication. From now on we will distinguish between the two. Our $\delta^{(4)}$ term has a dimension of 5×1 and $a^{(3)}$ has a dimension of 5×2. As we want the same dimension as W_3 meaning 2×1 there is only one way to achieve that:



By inverting $\delta^{(4)}$ and $a^{(3)}$ and using the transpose of $a^{(3)}$ we are able to get the desired result. $a^{(3)}$ contains the neurons values of the the third layer, two for each car and $\delta^{(4)}$ contains the error for each car. By doing the matrix multiplication we are actually summing the neurons of all the cars where each car neuron value is multiplied by the error for the given car. If you remember we removed the sum from our cost function, this step takes care of the summation of the errors using the matrix multiplication.



So we modify our third gradient so that the summation is handled and the dimension of the matrices are compatible (the little $\intercal$ means transpose):

$$\frac{\partial J(W)}{\partial W_3} = a^{(3)\intercal} \cdot \delta^{(4)}$$

$$\delta^{(4)} = (y - \hat{y}) \odot -(1 - \tanh(Z^{(4)})^2)$$

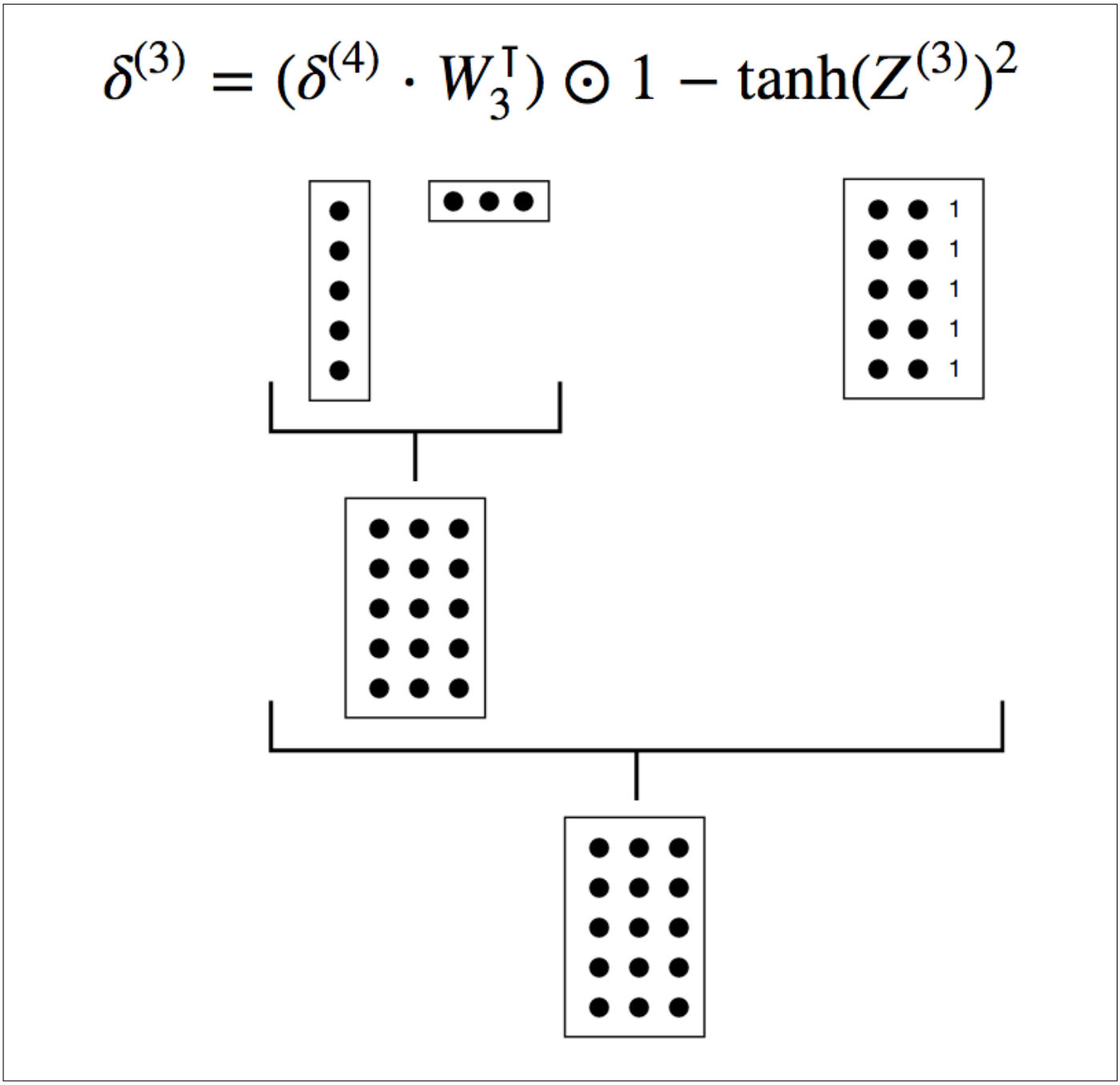
Our $\delta^{(l)}$ will always have the same size as $a^{(l)}$ because it mesures how much a neuron is responsible for any error in our output. This remark is important because during the forward propagation we added bias units to each of our layer, having the effect of increasing the $a^{(l)}$ dimension by one column. $\delta^{(l)}$ will also contain the error of the bias units. These bias units are not linked to the previous layers so when we backpropagate

our $\delta^{(l)}$ into our $(l-1)$ layer we will have to remove the bias error. We will detail the dimensions of the matrices, we also have five cars. Using the same reasoning as before (transpose + inversion) we find that:

$$\frac{\partial J(W)}{\partial W_2} = a^{(2)\intercal} \cdot \delta^{(3)}$$

$$\delta^{(3)} = (\delta^{(4)} \cdot W_3^{\intercal}) \odot 1 - \tanh(Z^{(3)})^2$$

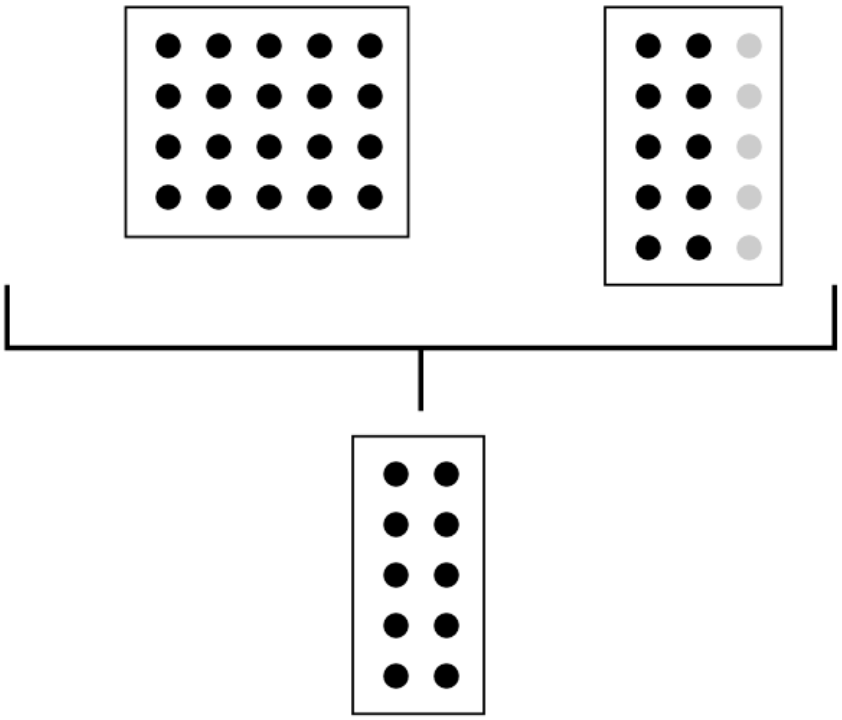
As you can see the order of the calculations follows the output toward input flow. We have:



The third element of W_3 is the bias value, as we also multiply it with $\delta^{(4)}$ the result of our product $\delta^{(4)} \cdot W_3^{\intercal}$ is of size 5×3 whereas $Z^{(3)}$ is of size 5×2. To allow a smooth element wise multiplication we add a column of 1 to $Z^{(3)}$. The result is unchanged and $\delta^{(3)}$ is of size 5×3 like $a^{(3)}$.

Nonetheless the bias is not linked to the previous layer, this means that when backpropagating the error $\delta^{(3)}$ we must remove the part about the bias in the error matrix. We have:

$$\frac{\partial J(W)}{\partial W_2} = a^{(2)\mathsf{T}} \cdot \delta^{(3)}$$



The greyed circles are removed from the $\delta^{(3)}$ before the multiplication, $\frac{\partial J(W)}{\partial W_2}$ has the same size as W_2.

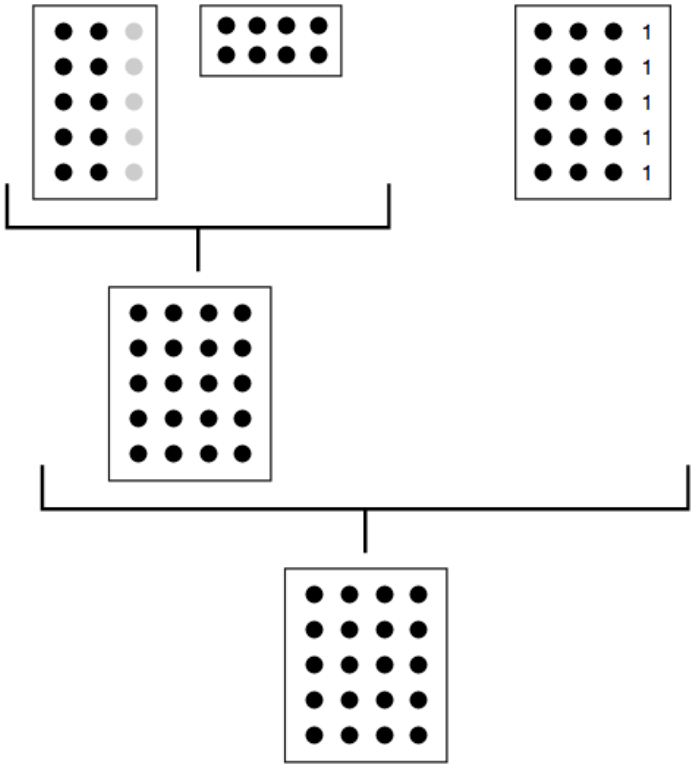
We use the same tricks for $\frac{\partial J(W)}{\partial W_1}$.

$\frac{\partial J(W)}{\partial W_1} = X^\top \cdot \delta^{(2)}$

$\delta^{(2)} = (\delta^{(3)} \cdot W_2^\top) \odot 1 - \tanh(Z^{(2)})^2$

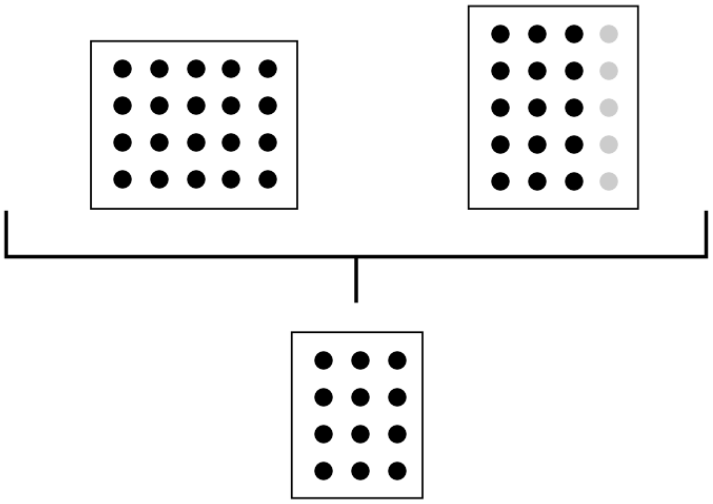
We have:

$$\delta^{(2)} = (\delta^{(3)} \cdot W_2^\intercal) \odot 1 - \tanh(Z^{(2)})^2$$



And:

$$\frac{\partial J(W)}{\partial W_1} = X^\intercal \cdot \delta^{(2)}$$



$\frac{\partial J(W)}{\partial W_1}$ has the same size as W_1.

This may seem difficult but using the dimension analysis and knowing when to remove the bias error and add a 1 column, there is only one way to obtain the same dimension as the weight matrix.

As you may have noticed the partial derivatives follow the same pattern, we did the differentiation manually to see the foundations but we can apply the backpropagation algorithm, for the last layer (l is the index of the last layer) we have:

$$\delta^{(l)} = -(y - a^{(l)}) \odot \sigma'(z^{(l)})$$

For non output layer (l is not the index of the last layer):

$$\delta^{(l)} = ((\delta^{(l+1)} \cdot W_{(l)}^{\top}) \odot \sigma'(z^{(l)}))$$

Then we compute the gradients:

$$\frac{\partial J(W)}{\partial W_{(l)}} = a^{(l)\top} \cdot \delta^{(l+1)}$$

You can stack a bunch of layers and use the backpropagation formula to easily compute the gradients. Let's use our previous examples to compute our gradients and do a gradient descent iteration.

$$\delta^{(4)} = (y - \hat{y}) \odot -(1 - \tanh(Z^{(4)})^2)$$

$$\delta^{(4)} = \begin{bmatrix} 0.45 \\ 0.8 \\ 0.2 \\ 0.5 \\ 0.55 \end{bmatrix} - \begin{bmatrix} 0.202354302599 \\ 0.144933682554 \\ 0.381054078721 \\ 0.262494787219 \\ 0.368818057375 \end{bmatrix} \odot - \begin{bmatrix} 0.95905273622 \\ 0.978994227661 \\ 0.85479778909 \\ 0.931096486683 \\ 0.863973240554 \end{bmatrix}$$

We took care of applying $1 - \tanh(x)^2$ element wise to $Z^{(4)}$ values.

$$\delta^{(4)} = \begin{bmatrix} 0.247645697401 \\ 0.655066317446 \\ -0.181054078721 \\ 0.237505212781 \\ 0.181181942625 \end{bmatrix} \odot - \begin{bmatrix} 0.95905273622 \\ 0.978994227661 \\ 0.85479778909 \\ 0.931096486683 \\ 0.863973240554 \end{bmatrix}$$

$$\delta^{(4)} = \begin{bmatrix} -0.237505283705 \\ -0.641306143515 \\ 0.154764626197 \\ -0.221140269189 \\ -0.156536350099 \end{bmatrix}$$

Our gradient for W3 is:

$$\frac{\partial J(W)}{\partial W_3} = a^{(3)\top} \cdot \delta^{(4)}$$

$$\frac{\partial J(W)}{\partial W_3} = \begin{bmatrix} 0.312422790277 & 0.16852505915 & 0.745005333554 & 0.448044091981 & 0.578398840625 \\ 0.226071339877 & 0.0956597075832 & 0.662175586399 & 0.367961400586 & 0.643703471035 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix} \cdot$$

$$\begin{bmatrix} -0.237505283705 \\ -0.641306143515 \\ 0.154764626197 \\ -0.221140269189 \\ -0.156536350099 \end{bmatrix}$$

$$\frac{\partial J(W)}{\partial W_3} = \begin{bmatrix} -0.25659878177 \\ -0.194693013848 \\ -1.10172342031 \end{bmatrix}$$

For W_2 we have:

$$\delta^{(3)} = (\delta^{(4)} \cdot W_3^\top) \odot 1 - \tanh(Z^{(3)})^2$$

$$\delta^{(3)} = \begin{bmatrix} -0.237505283705 \\ -0.641306143515 \\ 0.154764626197 \\ -0.221140269189 \\ -0.156536350099 \end{bmatrix} \cdot \begin{bmatrix} 0.04 & 0.41 & 0.1 \end{bmatrix} \odot \begin{bmatrix} 0.902392000116 & 0.948891749286 & 0.419974341614 \\ 0.971599304439 & 0.990849220345 & 0.419974341614 \\ 0.444967052977 & 0.561523492777 & 0.419974341614 \\ 0.799256491641 & 0.864604407679 & 0.419974341614 \\ 0.665454781164 & 0.585645841377 & 0.419974341614 \end{bmatrix}$$

As before we took care of applying $1 - \tanh(x)^2$ element wise to $Z^{(3)}$ values. We also concatenated a column of 1 to $Z^{(3)}$ as described before. The 1s became 0.419974341614 when applying $1 - \tanh(x)^2$.

$$\delta^{(3)} = \begin{bmatrix} -0.0095002113482 & -0.0973771663191 & -0.0237505283705 \\ -0.0256522457406 & -0.262935518841 & -0.0641306143515 \\ 0.00619058504786 & 0.0634534967406 & 0.0154764626197 \\ -0.00884561076757 & -0.0906675103676 & -0.0221140269189 \\ -0.00626145400397 & -0.0641799035407 & -0.0156536350099 \end{bmatrix} \odot \begin{bmatrix} 0.902392000116 & 0.94889174928 \\ 0.971599304439 & 0.99084922034 \\ 0.444967052977 & 0.56152349277 \\ 0.799256491641 & 0.86460440767 \\ 0.665454781164 & 0.58564584137 \end{bmatrix}$$

$$\delta^{(3)} = \begin{bmatrix} -0.00857291472002 & -0.092400389689 & -0.00997461251539 \\ -0.0249237041189 & -0.260529453845 & -0.0269332125396 \\ 0.00275460638495 & 0.0356306291187 & 0.0064997171992 \\ -0.0070699118285 & -0.078391529097 & -0.00928732389571 \\ -0.00416671450398 & -0.0375866936086 & -0.00657412505716 \end{bmatrix}$$

Our gradient for W2 is:

$$\frac{\partial J(W)}{\partial W_2} = a^{(2)\top} \cdot \delta^{(3)}$$

$$\frac{\partial J(W)}{\partial W_2} = \begin{bmatrix} -0.0698858903164 & -0.0917413131084 & 0.113508705786 & -0.0449696495836 & 0.34345116481 \\ 0.339033408721 & 0.134185809931 & 0.988329664432 & 0.600545525169 & 0.658975160566 \\ 0.139092447878 & 0.0309900734824 & 0.710404487737 & 0.221278467898 & 0.434961731831 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}$$

$$\begin{bmatrix} -0.00857291472002 & -0.092400389689 \\ -0.0249237041189 & -0.260529453845 \\ 0.00275460638495 & 0.0356306291187 \\ -0.0070699118285 & -0.078391529097 \\ -0.00416671450398 & -0.0375866936086 \end{bmatrix}$$

We took care of removing the last column of $\delta^{(3)}$ as the error on the bias is not backpropagated.

$$\frac{\partial J(W)}{\partial W_2} = \begin{bmatrix} 0.0020851994346 & 0.0250192301883 \\ -0.010520017991 & -0.102917746604 \\ -0.00338471099243 & -0.0293089952796 \\ -0.0419786387864 & -0.433277437121 \end{bmatrix}$$

Finally for W_1 we have:

$$\delta^{(2)} = (\delta^{(3)} \cdot W_2^\intercal) \odot 1 - \tanh(Z^{(2)})^2$$

$$\delta^{(2)} = \begin{bmatrix} -0.00857291472002 & -0.092400389689 \\ -0.0249237041189 & -0.260529453845 \\ 0.00275460638495 & 0.0356306291187 \\ -0.0070699118285 & -0.078391529097 \\ -0.00416671450398 & -0.0375866936086 \end{bmatrix} \cdot \begin{bmatrix} 0.04 & 0.4 & 0.65 & 0.1 \\ 0.78 & 0.45 & 0.23 & 0.1 \end{bmatrix} \odot \begin{bmatrix} 0.995115962335 & 0.88505634777 \\ 0.991583531469 & 0.98199416841 \\ 0.987115773711 & 0.023204474403 \\ 0.997977730616 & 0.6393450722 \\ 0.88204129739 & 0.56575173775 \end{bmatrix}$$

We took care of removing the last column of $\delta^{(3)}$, we also concatenated a column of 1 to $Z^{(2)}$ and applied $1 - \tanh(x)^2$ element wise to $Z^{(2)}$ values.

$$\delta^{(2)} = \begin{bmatrix} -0.0724152205462 & -0.0450093412481 & -0.0268244841965 & -0.0100973304409 \\ -0.204209922164 & -0.127207735878 & -0.0761221820616 & -0.0285453157964 \\ 0.0279020749679 & 0.0171356256574 & 0.00998553884751 & 0.00383852355036 \\ -0.0614281891688 & -0.0381041528251 & -0.0226254943808 & -0.00854614409256 \\ -0.0294842895948 & -0.0185806979255 & -0.0113533039576 & -0.00417534081126 \end{bmatrix} \odot \begin{bmatrix} 0.995115962335 \\ 0.991583531469 \\ 0.987115773711 \\ 0.997977730616 \\ 0.88204129739 \end{bmatrix}$$

$$\delta^{(2)} = \begin{bmatrix} -0.0720615418815 & -0.0398358031806 & -0.0263055187051 & -0.00424061970398 \\ -0.20249119578 & -0.124917254809 & -0.0760490754861 & -0.0119883002078 \\ 0.0275425783201 & 0.000397623186961 & 0.00494609166096 & 0.00161208140083 \\ -0.0613039648226 & -0.0243617023391 & -0.0215176560459 & -0.00358916123861 \\ -0.0260063610469 & -0.0105120621401 & -0.00920535298859 & -0.00175353600822 \end{bmatrix}$$

Our gradient for W_1 is:

$$\frac{\partial J(W)}{\partial W_1} = X^\intercal \cdot \delta^{(2)}$$

$$\frac{\partial J(W)}{\partial W_1} = \begin{bmatrix} 1.4 & 0.4 & 5.4 & 1.5 & 1.8 \\ -1.0 & -1.0 & -1.0 & -1.0 & 1.0 \\ 0.4 & 0.1 & 4.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} -0.0720615418815 & -0.0398358031806 & -0.0263055187051 \\ -0.20249119578 & -0.124917254809 & -0.0760490754861 \\ 0.0275425783201 & 0.000397623186961 & 0.00494609166096 \\ -0.0613039648226 & -0.0243617023391 & -0.0215176560459 \\ -0.0260063610469 & -0.0105120621401 & -0.00920535298859 \end{bmatrix}$$

We took care of removing the last column of $\delta^{(2)}$ as the error on the bias is not backpropagated.

$$\frac{\partial J(W)}{\partial W_1} = \begin{bmatrix} -0.171920111136 & -0.159054126528 & -0.0893845808607 \\ 0.282307763117 & 0.178205075002 & 0.109720805588 \\ -0.0262137489196 & -0.0617093184844 & -0.0290657574213 \\ -0.334320485211 & -0.199229199282 & -0.128131511565 \end{bmatrix}$$

Now that we have our gradients, we can apply one iteration of the gradient descent algorithm to update our weights. We saw previously the update formula:

$$W_{(l)} = W_{(l)} - \alpha \frac{1}{n} \frac{\partial J(W)}{\partial W_{(l)}}$$

We have five cars in our dataset so $n = 5$, we choose a learning rate of $\alpha = 0.1$.

$$W_1 = W_1 - \alpha \frac{1}{n} \frac{\partial J(W)}{\partial W_1}$$

$$W_1 = \begin{bmatrix} 0.01 & 0.05 & 0.07 \\ 0.2 & 0.041 & 0.11 \\ 0.04 & 0.56 & 0.13 \\ 0.1 & 0.1 & 0.1 \end{bmatrix} - 0.1 \odot \frac{1}{5} \odot \begin{bmatrix} -0.171920111136 & -0.159054126528 & -0.0893845808607 \\ 0.282307763117 & 0.178205075002 & 0.109720805588 \\ -0.0262137489196 & -0.0617093184844 & -0.0290657574213 \\ -0.334320485211 & -0.199229199282 & -0.128131511565 \end{bmatrix}$$

$$W_1 = \begin{bmatrix} 0.0134384022227 & 0.0531810825306 & 0.0717876916172 \\ 0.194353844738 & 0.0374358985 & 0.107805583888 \\ 0.0405242749784 & 0.56123418637 & 0.130581315148 \\ 0.106686409704 & 0.103984583986 & 0.102562630231 \end{bmatrix}$$

As you can see our weights matrix has been updated, some values are bigger, others smaller, we are moving in our space of solutions toward the best set of weights.

We do the exact same thing for W_2 and W_3:

$$W_2 = W_2 - \alpha \frac{1}{n} \frac{\partial J(W)}{\partial W_2}$$

$$W_2 = \begin{bmatrix} 0.04 & 0.78 \\ 0.4 & 0.45 \\ 0.65 & 0.23 \\ 0.1 & 0.1 \end{bmatrix} - 0.1 \odot \frac{1}{5} \odot \begin{bmatrix} 0.0020851994346 & 0.0250192301883 \\ -0.010520017991 & -0.102917746604 \\ -0.00338471099243 & -0.0293089952796 \\ -0.0419786387864 & -0.433277437121 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.0399582960113 & 0.779499615396 \\ 0.40021040036 & 0.452058354932 \\ 0.65006769422 & 0.230586179906 \\ 0.100839572776 & 0.108665548742 \end{bmatrix}$$

$$W_3 = W_3 - \alpha \frac{1}{n} \frac{\partial J(W)}{\partial W_3}$$

$$W_3 = \begin{bmatrix} 0.04 \\ 0.41 \\ 0.1 \end{bmatrix} - 0.1 \odot \frac{1}{5} \odot \begin{bmatrix} -0.25659878177 \\ -0.194693013848 \\ -1.10172342031 \end{bmatrix}$$

$$W_3 = \begin{bmatrix} 0.0451319756354 \\ 0.413893860277 \\ 0.122034468406 \end{bmatrix}$$

By doing the forward propagation, backward propagation and weights update in a loop you have an algorithm that learn.

Chapter 7

Gradient Checking

To do the backpropagation we found the derivative of our cost function. It is easy to do a mistake during the differentiation and while working ostensibly fine our neural network will perform poorly. We used the chain rule to find our gradients, this is called analytical differentiation.

Nonetheless the original definition of a derivative is the following: $\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$ The idea is that for a small enough h value the previous formula approximates correctly the value of the derivative of the function f. Because if you zoom enough on a tiny part of the graph of the function, it will appear linear, and so approximate the tangent of the graph. This is called numerical differentiation. As you can see we are computing twice $f(x)$, that's why the numerical differentiation is slower and not used (even if it is less error prone).

The idea is to use the numerical differentiation to check that our analytical differentiation implementation is correct. We will not use the strict definition but rather the centered formula (works better):

$\frac{df(x)}{dx} = \frac{f(x+h)-f(x-h)}{2h}$

In total we have three matrices containing the weights. We will test one weight at a time. We will proceed as follow.

We do a forward and backward propagation. We save our gradients into a vector V_1. For each weight: We compute f(weight + h). We compute f(weight - h). We compute the centered formula and save the value into a vector V_2. The difference between V_1 and V_2 should be less than 10^-8. The function that we differentiated during the backpropagation is our cost function:

$J(W) = \sum_1^n \frac{1}{2}(y - \hat{y})^2$

Where $\hat{y}$ is in reality $a^{(4)}$.

During the backpropagation we found our gradients equal to:

$$\frac{\partial J(W)}{\partial W_1} = \begin{bmatrix} -0.171920111136 & -0.159054126528 & -0.0893845808607 \\ 0.282307763117 & 0.178205075002 & 0.109720805588 \\ -0.0262137489196 & -0.0617093184844 & -0.0290657574213 \\ -0.334320485211 & -0.199229199282 & -0.128131511565 \end{bmatrix}$$

$$\frac{\partial J(W)}{\partial W_2} = \begin{bmatrix} 0.0020851994346 & 0.0250192301883 \\ -0.010520017991 & -0.102917746604 \\ -0.00338471099243 & -0.0293089952796 \\ -0.0419786387864 & -0.433277437121 \end{bmatrix}$$

$$\frac{\partial J(W)}{\partial W_3} = \begin{bmatrix} -0.25659878177 \\ -0.194693013848 \\ -1.10172342031 \end{bmatrix}$$

We save them into a vector $V^{(1)}$ of size 23×1. These gradients have been computed during the forward pass with the analytical differentiation.

As you will see later in the code I am using a perturbation vector, I will keep the explanation simple and don't talk about it.

The weights that we used are:

$$W_1 = \begin{bmatrix} 0.0134384022227 & 0.0531810825306 & 0.0717876916172 \\ 0.194353844738 & 0.0374358985 & 0.107805583888 \\ 0.0405242749784 & 0.56123418637 & 0.130581315148 \\ 0.106686409704 & 0.103984583986 & 0.102562630231 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.0399582960113 & 0.779499615396 \\ 0.40021040036 & 0.452058354932 \\ 0.65006769422 & 0.230586179906 \\ 0.100839572776 & 0.108665548742 \end{bmatrix}$$

$$W_3 = \begin{bmatrix} 0.0451319756354 \\ 0.413893860277 \\ 0.122034468406 \end{bmatrix}$$

We disturb the first weight by adding a small value, we choose $h = 10^{-4}$.

$$W_1 = \begin{bmatrix} 0.0134384022227 + 0.0001 & 0.0531810825306 & 0.0717876916172 \\ 0.194353844738 & 0.0374358985 & 0.107805583888 \\ 0.0405242749784 & 0.56123418637 & 0.130581315148 \\ 0.106686409704 & 0.103984583986 & 0.102562630231 \end{bmatrix}$$

All the other weights stay the same. We do a forward propagation using the weights disturbed (just one value has been disturbed). We obtain:

$$a^{(4)} = \begin{bmatrix} 0.202395039754 \\ 0.144946047163 \\ 0.381136194829 \\ 0.262533502845 \\ 0.368843890153 \end{bmatrix}$$

We compute the sum of the costs (square the cost before summing):

$$J(W) = \frac{1}{2}\sum\left(\begin{bmatrix}0.45\\0.8\\0.2\\0.5\\0.55\end{bmatrix} - \begin{bmatrix}0.202395039754\\0.144946047163\\0.381136194829\\0.262533502845\\0.368843890153\end{bmatrix}\right)^2$$

$$J(W) = loss2 = 0.30621105$$

We disturb again the first weight by removing the small value h.

$$W_1 = \begin{bmatrix}0.0134384022227 - 0.0001 & 0.0531810825306 & 0.0717876916172\\0.194353844738 & 0.0374358985 & 0.107805583888\\0.0405242749784 & 0.56123418637 & 0.130581315148\\0.106686409704 & 0.103984583986 & 0.102562630231\end{bmatrix}$$

All the other weights stay the same. We do a forward propagation using the weights disturbed (just one value has been disturbed). We obtain:

$$a^{(4)} = \begin{bmatrix}0.202313563549\\0.144921317917\\0.380971901463\\0.262456067958\\0.368792216736\end{bmatrix}$$

We also compute the sum of the costs:

$$J(W) = \frac{1}{2}\sum\left(\begin{bmatrix}0.45\\0.8\\0.2\\0.5\\0.55\end{bmatrix} - \begin{bmatrix}0.202313563549\\0.144921317917\\0.380971901463\\0.262456067958\\0.368792216736\end{bmatrix}\right)^2$$

$$J(W) = loss1 = 0.30624543$$

Then we compute the numerical gradient by doing:

$$V_1^{(2)} = \frac{(loss2 - loss1)}{(2 * h)}$$

$$V_1^{(2)} = -0.17192014$$

This is the first weight computed using the numerical differentiation. Using the analytical differentiation we found -0.171920111136. As you can see they are almost identical.

We repeat the process for each weight until we have computed our 23 numerical gradients into the vector $V_1^{(2)}$. We then want to compare our analytical gradients with our numerical gradients. To quantify the difference we divide the norm of the difference by the norm of the sum of the vectors we would like to compare. Typical results should be on the order of 10^-8 or less if we've computed our gradient correctly.

$$\frac{|V^{(1)} - V^{(2)}|}{|V^{(1)} + V^{(2)}|} = 1.15700817288e-08$$

As you can see the difference is on the order of 10^-8 this means that our backward propagation computes correctly the gradients $\nabla(J(W))$. If it is not the case, it means that there is an error in the backpropagation

algorithm and you have to debug it, good luck, as you will see, you sometimes lose a great amount of time because of a little mistake.

Chapter 8

Regularization

One last step is missing, the regularization. During the training our weights will evolve in order to predict correctly the price of a car from its attributes. Nonetheless it could happen that the weights are perfectly predicting the price for the training data but can't generalize for an unseen car. This problem is called overfitting. During the training steps our network will display a small loss but when testing it with unseen data it will display a larger loss. It means that our weights are well suited for the training data only.

Regularization is useful when the network has a lot of parameters and not enough data. Enough data is at least ten times the number of parameters. Our network has 23 parameters so we need at least 230 cars. As our dataset of cars has more than 9k cars, overfitting is not really a problem for us, nonetheless we will solve the overfitting problem as it regularly occurs.

One way to reduce overfitting is called regularization. The idea is to penalize large weights by modifying our cost function. There are several way to do regularization, the most used are L1, L2, max norm and dropout. We will use the L2 regularization because it is the most common. It is pretty simple, we only add one term to our cost function:

$J(W) = \sum_1^n \frac{1}{2}(y - \hat{y})^2 - \frac{1}{2}\lambda \sum_1^3 W_{(n)}^2$

The λ is a hyper parameter that will change the impact of the regularization, $\lambda = 1$ means a strong regularization whereas $\lambda = 0.001$ means a light regularization. We sum all the weights squared and we add this sum times lambda to the cost function. If the weights are high, the regularization term will be high and the cost will increase, conversely if the weights are low the regularization term will be low.

Because of that, the weights will be smoothed over all the features. If a feature has a strong impact on the price, let's say the age, the weights corresponding to the age should be high. As we regularize it will not happen. As for the cost function, the regularization term contains $\frac{1}{2}$ to ease things during the differentiation. Previously we differentiated our cost function in order to find our gradients. We did not take into account the regularization term, but the differentiation is easy.

When we expand our regularization sum we have:

$\frac{1}{2}\lambda \sum_1^3 W_{(n)}^2 = \frac{1}{2}\lambda(W_{(1)}^2 + W_{(2)}^2 + W_{(3)}^2)$

When differentiating our cost function w.r.t $W_{(3)}$, we found:

$\frac{\partial J(W)}{\partial W_3} = a^{(3)\top} \cdot \delta^{(4)}$

If we differentiate the regularization term w.r.t $W_{(3)}$ we have:

$\frac{1}{2}\lambda(W_{(1)}^2 + W_{(2)}^2 + W_{(3)}^2)$

$\frac{1}{2}\lambda(0 + 0 + W_{(3)}^2)$

$2 \times \frac{1}{2}\lambda W_{(3)}$

$\lambda W_{(3)}$

If we add the regularization, our gradient w.r.t $W_{(3)}$ is:

$\frac{\partial J(W)}{\partial W_3} = a^{(3)\intercal} \cdot \delta^{(4)} - \lambda \cdot W_{(3)}$

The process is the same for the gradient w.r.t each weight matrix. We remove from the gradient $\lambda \cdot W_{(n)}$. This means that during each gradient descent iteration, using the L2 regularization, each weight will linearly decay towards zero.

When building a model, here is the list of things you have to choose:

The number of hidden layers The number of neurons for each hidden layer The activation function The cost function The optimization algorithm The learning rate The type of regularization The regularization rate The number of gradient descent steps The way of evaluating the accuracy of the network We saw the theoritical part of a deep neural network, there are other types of neural network that will be the topic of other blog posts. To name a few: Convolutionnal Neural Network, Long Short Term Memory, Generative Adversarial Networks...