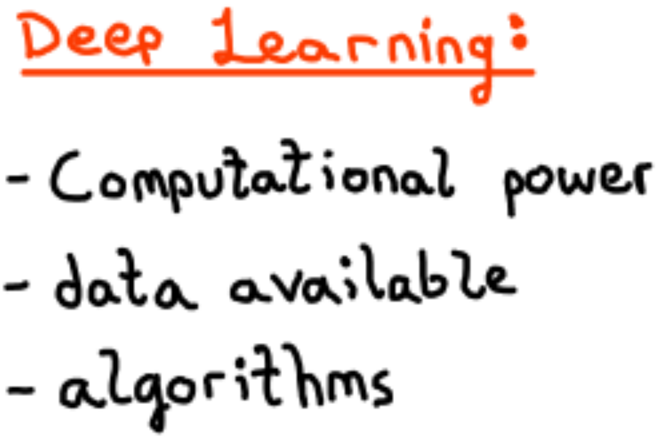
Lecture 11 [Introduction to Neural Networks]

Deep Learning:

Deep learning is a set of techniques that is a subset of ***machine learning*** and it's one of the growing techniques that have been used in the industry specifically for problems in ***computer vision***, ***natural language processing***, and ***speech recognition***. The reason it came to work very well is primarily the new computational methods. ***Deep learning*** is really really computationally expensive and people had to find techniques in order to parallelize the code and use ***GPU***s specifically in order to be able to compute the computations in ***deep learning***. The second reason is the data available has been growing after the internet bubble (the digitilization of the world), so now people have access to large amounts of data and this type of algorithm has the specificity of being able to learn a lot when there is a lot of data. These models are very flexible and the more data you give them, the more they will be able to understand the salient feature of the data. The last reason is algorithms. People have come up with new techniques in order to use the data, use the computation power, and build models

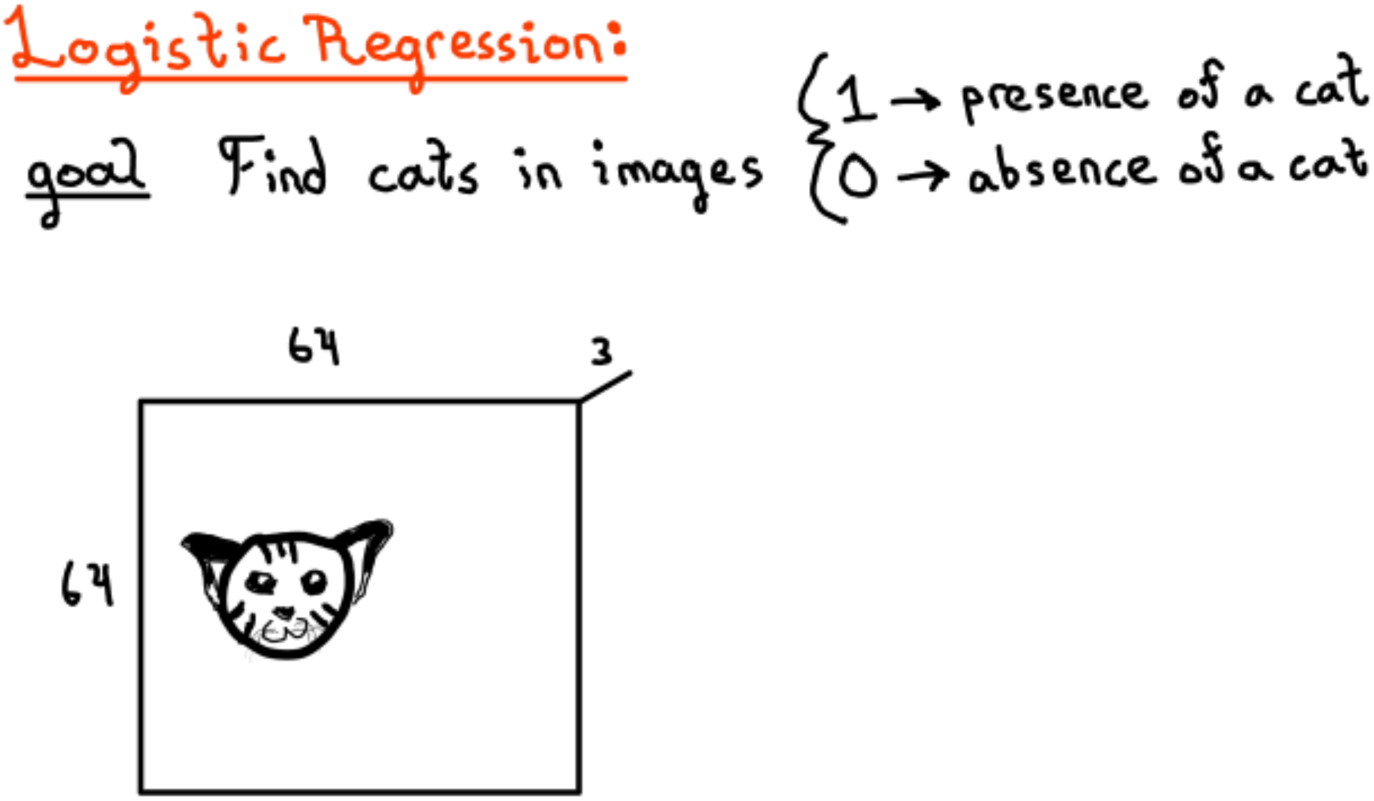


Deep Learning:

- Computational power
- data available
- algorithms

Logistic Regression:

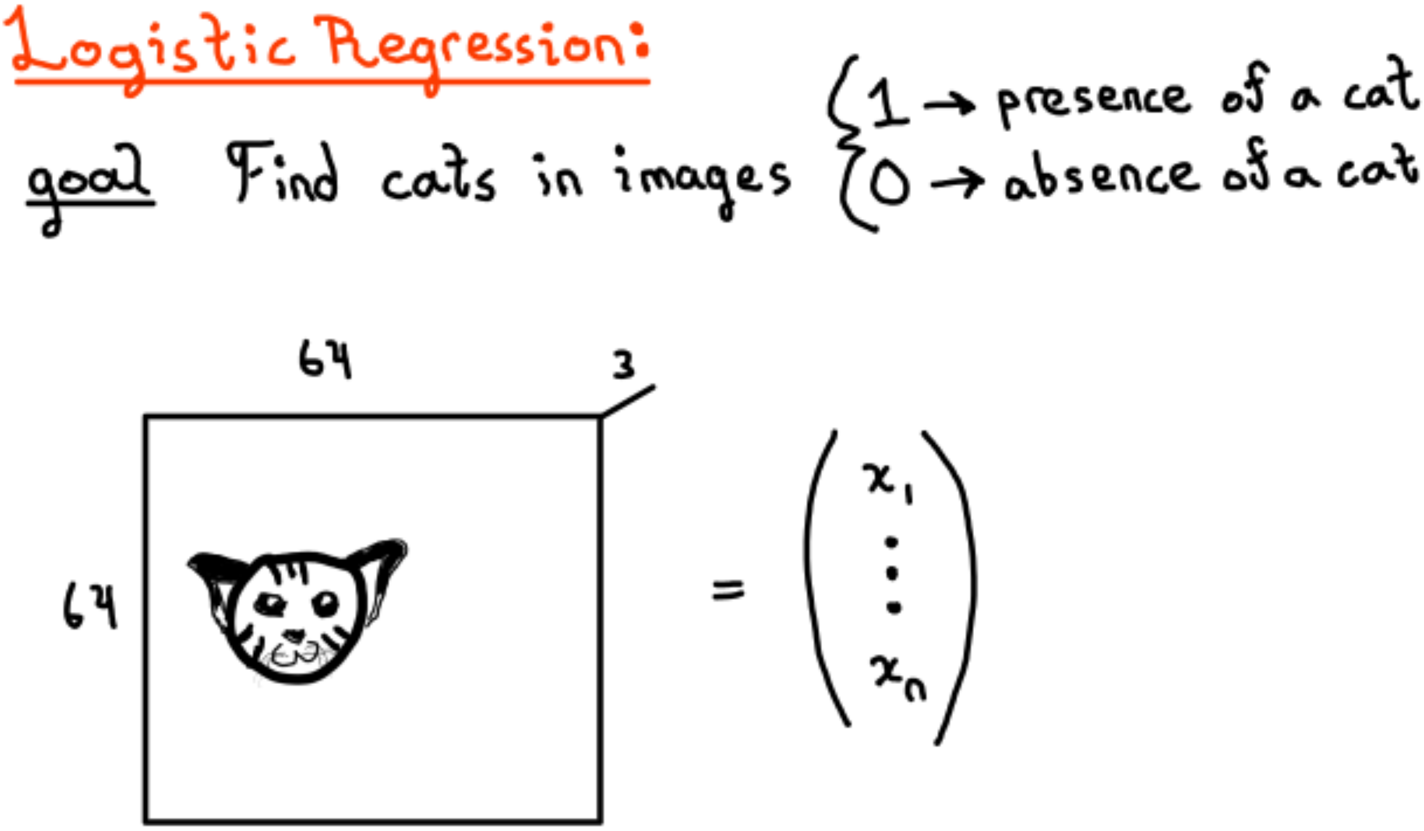
We're going to set a classification goal where we are goint to find cats in images, meaning ***binary classification***. For now, we'll say that we're constrained to the fact that there is at maximum one cat per image, there's no more. If you had to draw the ***logistic regression*** model, that's what you'd do. You would take an image of a cat, and in computer science you know that images can be represented as 3D matrices, so if this is a color image of size **64** by **64**, you have **3** numbers to represent those pixels for the ***RGB channel*** (***red***, ***green***, ***blue***). So this image is actually of size **64x64x3**



Logistic Regression:

goal Find cats in images $\begin{cases} 1 \rightarrow \text{presence of a cat} \\ 0 \rightarrow \text{absence of a cat} \end{cases}$

The first thing we will do in order to use ***logistic regression*** to find if there is a cat in this image is we're going to flatten this image into a vector. We're going to take all the numbers in this matrix (the image) and flatten them in a vector. Just an ***image to vector*** operation

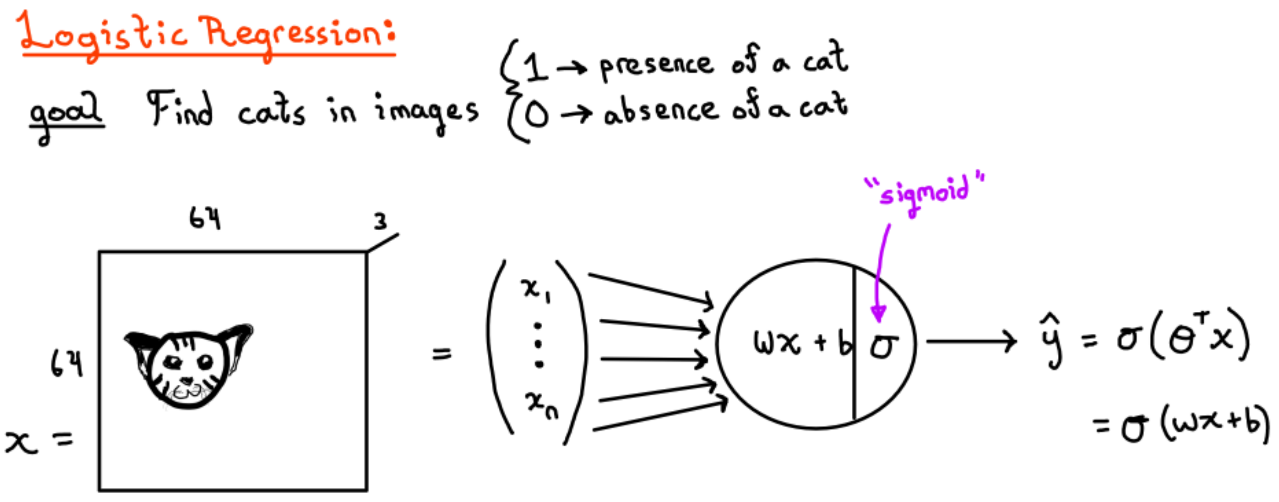


Logistic Regression:

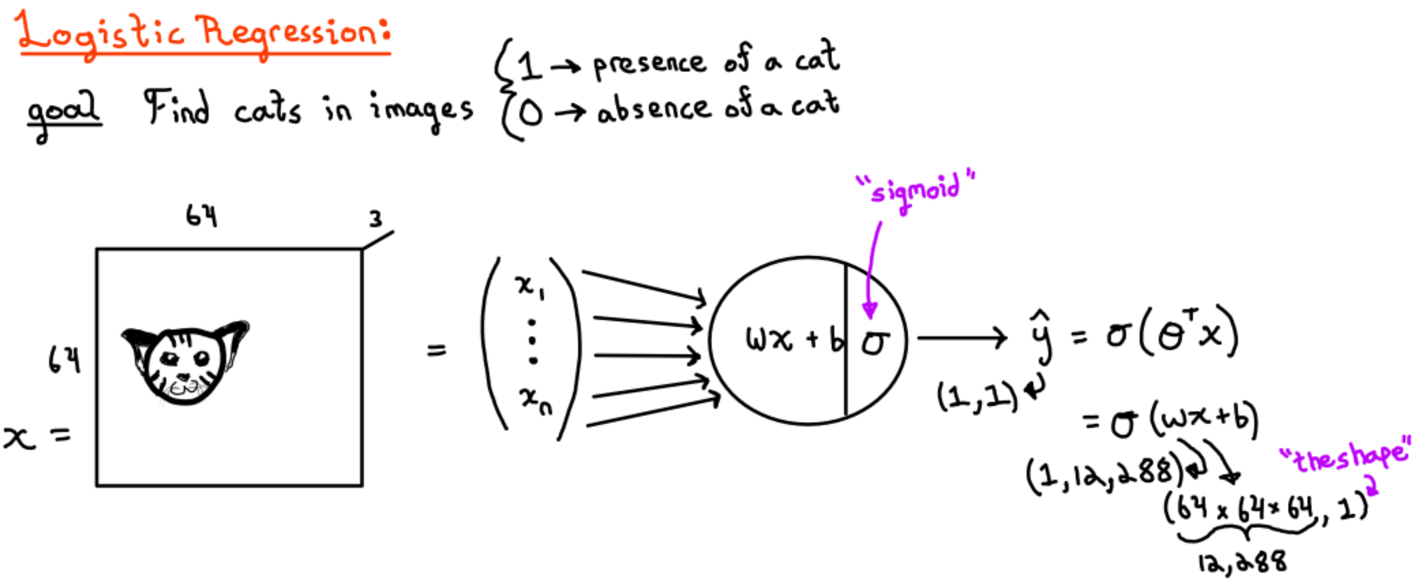
goal Find cats in images $\begin{cases} 1 \rightarrow \text{presence of a cat} \\ 0 \rightarrow \text{absence of a cat} \end{cases}$

$$= \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

Now we can use ***logistic regression*** because we have a vector input. So we're going to take all of these and push them in an operation, that we call the ***logistic operation***, which has one part that is **wx+b** where **x** is going to be the image, and the second part is going to be the ***sigmoid*** (the ***sigmoid function*** is a function that takes a number between **-∞** and **+∞** and maps it between **0** and **1** ; it's very convenient for classification problems). The output will be called **y^** which is **$\sigma(\Theta^T x)$** but here we will just separate the notation into **w** and **b**

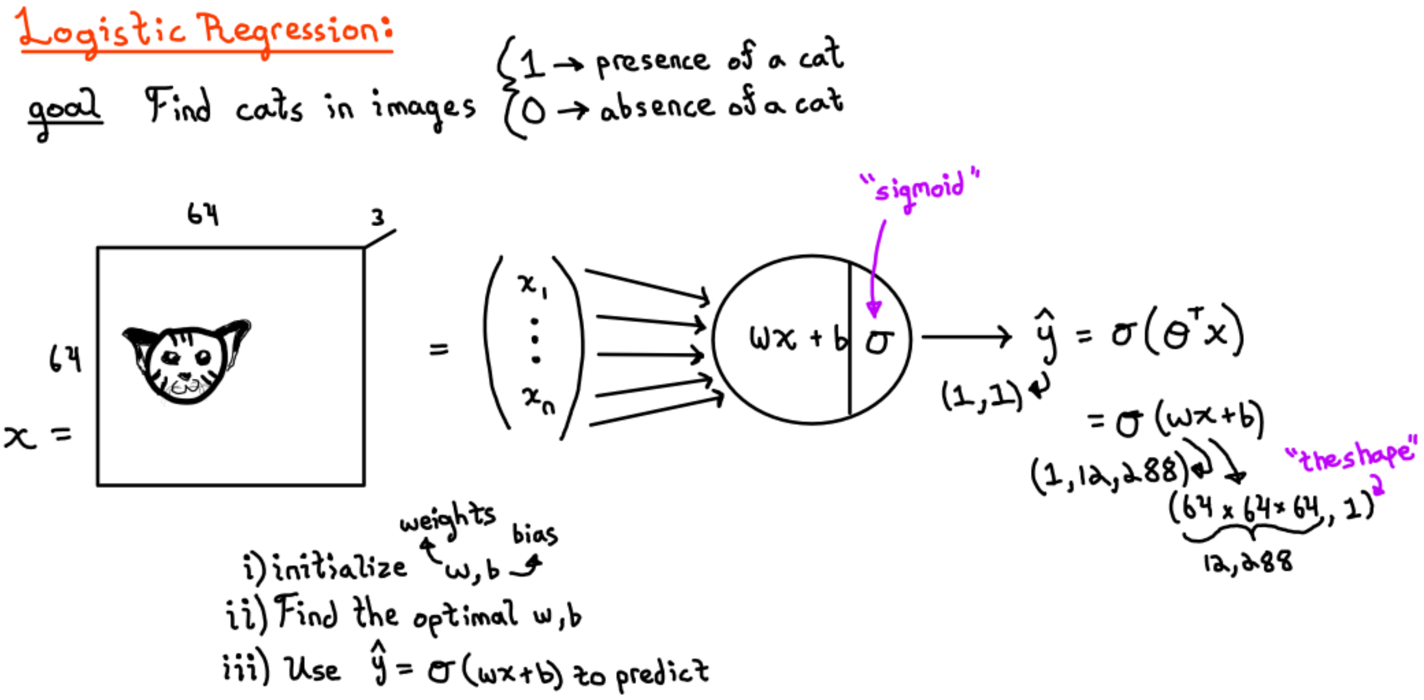


The shape of the vector matrix **w** is a **64x64x3**, a ***column vector***. So the shape of **x** is going to be **64x64x3** times **1** and because we want **y^** to be **1x1**, the **w** has to be **1x12288**



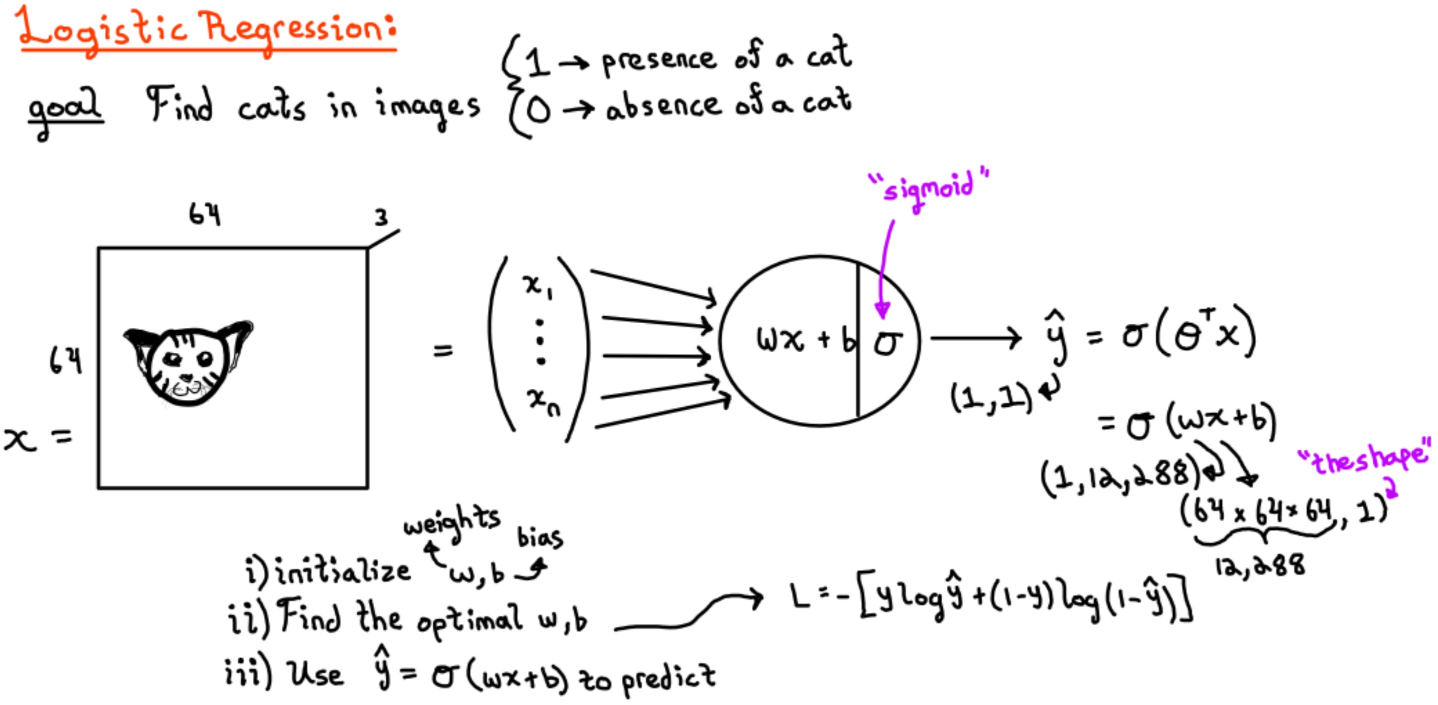
We have a ***row vector*** as our parameter. We're just changing the notations of the ***logistic regression*** that we have seen so far. Once we have this model, we need to train it and the process of training is that first we will initialize our parameters, using the specific vocabulary of ***weights*** and ***bias***. So we're going to find the right **w** and the right **b** in order to be able to use this model correctly

Once we initialize them, we will optimize them, and after we found the optimal **w** and **b**, we will use them to predict



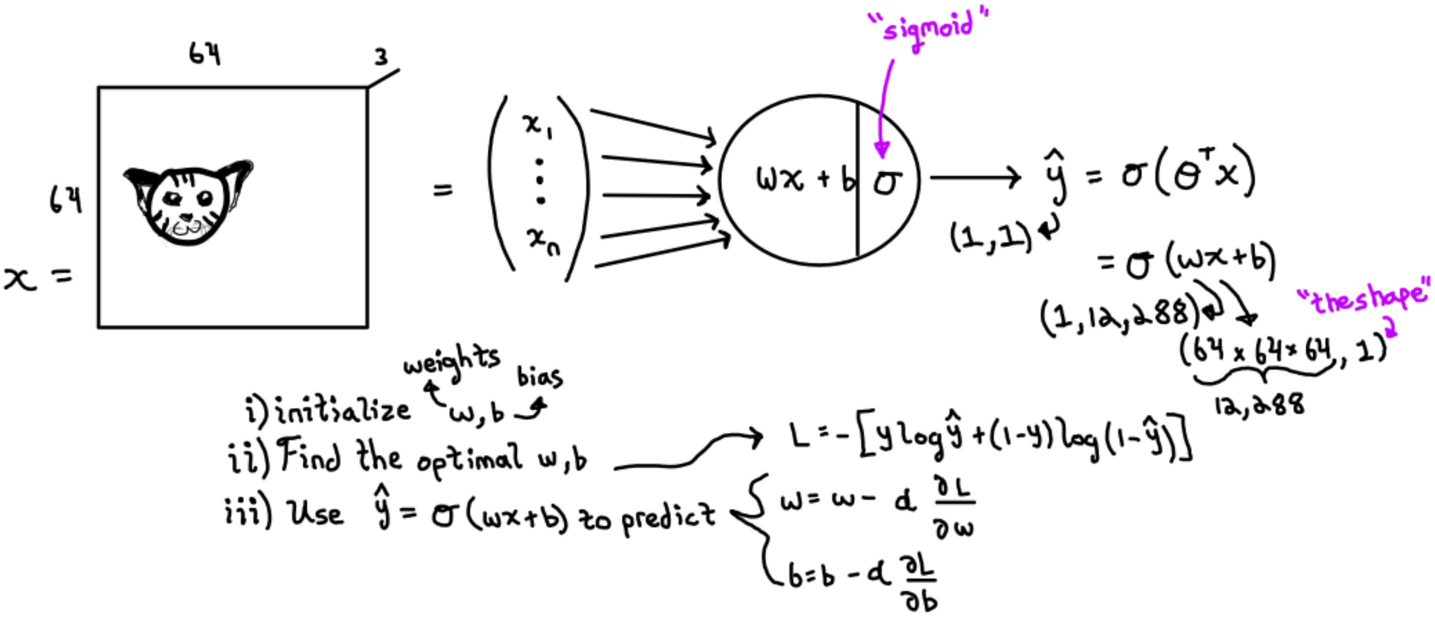
Finding the optimal **w** and **b** means defining your ***loss function***, which is the objective. In machine learning we often have this specific problem where you have a function that you know you want to find, the ***network function***, but you don't know the values of its parameters and in order to find them, you're going to use a ***proxy*** that is going to be your ***loss function***. If you manage to minimize the ***loss function***, you will find the right parameters

So, you define a ***loss function*** that is the ***logistic loss***:



This comes from a ***maximum likelihood estimation***, starting from a ***probabilistic model***

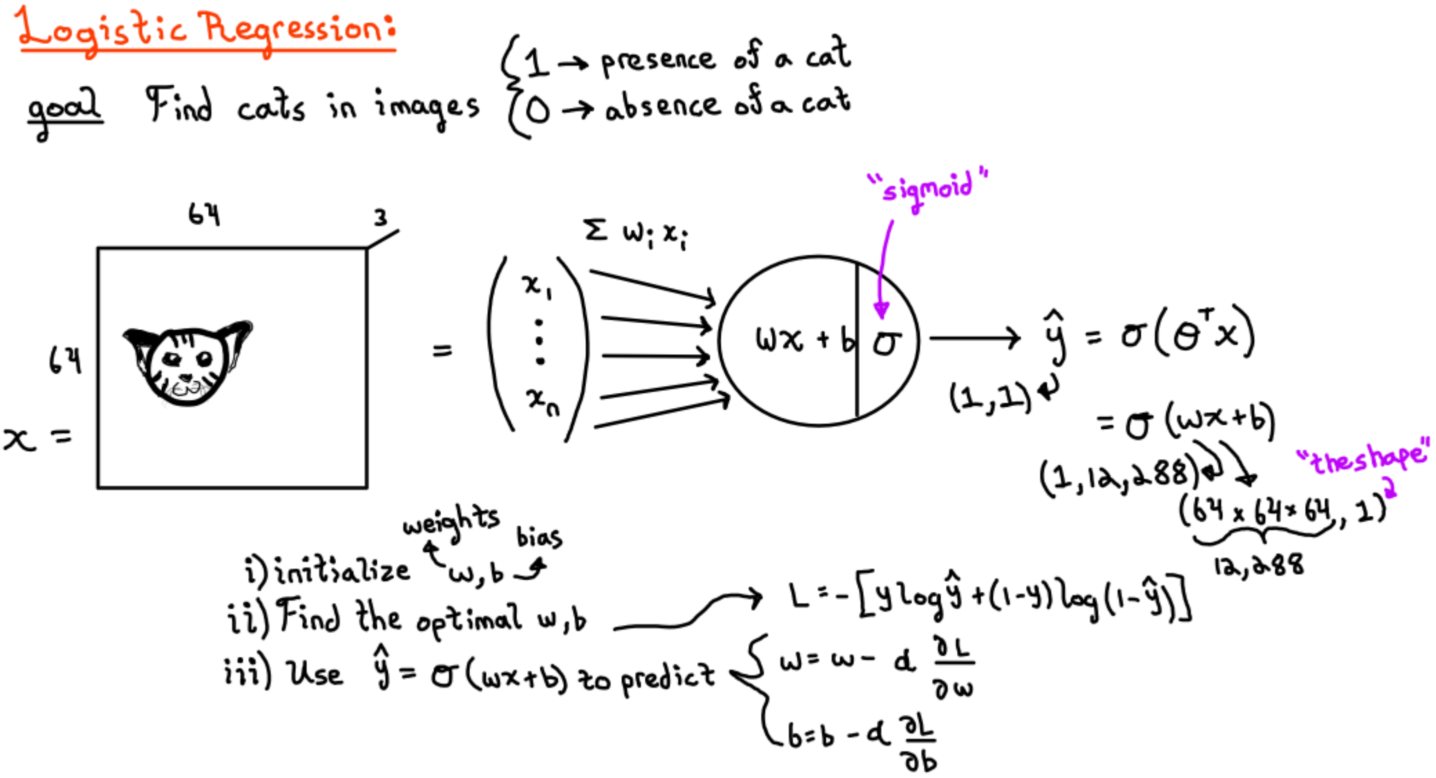
The idea is how can we minimize this function? We want to minimize because we put the minus sign there. We want to find **w** and **b** that minimize this function and we're going to use a ***gradient descent*** algorithm, which means we're going to iteratively compute the derivative of the ***loss*** with respect to our parameters. And at every step, we will update them to make this ***loss function*** go a little down at every iterative step


"sigmoid"
64
3
64
x =
x_1
x_n
=
wx + b
σ
$\hat{y} = \sigma(\theta^T x)$
(1,1)
$= \sigma(wx+b)$
(1,12,288)
"the shape"
$(64 \times 64 \times 64, 1)$
12,288
i) initialize w, b
weights
bias
ii) Find the optimal w,b
$L = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$
iii) Use $\hat{y} = \sigma(wx+b)$ to predict
$w = w - \alpha \frac{\partial L}{\partial w}$
$b = b - \alpha \frac{\partial L}{\partial b}$

In terms of implementation, this is a ***for loop***. You will loop over a certain number of iteration and at every point you will compute the derivative of the ***loss*** with respect to your parameters

#*# The sigmoid function has a derivate of σ' = σ * (1 - σ)

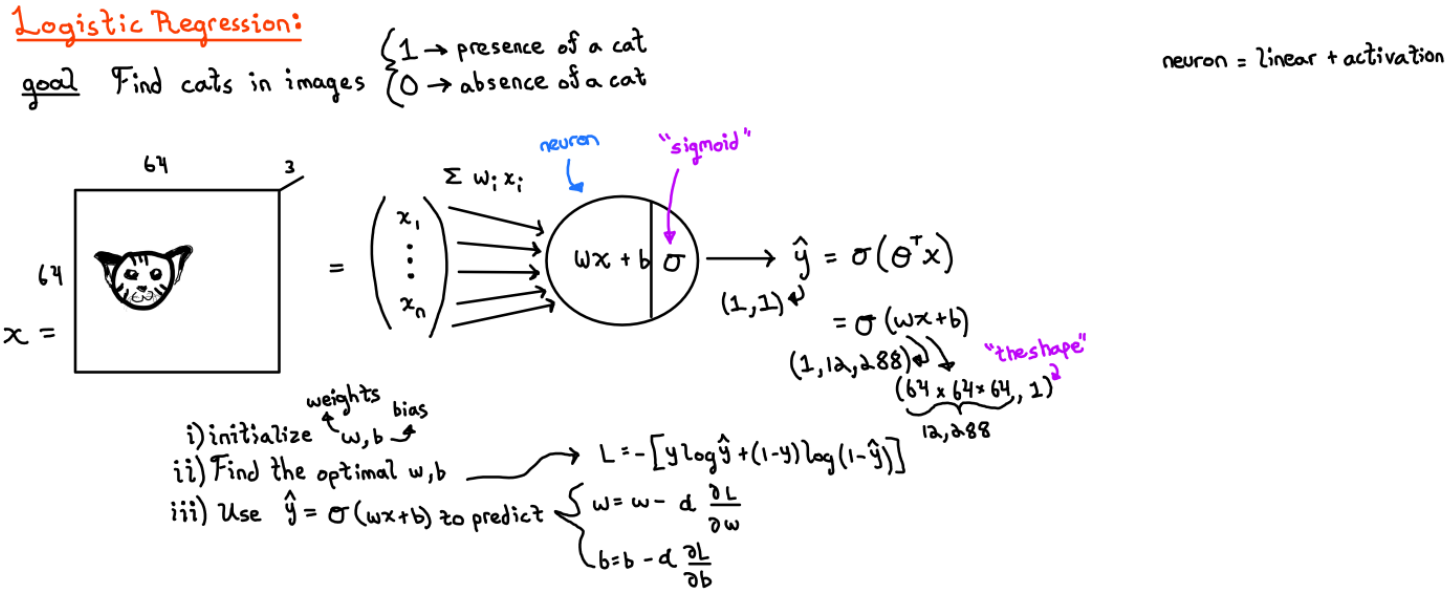
How many parameters does this ***logistic regression*** model have? It has **12,289** parameters, **12,288 *weights*** and **1 *bias***. You can quickly count it by just counting the number of edges on the drawing plus **1**. Every ***circle*** has a ***bias***, every ***edge*** has a ***weight*** because ultimately you could re-write the operation like this:


Logistic Regression:
goal Find cats in images
1 → presence of a cat
0 → absence of a cat
"sigmoid"
64
3
64
x =
$\Sigma w_i x_i$
x_1
x_n
=
wx + b
σ
$\hat{y} = \sigma(\theta^T x)$
(1,1)
$= \sigma(wx+b)$
(1,12,288)
"the shape"
$(64 \times 64 \times 64, 1)$
12,288
i) initialize w, b
weights
bias
ii) Find the optimal w,b
$L = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$
iii) Use $\hat{y} = \sigma(wx+b)$ to predict
$w = w - \alpha \frac{\partial L}{\partial w}$
$b = b - \alpha \frac{\partial L}{\partial b}$

It means every ***weight*** corresponds to an ***edge***, so that's another way to count it

We're starting with not too many parameters, and one thing that we notice is that the number of parameters of our model depends on the size of the input. We probably don't want that at some point, so we're going to change it later on

There are two equations to remember. The first one is **neuron = linear + activation**. This is the vocabulary we will use in ***neural networks***. We define a ***neuron*** as an operation that has two parts, one ***linear*** part and one ***activation*** part and it's exactly that. This is actually a neuron:



We have a ***linear*** part, **wx + b**, and then we take the output of this ***linear*** part and we put it in an ***activation***, that in this case, is the ***sigmoid function***. It can be other functions

The second equation we want to set now is the model

(Equation1) neuron = linear + activation

(Equation2) model = architecture + parameters

This means we're trying to train a ***logistic regression*** in order to be able to use it. We need an ***architecture*** (which is the ***1-neuron neural network*** we defined above) and the ***parameters*** (***w*** and ***b***)

So basically, when people say "we've shipped a model", like in the industry, what they're saying is that they found the right parameters with the right architecture. They have two files and these two files are predicting a bunch of things; one ***parameter file*** and one ***architecture file***

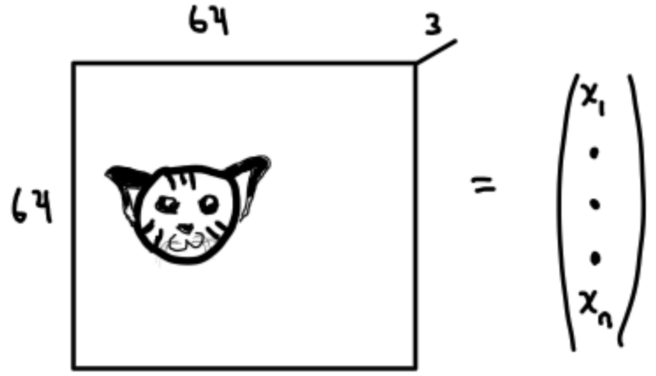
This was just to set up the problem with logistic regression so now we're going to try to set a new goal:

goal 2: Find cat/lion/iguana in images

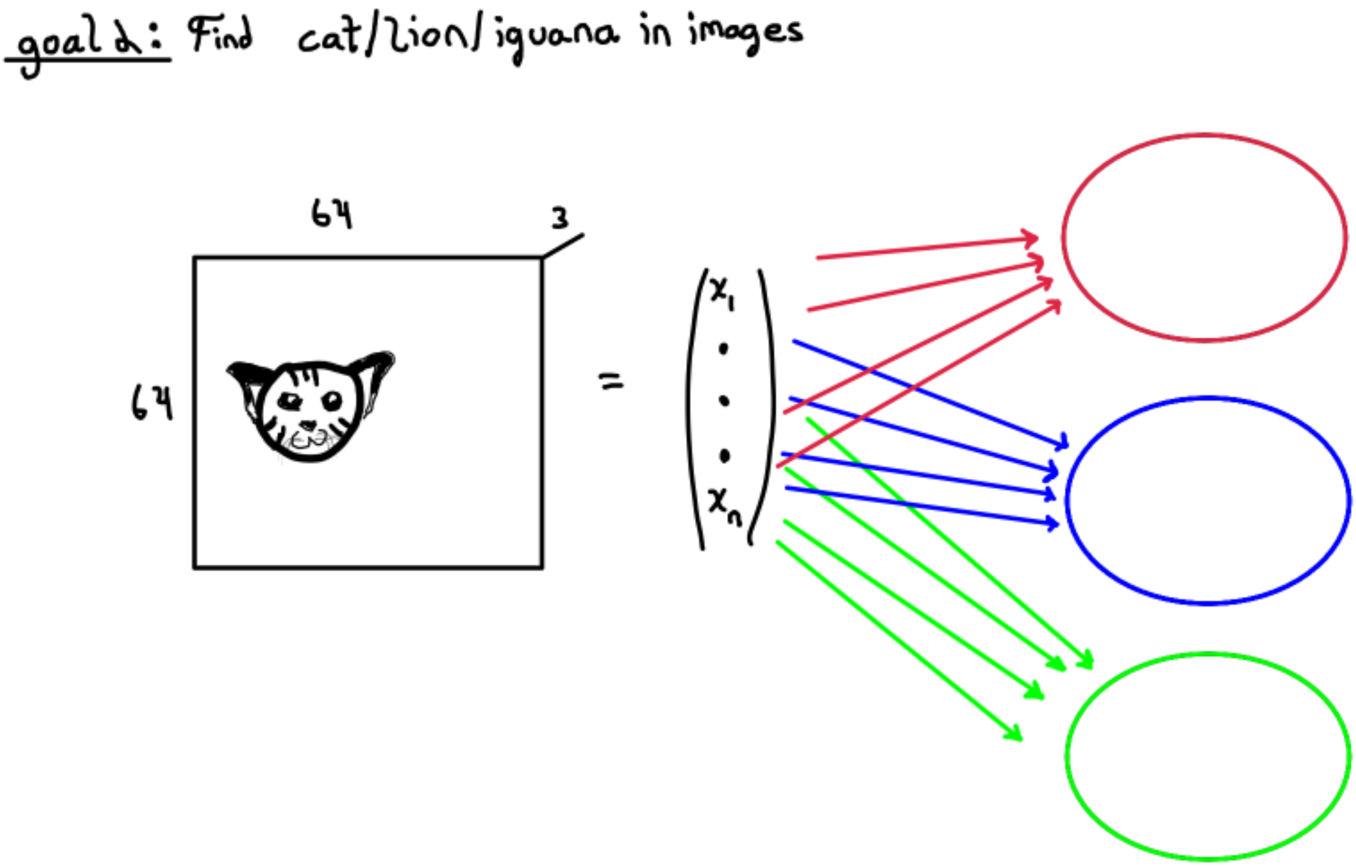
Now we want to detect three types of animals

To modify the network we previously had in order to take this into account, we would add two more ***neurons*** and do the same thing

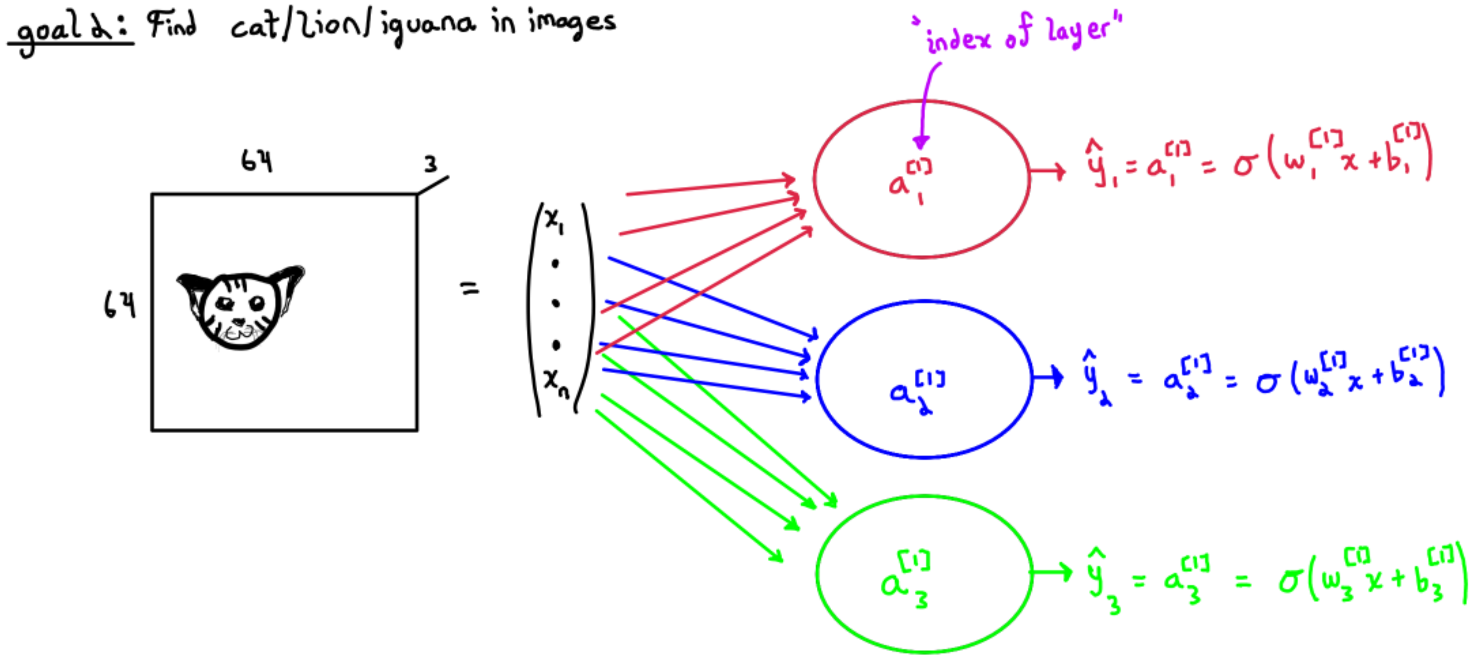
goal 2: Find cat/lion/iguana in images



We take the **64x64x3** color image and flatten it from $\mathbf{x_1}$ to $\mathbf{x_n}$ where **n** represents **64x64x3** and what we will do is we will use **3** ***neurons*** that are all computing the same thing. They're all connected to all these inputs



You connect all your inputs, **x**$_1$ to **x**$_n$, to each of these ***neurons***. We will use a specific set of notation here where the **[]** represents a ***layer***


goal 2: Find cat/lion/iguana in images
"index of layer"
64
3
64
=
x_1
x_n
$a_1^{[1]}$
$\hat{y}_1 = a_1^{[1]} = \sigma(w_1^{[1]} x + b_1^{[1]})$
$a_2^{[1]}$
$\hat{y}_2 = a_2^{[1]} = \sigma(w_2^{[1]} x + b_2^{[1]})$
$a_3^{[1]}$
$\hat{y}_3 = a_3^{[1]} = \sigma(w_3^{[1]} x + b_3^{[1]})$

If you look at this network, it looks like there is one ***layer*** here. There is one ***layer*** in which ***neurons*** don't communicate with each other. We denote with **[]** the index of the ***layer***

The index that is the subscript to this **a**, is the number identifying the ***neuron*** inside the ***layer***. We then have our **y^** that, instead of being a single number as it was before, is now a vector of size **3**

How many parameters does this network have?

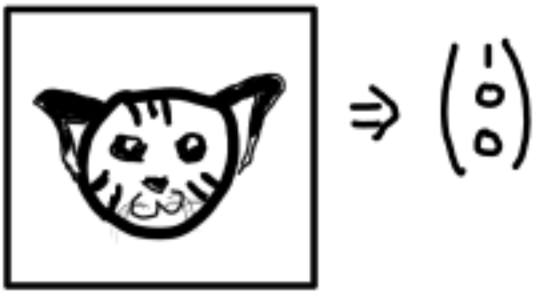
We just have three times the number we had before because we added two more ***neurons*** and they all have their own set of parameters. We have to replicate parameters for each of these, so we have to add two more ***parameter vectors*** and ***biases***

When you had to train the ***logistic regression*** above, what dataset did you need?

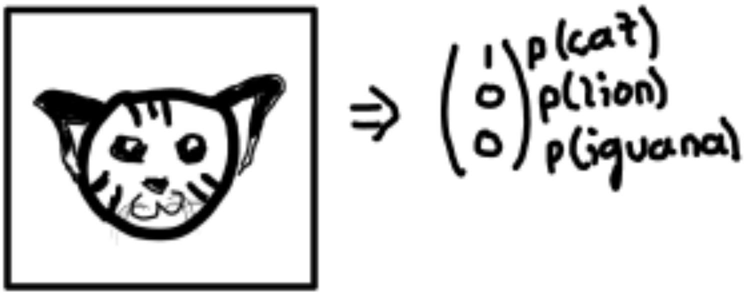
We needed ***images*** and ***labels*** with it labeled as ‘cat’, **1**, or ‘no cat’, **0**. It's a ***binary classification*** with ***images*** and ***labels***

Now, what do you think should be the dataset to train this network?

A label for an image that has a cat would probably be a vector with a **1** and two **0**s:



Where the **1** should represent the presence of a cat, **0** should represent the presence of a lion, and **0** should represent the presence of an iguana:

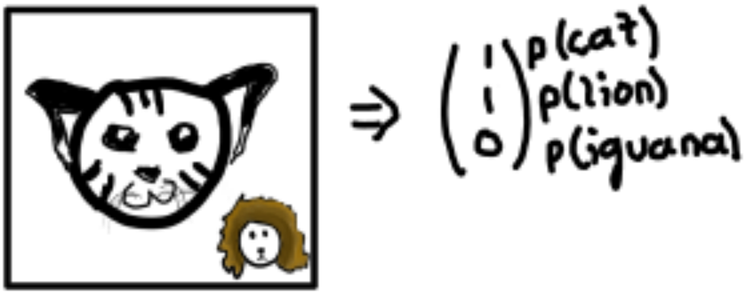


Let's assume we use this scheme to label our dataset. We train this network using the same techniques before; initialize all our ***weights*** and ***biases*** with a value, a starting value, optimize a ***loss function*** by using ***gradient descent***, and then use **y^** to predict

Because we decided to label our dataset like this, after training, the ***red neuron*** is naturally going to be there to detect cats. If we had changed the labeling scheme and we said that the second entry would correspond to the presence of a cat, then after training, you will detect that that ***neuron*** is responsible for detecting a cat. The network is going to evolve depending on the way you label your dataset

Do you think that this network can still be robust to different animals in the same picture?

Let's add a lion here to this picture:



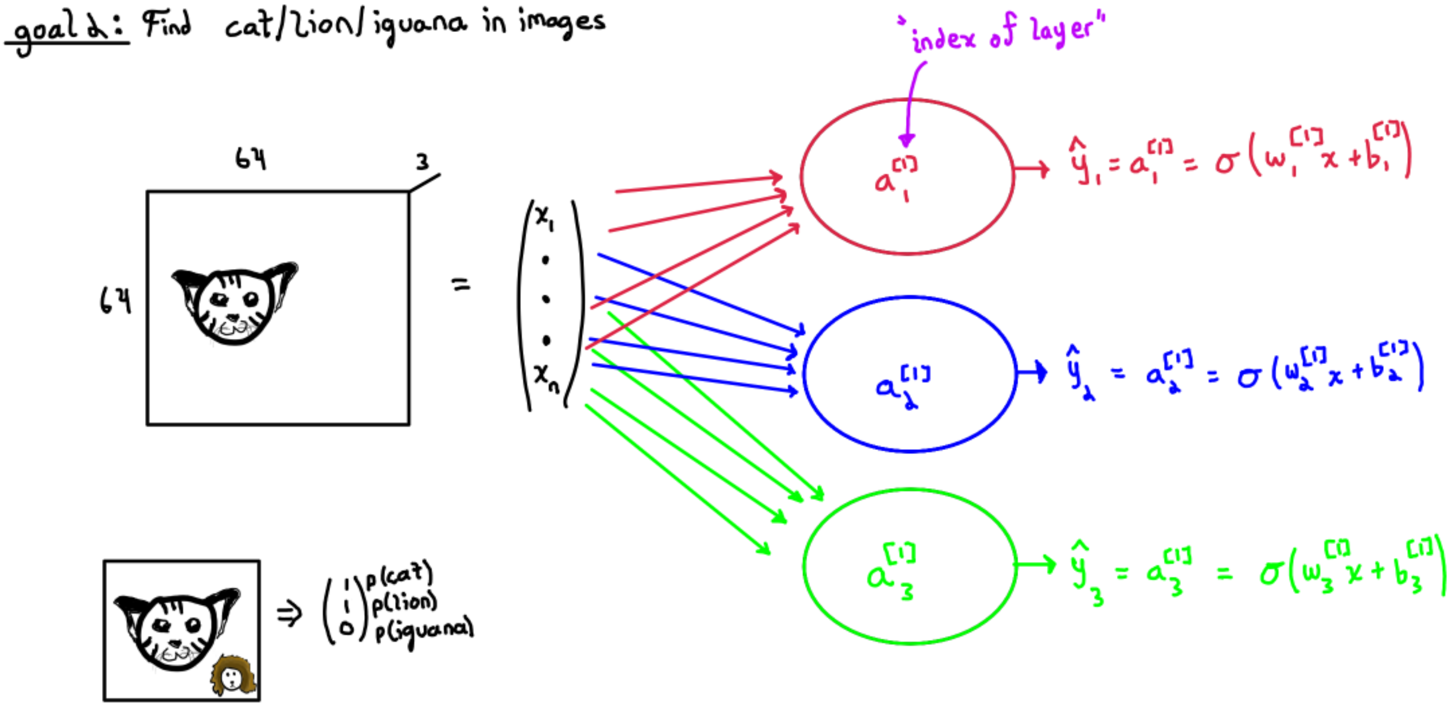
Do you think this network is robust to this type of labeling?

It should be, the neurons aren't talking to each other

The network is seeing **(1,1,0)** and an image. It doesn't see that the first **1** and the lion corresponds to the second **1**. This is a property of ***neural networks***. It's the fact that you don't need to tell them everything. If you have enough data, they're going to figure it out

So because you will also have cats with iguanas, cats alone, lions with iguanas, lions alone, ultimately this neuron (**blue**) will understand what it's looking for, and it will understand that the second **1** corresponds to the lion, it just needs a lot of data

So yes, it's going to be robust to that because the three ***neurons*** aren't communicating together, so we can totally train them independently from each other, and in fact, the ***sigmoids*** don't depend on each other. It means we can have **(1,1,1)** as an output

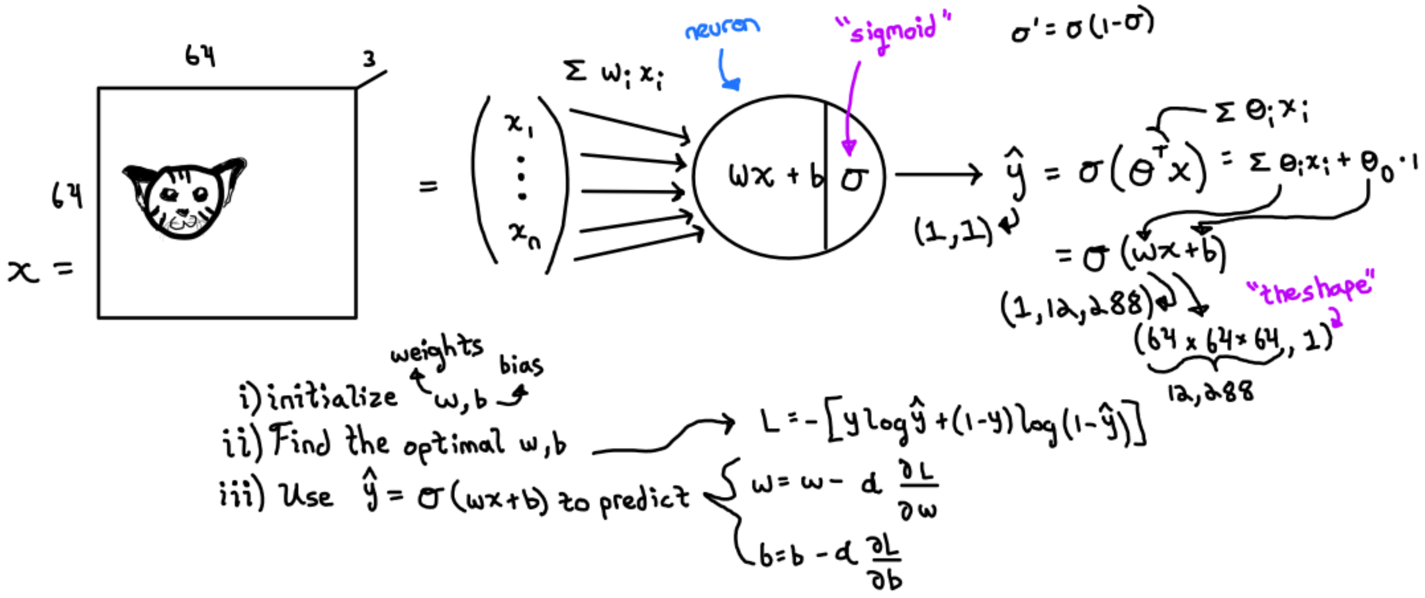

goal 2: Find cat/lion/iguana in images
"index of layer"
64
64
3
=
x_1
x_n
$a_1^{[1]}$
$a_2^{[1]}$
$a_3^{[1]}$
$\hat{y}_1 = a_1^{[1]} = \sigma(w_1^{[1]} x + b_1^{[1]})$
$\hat{y}_2 = a_2^{[1]} = \sigma(w_2^{[1]} x + b_2^{[1]})$
$\hat{y}_3 = a_3^{[1]} = \sigma(w_3^{[1]} x + b_3^{[1]})$
⇒
1 p(cat)
1 p(lion)
0 p(iguana)

You can think about it as **3 *logistic regressions***. We wouldn't call that a ***neural network*** yet, but it's a **3 *neuron neural network*** or **3 *logistic regressions*** with each other

Usually you would have $\mathbf{\Theta^T x}$ which is $\mathbf{\Sigma\Theta_i x_i}$

We will split it into $\mathbf{\Sigma\Theta_i x_i + \Theta_0 * 1}$

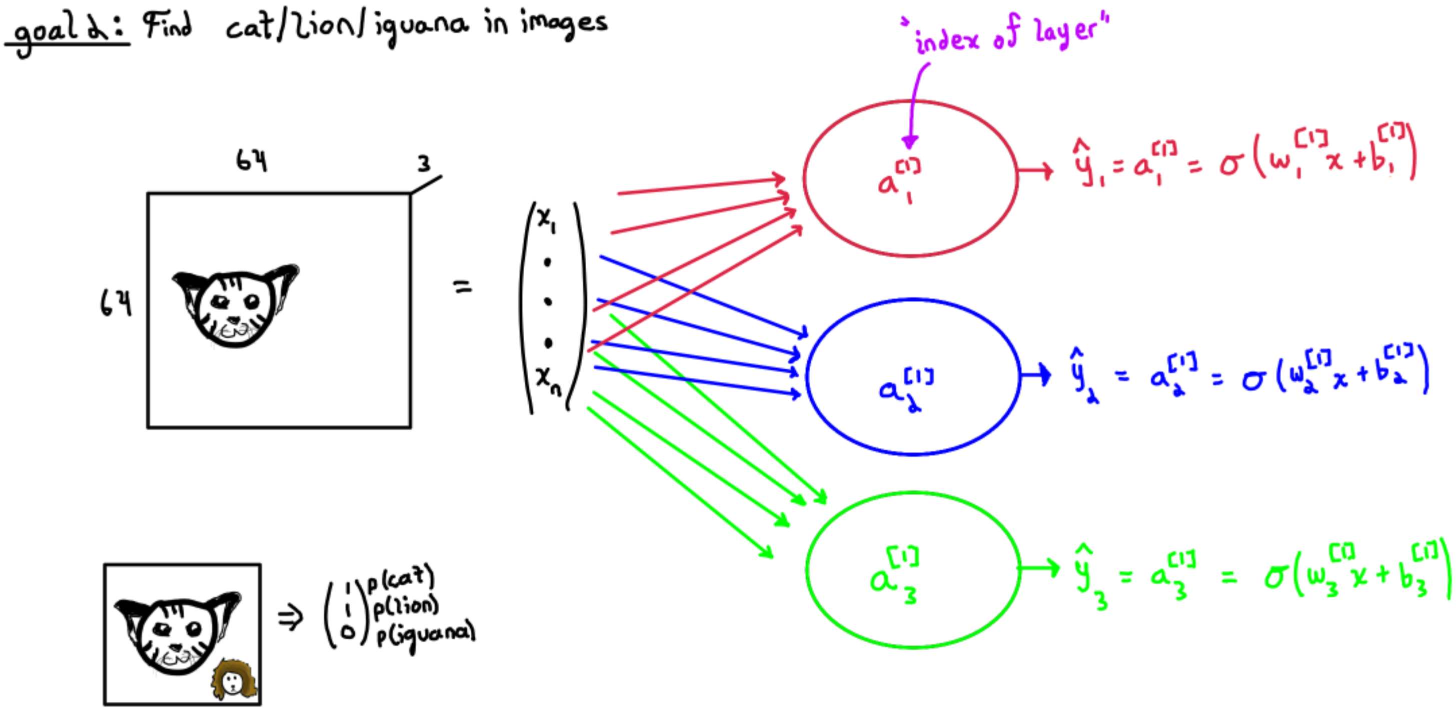
$\mathbf{\Theta_0}$ would correspond to **b** and the $\mathbf{\Theta_i}$ would correspond to $\mathbf{w_i}$


64
3
64
$x =$
=
x_1
x_n
$\Sigma\, w_i x_i$
neuron
"sigmoid"
$\sigma' = \sigma(1-\sigma)$
$wx + b$
σ
$\hat{y} = \sigma(\theta^T x) = \Sigma\, \theta_i x_i + \theta_0 \cdot 1$
$\Sigma\, \theta_i x_i$
$(1,1)$
$= \sigma(wx+b)$
$(1,12,288)$
"the shape"
$(64 \times 64 \times 64, 1)^2$
12,288
i) initialize w, b
weights
bias
ii) Find the optimal w, b
$L = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$
iii) Use $\hat{y} = \sigma(wx+b)$ to predict
$w = w - \alpha \frac{\partial L}{\partial w}$
$b = b - \alpha \frac{\partial L}{\partial b}$

Are there cases where we have a constraint where there is only one possible outcome? Meaning, there is no ‘cat and lion’, there is either a cat or a lion?

Think about healthcare. There are many models that are made to detect if a disease is present based on cell microscopic images. Usually there is no overlap between these. It means you want to classify a specific disease among a large number of diseases

So this model would still work but would not be optimal because it's longer to train:


goal 2: Find cat/lion/iguana in images
64
3
64
=
x_1
x_n
"index of layer"
$a_1^{[1]}$
$\hat{y}_1 = a_1^{[1]} = \sigma(w_1^{[1]}x + b_1^{[1]})$
$a_2^{[1]}$
$\hat{y}_2 = a_2^{[1]} = \sigma(w_2^{[1]}x + b_2^{[1]})$
$a_3^{[1]}$
$\hat{y}_3 = a_3^{[1]} = \sigma(w_3^{[1]}x + b_3^{[1]})$
$\Rightarrow \begin{pmatrix}1\\1\\0\end{pmatrix}$
p(cat)
p(lion)
p(iguana)

Maybe one disease is incredibly rare and one of the ***neurons*** is never going to be trained. So you'll want to start with another model where you put the constraint "*there's only one disease that we want to predict*" and let the model learn, with all the ***neurons*** learning together by creating interaction between them

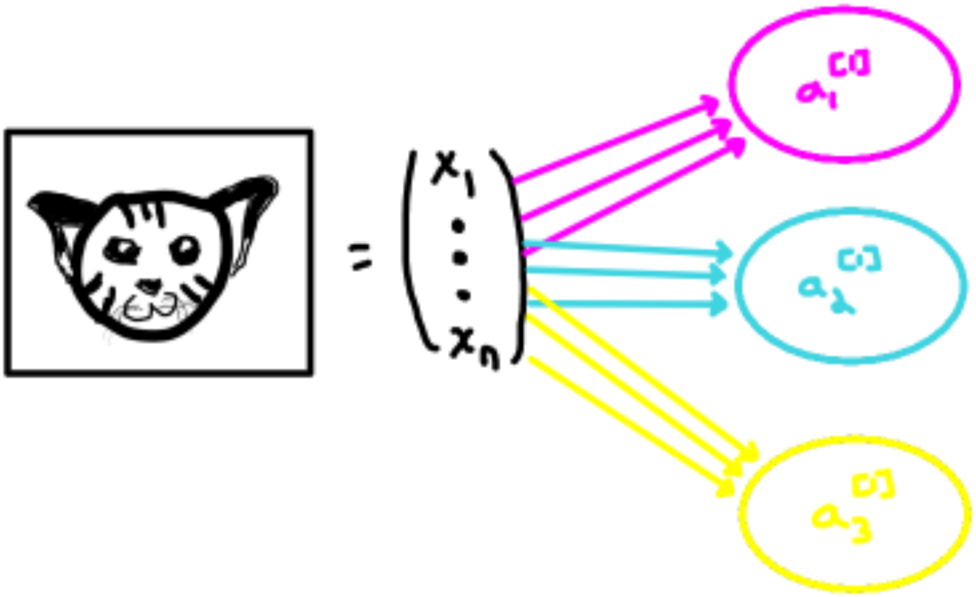
Softmax:

We'll set a new goal now. We will add a constraint which is an unique animal on an image. So at most, one animal on an image. We're going to modify our network a little bit

We have our cat, we flatten it, and now we're going to use the same scheme with the 3 ***neurons***

goal 3: + constraint

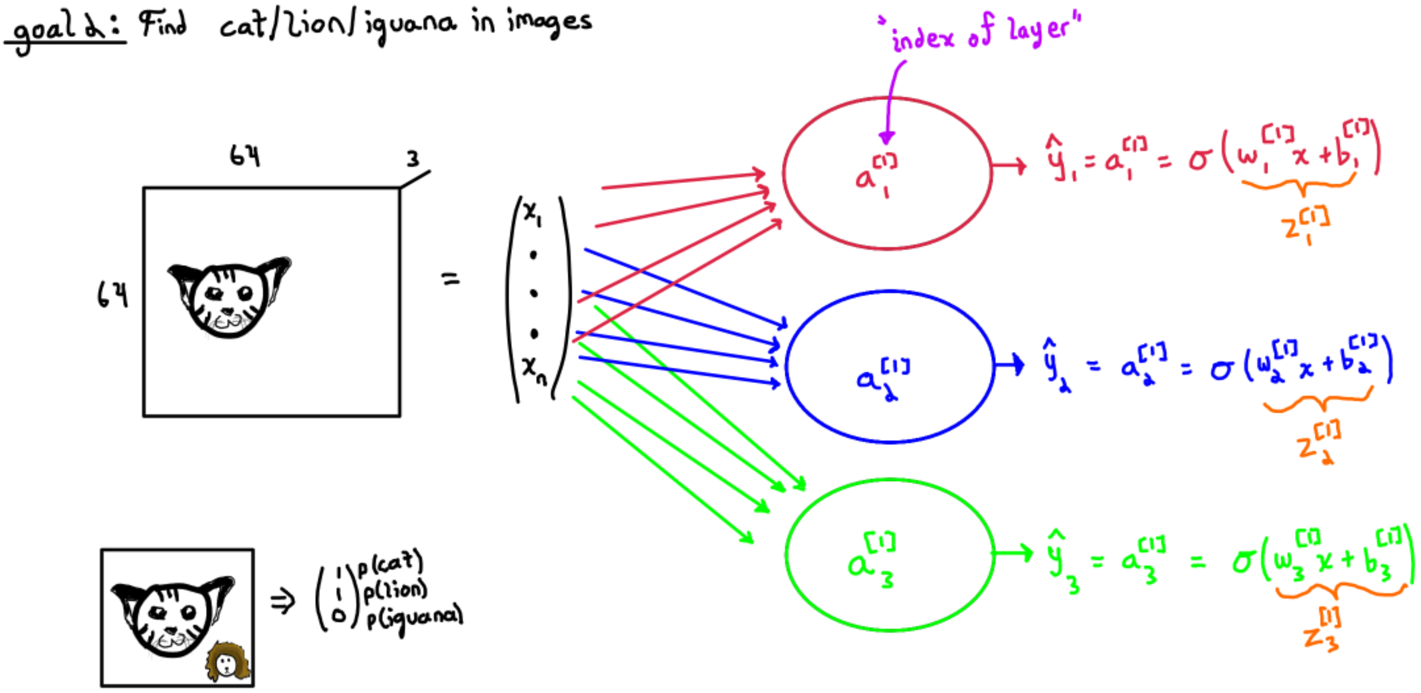
unique animal on an image



But as an output, what we're going to use is a ***softmax function***. We're going to introduce another notation to make it easier

As you know, the ***neuron*** is a ***linear*** part plus an ***activation***. So we're going to introduce a notation for the ***linear*** part

goal 2: Find cat/lion/iguana in images



$$\hat{y}_1 = a_1^{[1]} = \sigma(\underbrace{w_1^{[1]} x + b_1^{[1]}}_{z_1^{[1]}})$$

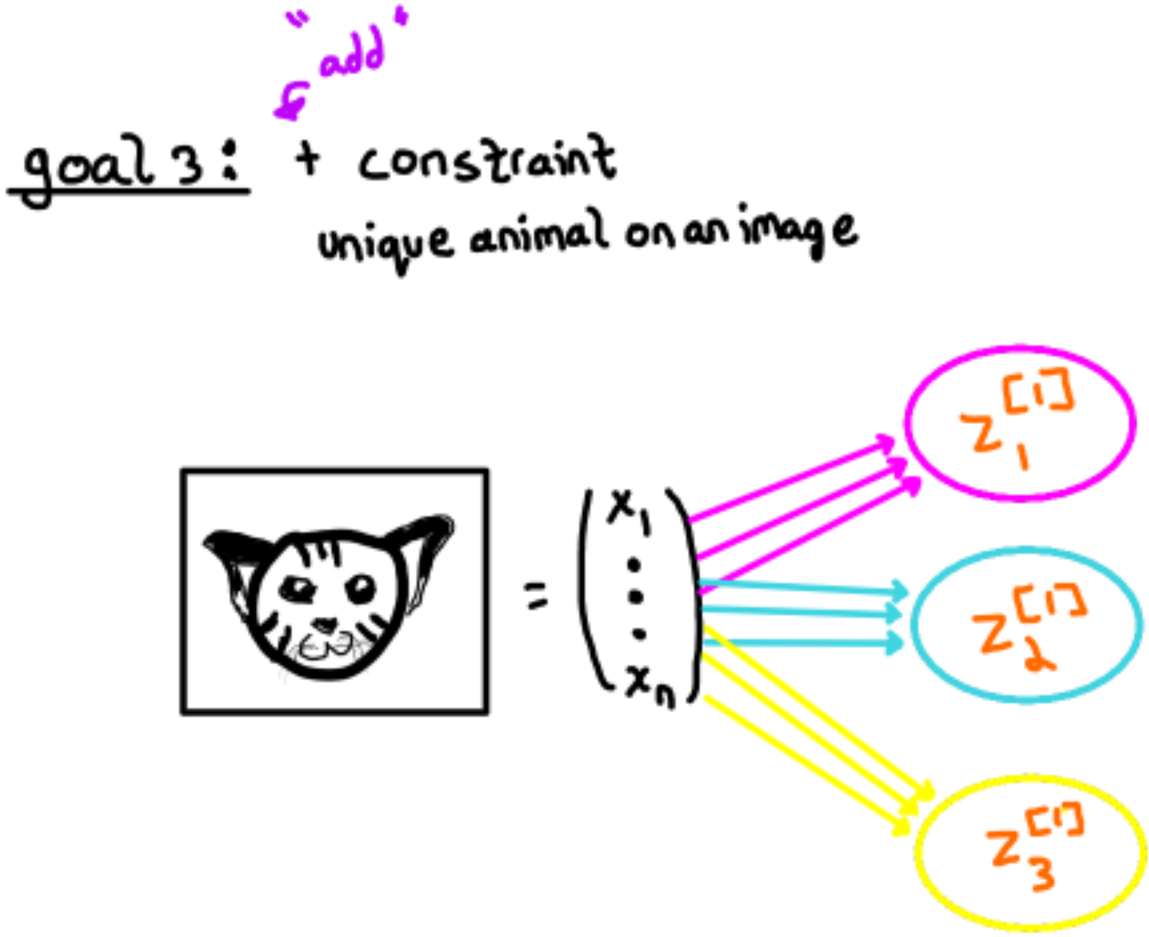
$$\hat{y}_2 = a_2^{[1]} = \sigma(\underbrace{w_2^{[1]} x + b_2^{[1]}}_{z_2^{[1]}})$$

$$\hat{y}_3 = a_3^{[1]} = \sigma(\underbrace{w_3^{[1]} x + b_3^{[1]}}_{z_3^{[1]}})$$

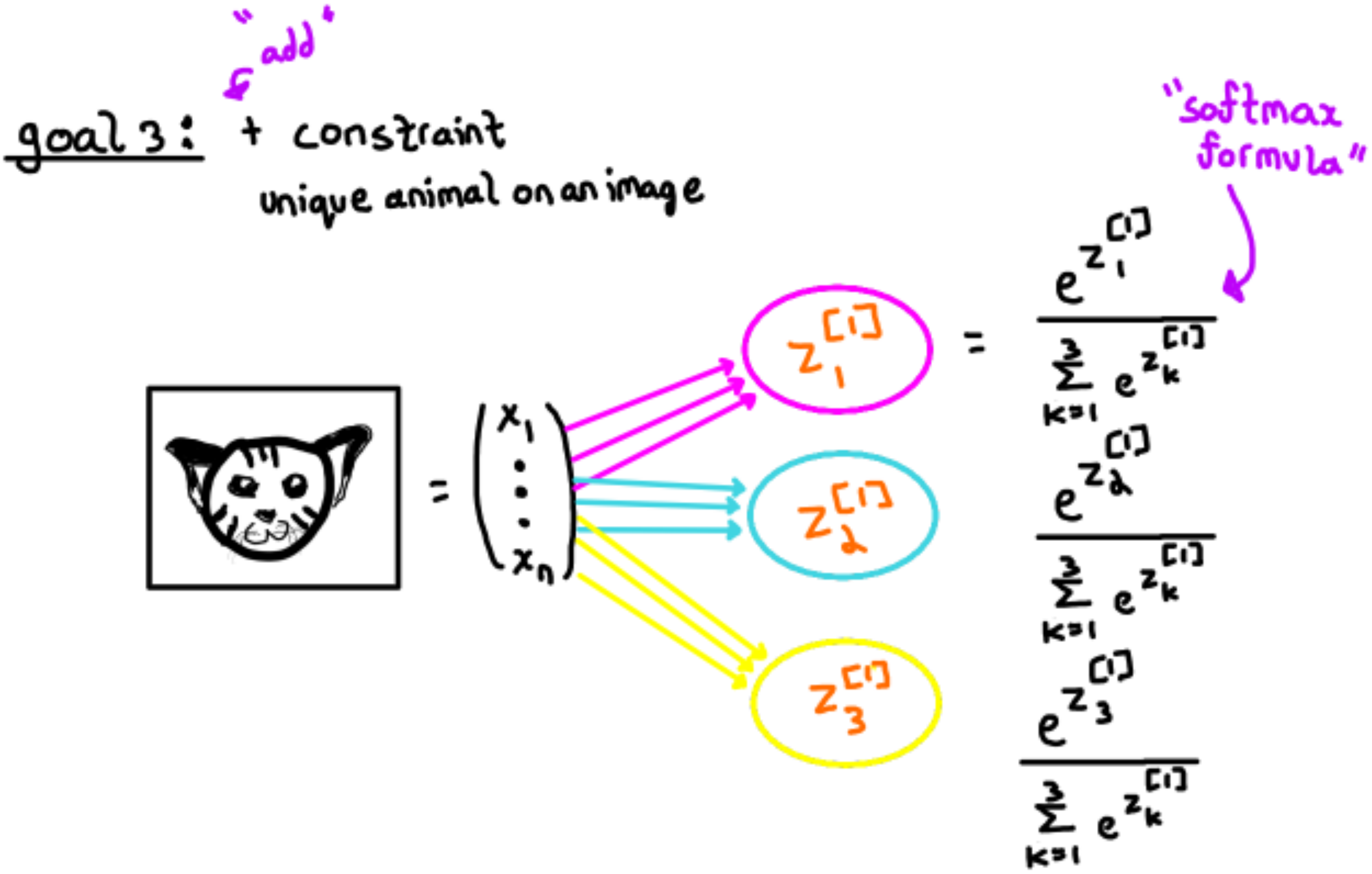
$$\Rightarrow \begin{pmatrix}1\\1\\0\end{pmatrix} \begin{matrix}p(cat)\\p(lion)\\p(iguana)\end{matrix}$$

So now our ***neuron*** has two parts, one which computes **Z**, and one which computes **a**, equals **σ(Z)**

Now we're going to remove all the ***activations*** and make them **Z**s



And we're going to use the specific formula (***softmax formula***):


"add"
goal 3: + constraint
unique animal on an image
"softmax formula"
=
x_1
x_n
$z_1^{[1]}$
$z_2^{[1]}$
$z_3^{[1]}$
$= \frac{e^{z_1^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$
$\frac{e^{z_2^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$
$\frac{e^{z_3^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$

Why is this formula interesting and why is it not robust to this labeling scheme anymore?

It's because the sum of the outputs of this network have to sum up to **1**. You can try it, if you sum the three outputs, you get the same thing in the numerator and on the denominator and you get **1**. So instead of getting a probabilistic output for each **y^**, we will get a probability distribution over all the classes. So that means we can't get **(0.7, 0.6, 0.1)** telling us roughly that there is probably a cat and a lion and no iguana. We have to sum these to **1**. So it means, if there is no cat and no lion, it means there's very likely an iguana. The three probabilities are ***dependent*** on each other

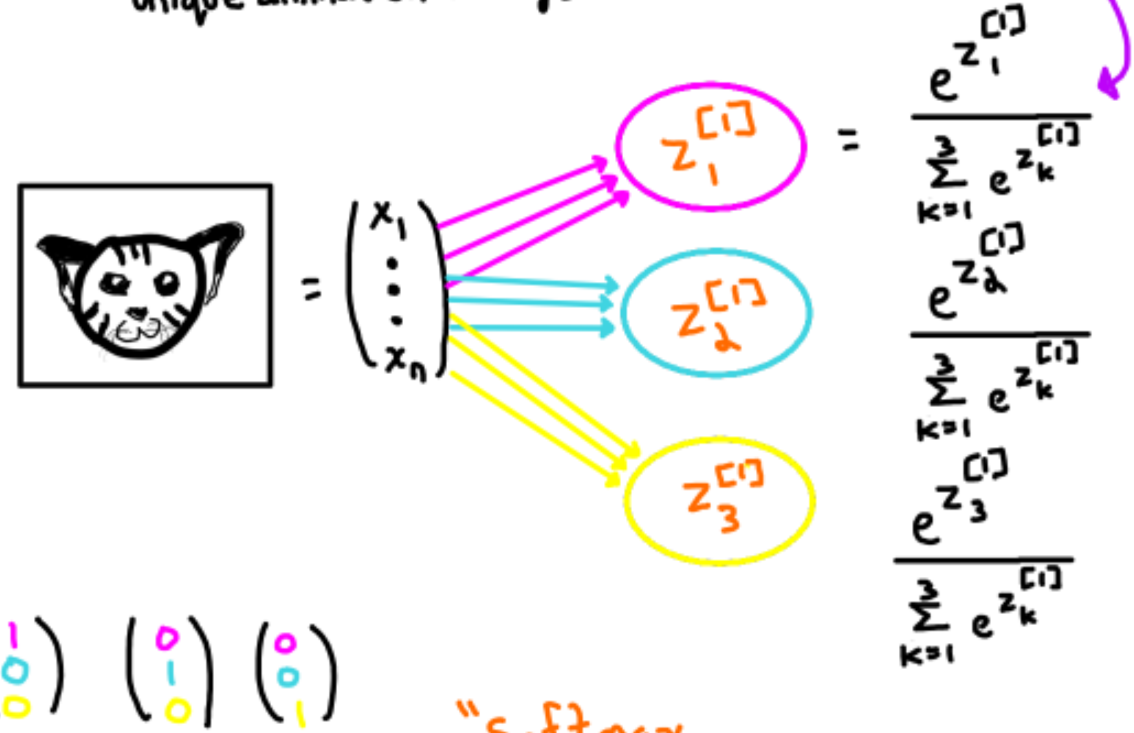
And for this one, we have to label the following to represent a cat, a lion, and an iguana. This is called a ***softmax multi-class network***:

"add"

goal 3: + constraint

unique animal on an image

"softmax formula"



$$\frac{e^{z_1^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\frac{e^{z_2^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\frac{e^{z_3^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\begin{pmatrix}1\\0\\0\end{pmatrix} \quad \begin{pmatrix}0\\1\\0\end{pmatrix} \quad \begin{pmatrix}0\\0\\1\end{pmatrix}$$

"softmax multi-class regression"

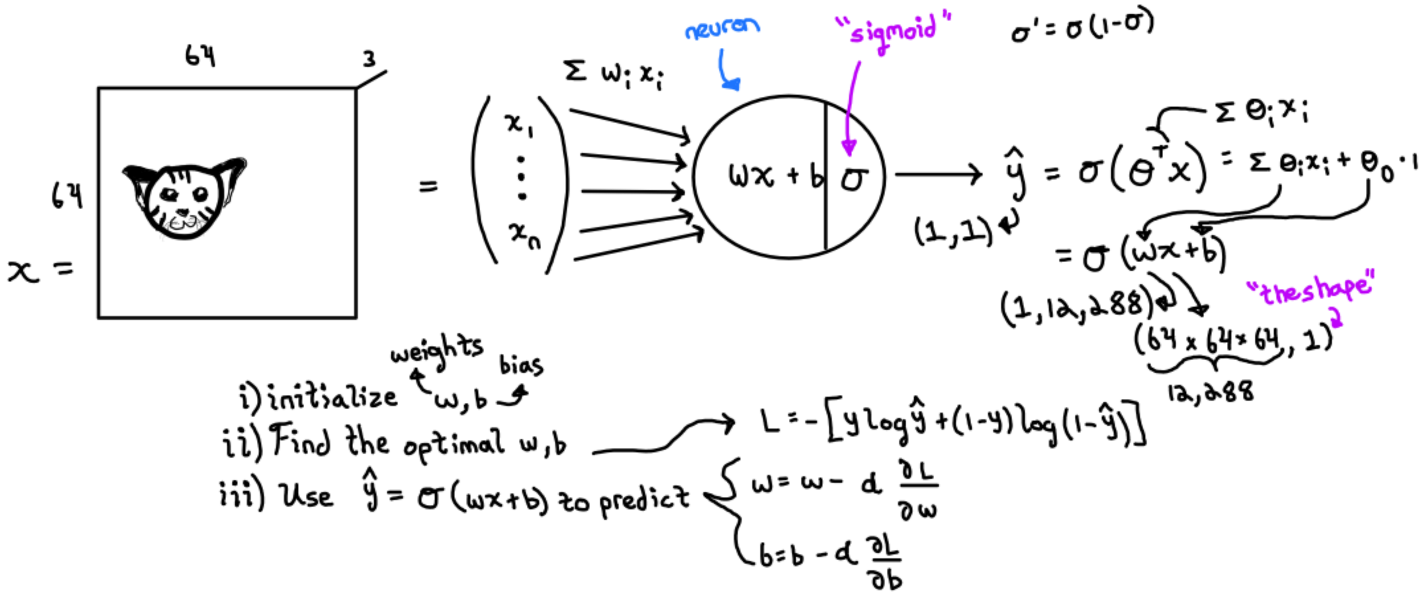
You assume there is at least one of the three classes, otherwise you have to add a fourth input that will represent an abscence of an animal

How many parameters does the ***network*** have?

It still has the same amount of parameters as our second ***network***. We still have **3** ***neurons*** so there **3n + 3** parameters

How do we train these parameters (the **3n + 3** parameters) ?

The problem with our previous ***scheme***'s ***loss function*** is that there's only two outcomes



So in this ***loss function***, **y** is either a **0** and **1**, **y^** is a number between **0** and **1**, so it cannot match this labeling, so we need to modify the ***loss function***

We'll call it ***loss 3 neuron***. We're going to just sum it up for the **3 *neurons***:

$$\mathcal{L}_{3N} = -\sum_{k=1}^{3}\left[y_k \log \hat{y}_k + (1-y_k)\log(1-\hat{y}_k)\right]$$

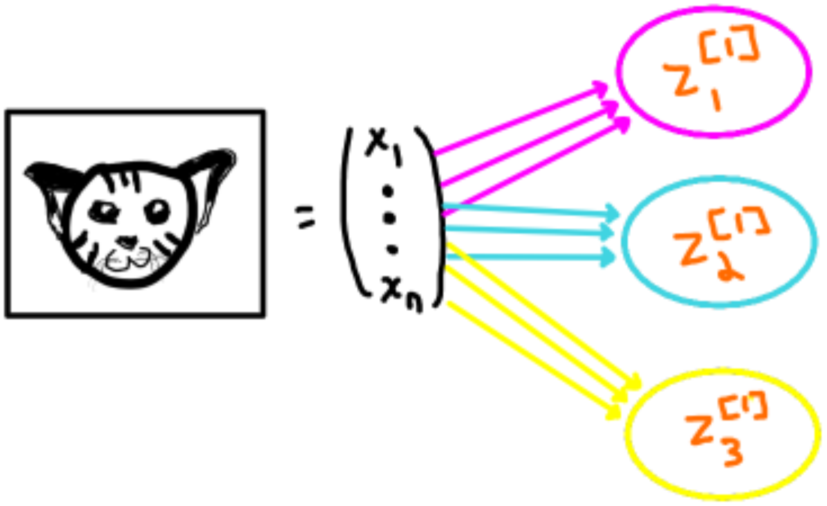
"add"

"Neuron"

goal 3: + constraint

unique animal on an image

"softmax formula"



$$z_1^{[1]} = \frac{e^{z_1^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad \frac{e^{z_2^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad \frac{e^{z_3^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\begin{pmatrix}1\\0\\0\end{pmatrix} \begin{pmatrix}0\\1\\0\end{pmatrix} \begin{pmatrix}0\\0\\1\end{pmatrix}$$

"softmax multi-class regression"

So we're just doing three times the ***loss*** for each of the ***neurons***. So we have exactly three times the ***loss function***. We sum them together. And if you train this ***loss function***, you should be able to train the **3** ***neurons*** that you have

Again, talking about scarcity of one of the classes, if there's not many iguanas, then the third term of this sum is not going to help the third ***neuron*** train towards detecting an iguana. It's going to push it to detect no iguana

The output of this network once it's trained is going to be a probability distribution. You will pick the maximum of those and you will set it to **1** and the others to **0**, as your prediction

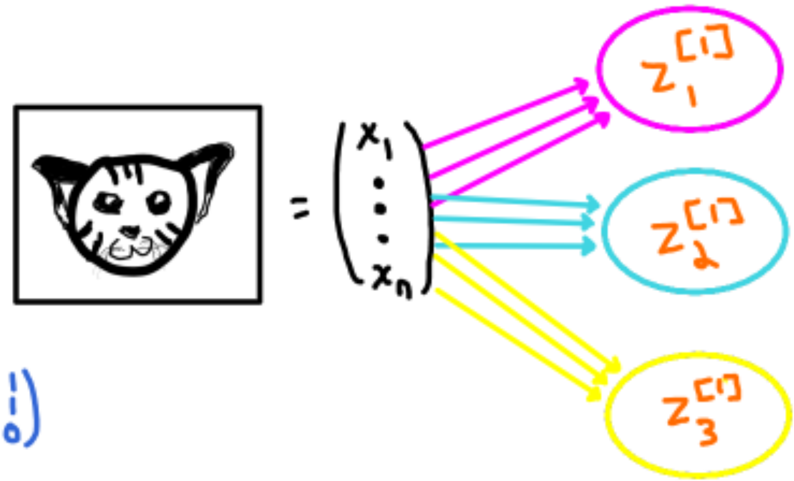
If you use this labeling scheme, like **(1,1,0)** for this ***network***:

$$\mathcal{L}_{3N} = -\sum_{k=1}^{3}\left[y_k \log \hat{y}_k + (1-y_k)\log(1-\hat{y}_k)\right]$$

"add" "Neuron"

goal 3: + constraint

unique animal on an image



"softmax formula"

$$= \frac{e^{z_1^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\frac{e^{z_2^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\frac{e^{z_3^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\begin{pmatrix}1\\1\\0\end{pmatrix}$$

$$\begin{pmatrix}1\\0\\0\end{pmatrix} \quad \begin{pmatrix}0\\1\\0\end{pmatrix} \quad \begin{pmatrix}0\\0\\1\end{pmatrix}$$

"softmax multi-class regression"

It will probably not work and the reason is because the sum of these entries will be equal to **2**, while the sum of the ***softmax formula*** entries is equal to **1**. So you will never be able to match the ***output*** to the ***input*** to the ***label***. So what the network is probably going to do is it's probably going to send these to:

$$\mathcal{L}_{3N} = -\sum_{k=1}^{3}\left[y_k \log \hat{y}_k + (1-y_k)\log(1-\hat{y}_k)\right]$$

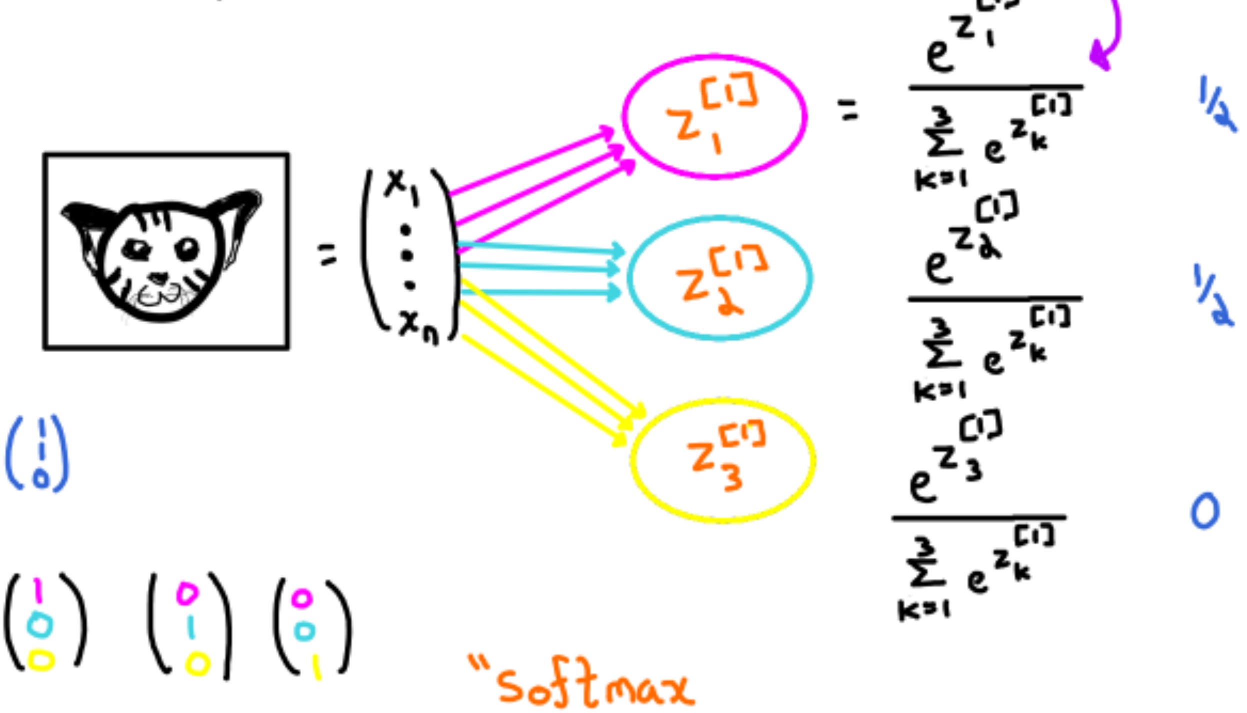
"Neuron"

"add"

goal 3: + constraint

unique animal on an image

"softmax formula"



$$z_1^{[1]} = \frac{e^{z_1^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad \frac{e^{z_2^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad \frac{e^{z_3^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\begin{pmatrix}1\\1\\0\end{pmatrix}$$

$$\begin{pmatrix}1\\0\\0\end{pmatrix} \quad \begin{pmatrix}0\\1\\0\end{pmatrix} \quad \begin{pmatrix}0\\0\\1\end{pmatrix}$$

"Softmax multi-class regression"

Which is not what you want

Loss function for softmax regression:

What's interesting about this loss:

$$\mathcal{L}_{3N} = -\sum_{k=1}^{3}\left[y_k \log \hat{y}_k + (1-y_k)\log(1-\hat{y}_k)\right]$$

"Neuron"

is if you take this derivative:

$$\frac{\partial \mathcal{L}_{3N}}{\partial w_2^{[1]}}$$

It will not be harder than this derivative

$$L = -\left[y \log \hat{y} + (1-y)\log(1-\hat{y})\right]$$

$$w = w - \alpha \frac{\partial L}{\partial w}$$

$$b = b - \alpha \frac{\partial L}{\partial b}$$

It's going to be exactly the same because only one of these three terms depends on **w$^{[1]}$$_2$** :

$$\mathcal{L}_{\underset{\overset{\uparrow}{\text{"Neuron"}}}{3N}} = -\sum_{k=1}^{3}\left[y_k \log \hat{y}_k + (1-y_k)\log(1-\hat{y}_k)\right]$$

It means the derivative of the two others are **0**. So we are exactly at the same complexity during the derivation

$$\mathcal{L}_{3N} = -\sum_{k=1}^{3}\left[y_k \log \hat{y}_k + (1-y_k)\log(1-\hat{y}_k)\right]$$

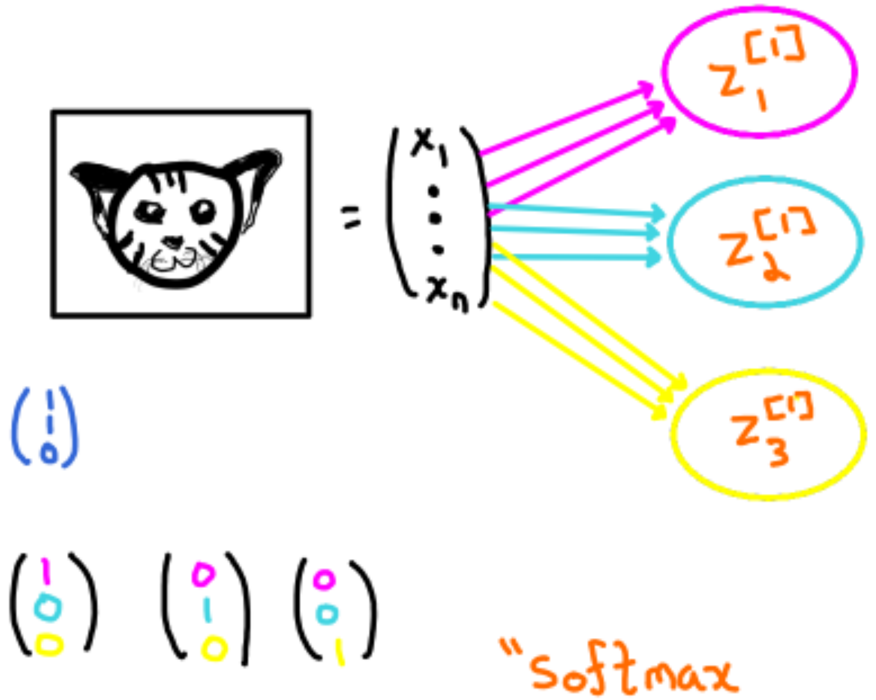
"add" "Neuron"

goal 3: + constraint

unique animal on an image

$$\frac{\partial \mathcal{L}_{3N}}{\partial w_2^{[1]}}$$

"softmax formula"



$$= \frac{e^{z_1^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad 1/2$$

$$\frac{e^{z_2^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad 1/2$$

$$\frac{e^{z_3^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad 0$$

"softmax multi-class regression"

Let's say we define a ***loss function*** that corresponds roughly to this:

$$\mathcal{L}_{3N} = -\sum_{k=1}^{3}\left[y_k \log \hat{y}_k + (1-y_k)\log(1-\hat{y}_k)\right]$$

"Neuron"

If you try to compute the derivative of the loss with respect to $\mathbf{w}_2$, it will become much more complex because this number:

soft

$$\frac{e^{z_1^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\frac{e^{z_2^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

$$\frac{e^{z_3^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

the output here that is going to impact the ***loss function*** directly, not only depends on the parameters of $\mathbf{w}_2$, it also depends on the parameters of $\mathbf{w}_1$ and $\mathbf{w}_3$. And the same for this output, this output also depends on the parameters $\mathbf{w}_2$, because of this denominator:

$$\frac{e}{\sum_{k=1}^{3} e^{z_k^{[1]}}}$$

So the ***softmax regression*** needs a different ***loss function*** and a different derivative. So the ***loss function*** we'll define is a very common one in ***deep learning***, it's called the ***softmax cross entropy loss***

Cross-Entropy Loss

$$\mathcal{L}_{CE} = -\sum_{k=1}^{3} y_k \log \hat{y}_k$$

It surprisingly looks like the ***logistic loss function***. The only difference is that we will sum it up on all the classes

Cross-Entropy Loss

$$\mathcal{L}_{CE} = -\sum_{k=1}^{3} y_k \log \hat{y}_k$$

$$\mathcal{L}_{3N} = -\sum_{k=1}^{3}\left[y_k \log \hat{y}_k + (1-y_k)\log(1-\hat{y}_k)\right]$$

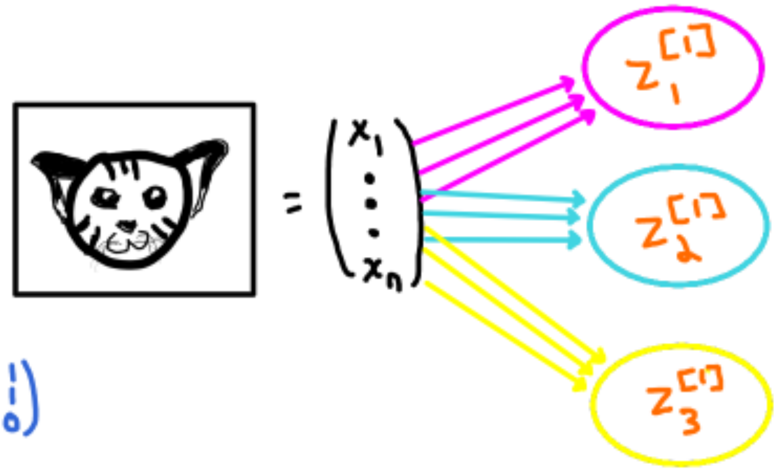
"add" "Neuron"

goal 3: + constraint

unique animal on an image

$$\frac{\partial \mathcal{L}_{3N}}{\partial w_2^{[1]}}$$

"softmax formula"



$$= \frac{e^{z_1^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad \frac{1}{2}$$

$$\frac{e^{z_2^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad \frac{1}{2}$$

$$\frac{e^{z_3^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \qquad 0$$

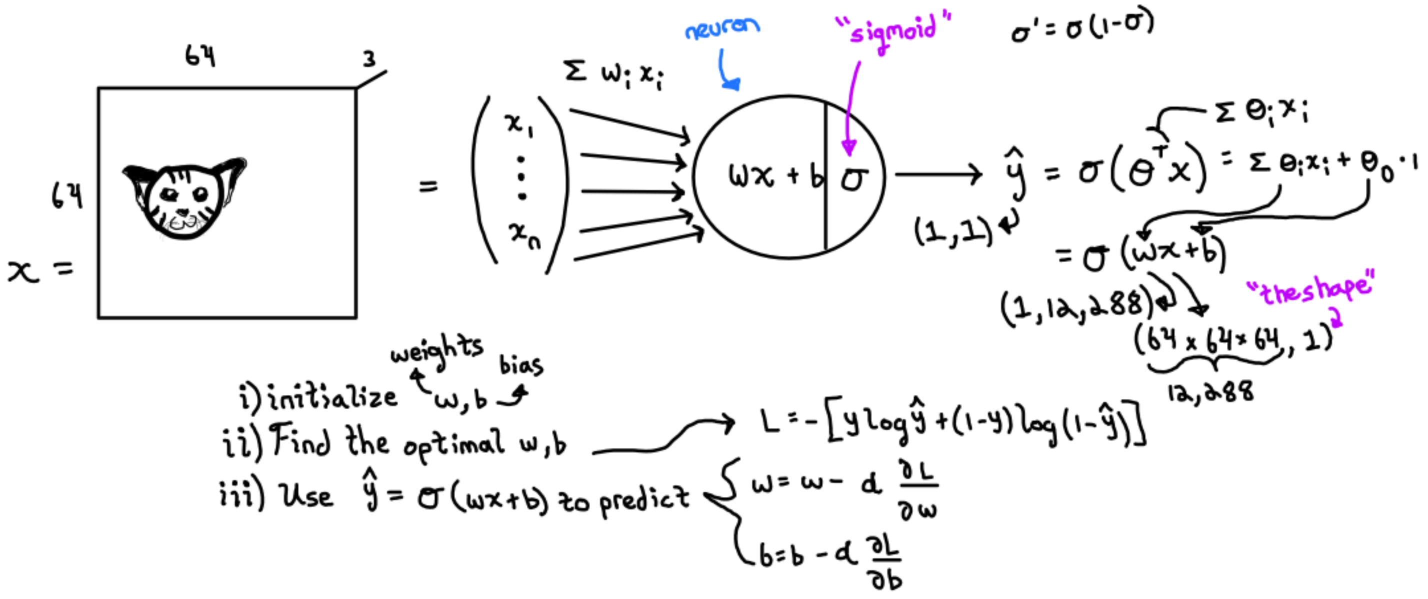
$$\begin{pmatrix}1\\1\\0\end{pmatrix}$$

$$\begin{pmatrix}1\\0\\0\end{pmatrix} \quad \begin{pmatrix}0\\1\\0\end{pmatrix} \quad \begin{pmatrix}0\\0\\1\end{pmatrix}$$

"Softmax multi-class regression"

This ***binary cross entropy loss*** is very likely to be used in ***classification problems*** that are ***multi-class***

##Let's say, instead of trying to predict if there is a cat or no cat, we were trying to predict the age of the cat based on the image, what would you change in this network? Instead of predicting **1**,**0** , you want to predict the age of the cat


64
3
64
x =
=
x_1
x_n
$\Sigma\, w_i x_i$
neuron
"sigmoid"
$\sigma' = \sigma(1-\sigma)$
$wx + b$
σ
$\hat{y} = \sigma(\theta^T x) = \Sigma\, \theta_i x_i + \theta_0 \cdot 1$
$\Sigma\, \theta_i x_i$
(1,1)
$= \sigma(wx+b)$
(1,12,288)
"the shape"
$(64 \times 64 \times 64, 1)$
12,288
i) initialize w,b
weights
bias
ii) Find the optimal w,b
$L = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$
iii) Use $\hat{y} = \sigma(wx+b)$ to predict
$w = w - \alpha \frac{\partial L}{\partial w}$
$b = b - \alpha \frac{\partial L}{\partial b}$

You basically make several output nodes where each of them corresponds to one age of cats, so would you use the above network or the third one?

Cross-Entropy Loss

$$\mathcal{L}_{CE} = -\sum_{k=1}^{3} y_k \log \hat{y}_k$$

$$\mathcal{L}_{3N} = -\sum_{k=1}^{3} \left[y_k \log \hat{y}_k + (1-y_k) \log (1-\hat{y}_k) \right]$$

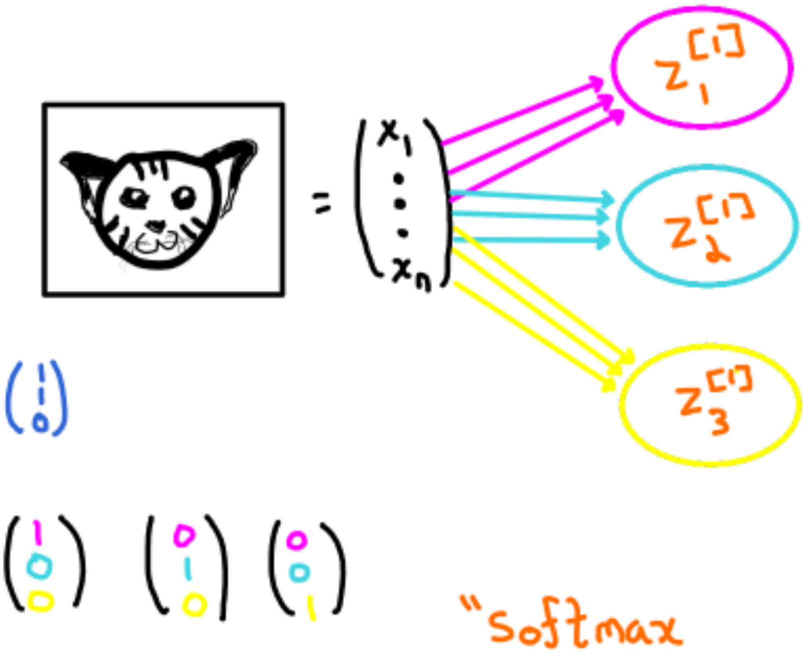
"add" "Neuron"

goal 3: + constraint

unique animal on an image

$$\frac{\partial \mathcal{L}_{3N}}{\partial w_2^{[1]}}$$

"softmax formula"


x_1
x_n
$z_1^{[1]}$
$z_2^{[1]}$
$z_3^{[1]}$
=
$\begin{pmatrix}1\\1\\0\end{pmatrix}$
$\begin{pmatrix}1\\0\\0\end{pmatrix}$
$\begin{pmatrix}0\\1\\0\end{pmatrix}$
$\begin{pmatrix}0\\0\\1\end{pmatrix}$

$$= \frac{e^{z_1^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \quad 1/2$$

$$\frac{e^{z_2^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \quad 1/2$$

$$\frac{e^{z_3^{[1]}}}{\sum_{k=1}^{3} e^{z_k^{[1]}}} \quad 0$$

"softmax multi-class regression"

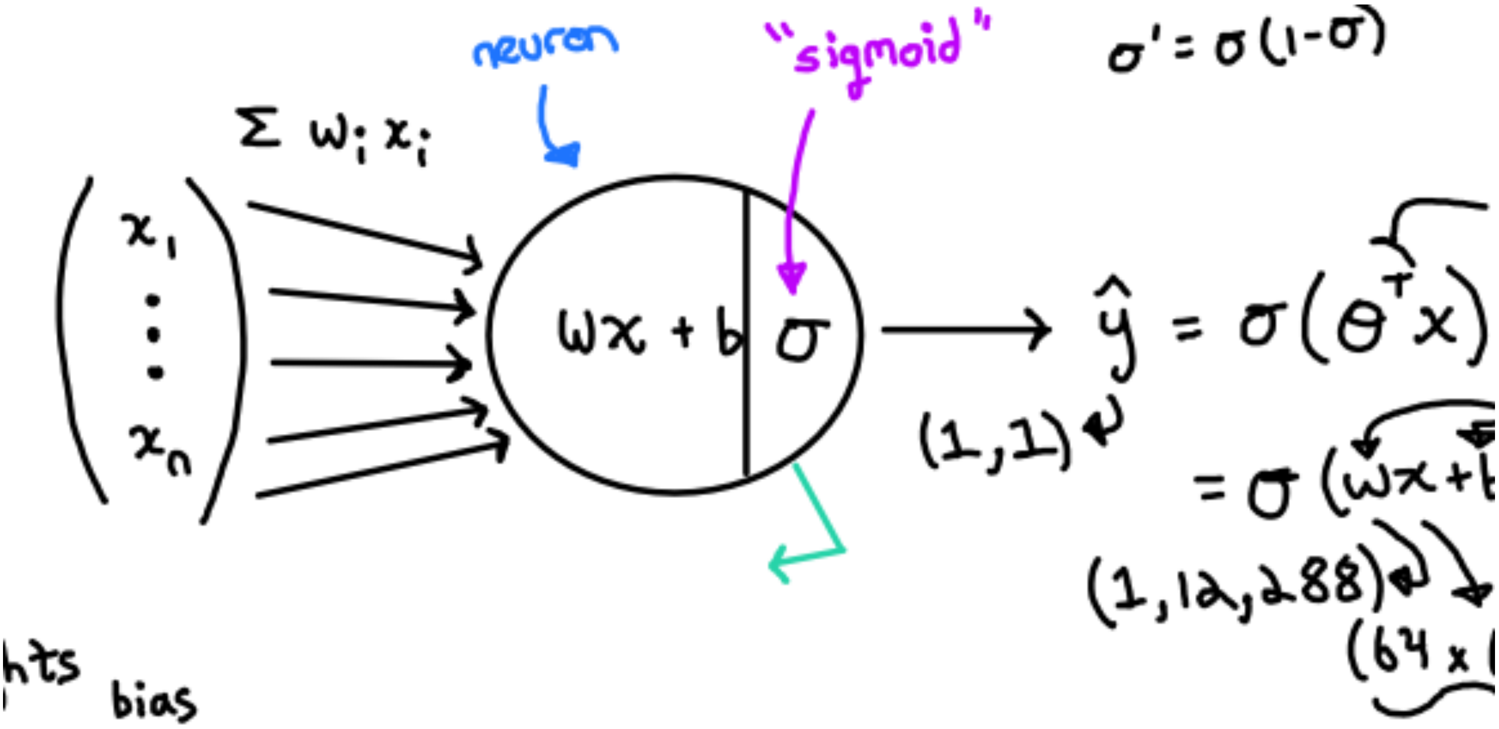
Would you use the **3 *neuron neural network*** or the ***softmax regression***?

You would use the third one because you have a unique age, you cannot have two ages. So we would use the ***softmax regression*** one because we want a probability distribution along the age. There is also another approach which is using directly, a ***regression*** to predict an age. An age can be between **0** and a certain number

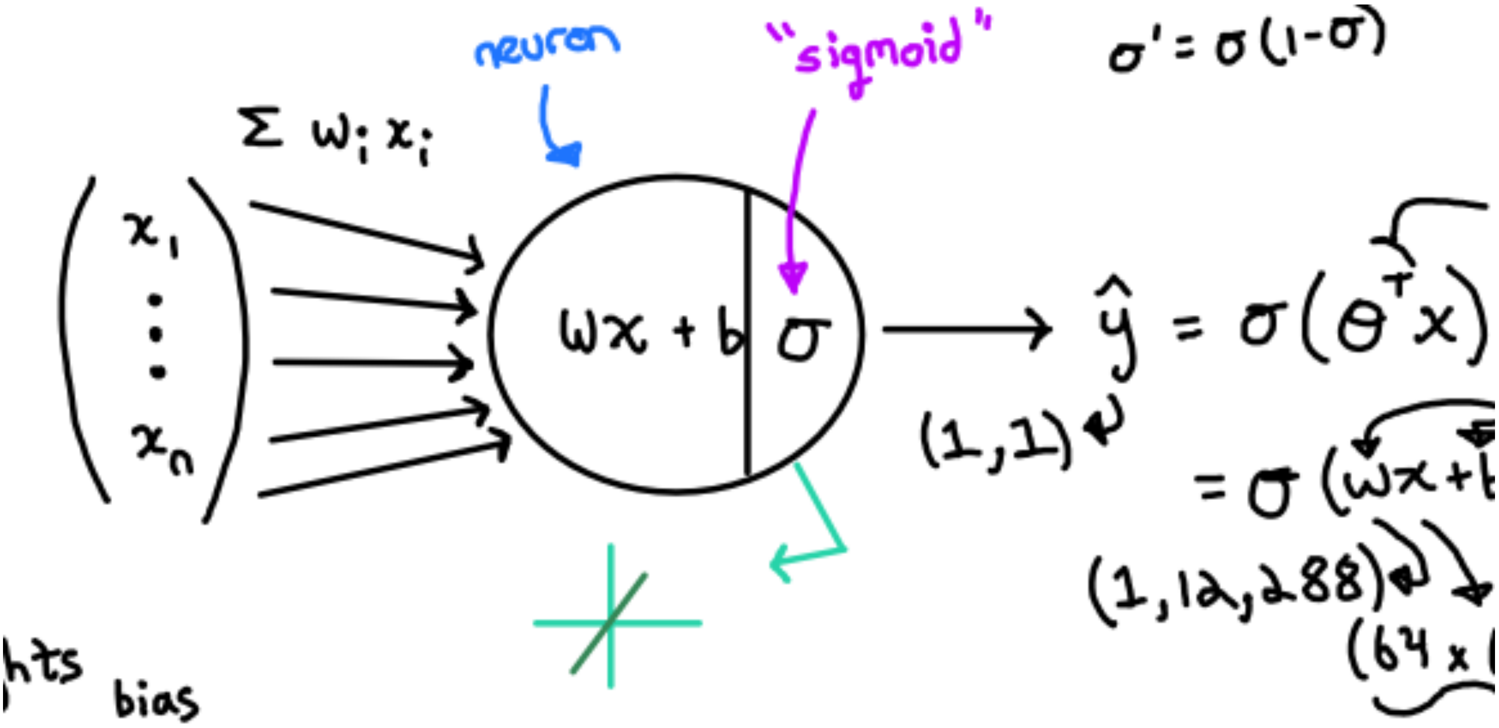
Let's say you want to do a ***regression***, how would you modify your network?

By changing the ***sigmoid***. The ***sigmoid*** puts the **Z** between **0** and **1**, we don't want this to happen

Into what function would you change the ***sigmoid***?

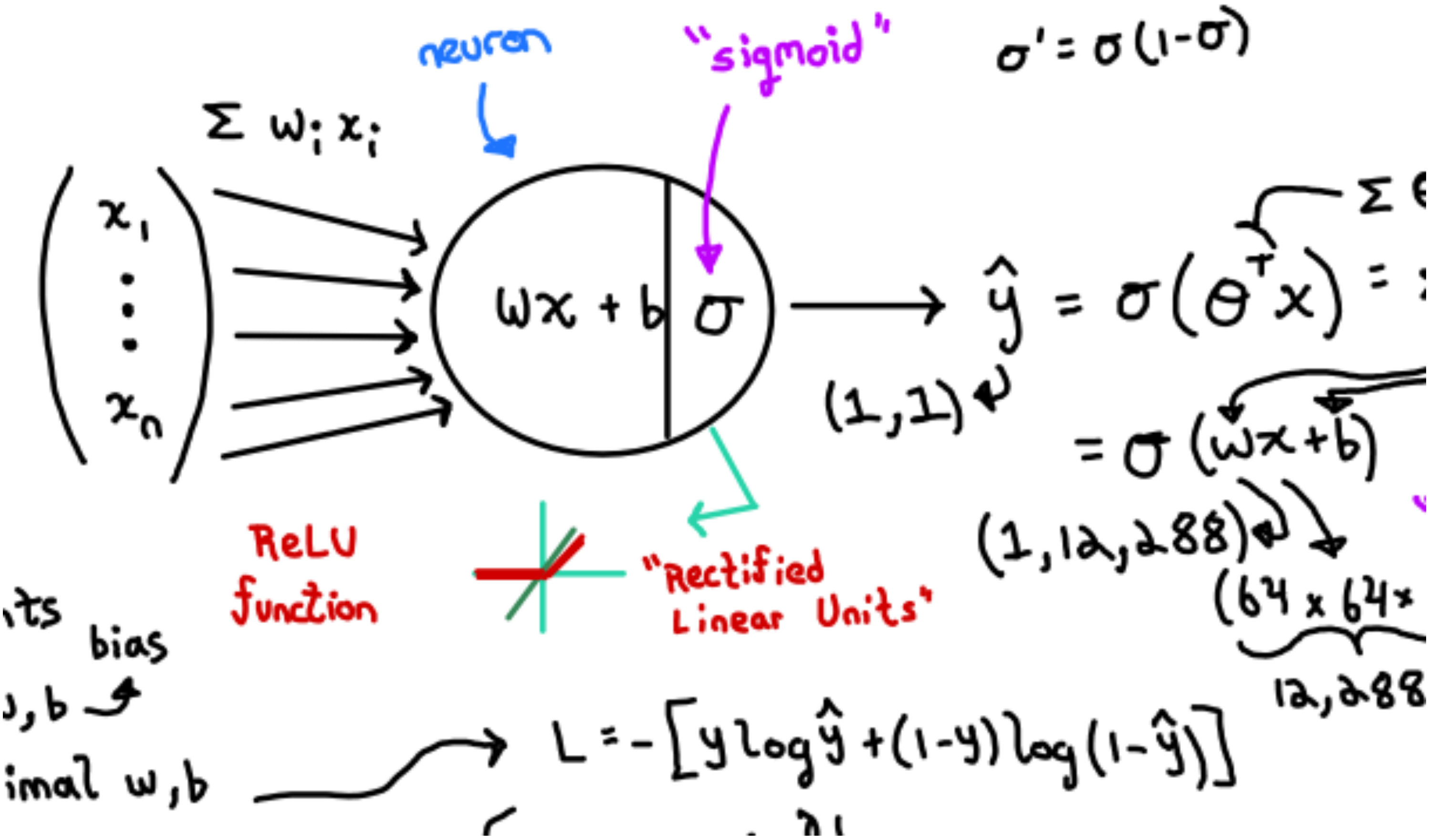


We could just use a ***linear function*** for the ***sigmoid***, but this becomes a linear regression:



The whole network becomes a ***linear regression***

Another one that is very common in ***deep learning*** is called the ***ReLU (Rectified Linear Units) function***. It is a function that is almost ***linear***, but for every input that is negative, it's equal to **0**. Because we cannot have negative age, it makes sense to use this one



Now, what else would you change?

We talked about ***linear regression***. The loss function we were using in ***linear regression*** was one of these two:

| y^ - y | , this comparison between the output label and **y^**, the prediction

or it was the **L2 loss**:

|| y^ - y ||$^2_2$, **y^** minus **y** in **L2** norm

So that's what we would use. We would modify our loss function to fit the ***regression*** type of problem. And the reason we would use this ***loss*** instead of the one we have for a ***regression*** task is because in ***optimization***, the shape of this loss is much easier to optimize for a ***regression*** task than it is for a ***classification*** task and vice versa

Neural Networks:

We stick to our first goal where given an image, tell us if there is cat or no cat

Neural Networks:

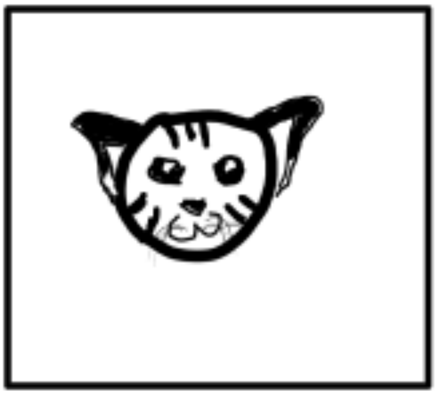
goal: image ⇒ cat (1) vs no cat (0)

Now we're going to make the network a little more complex by adding some parameters

We get our picture of the cat and flatten it:

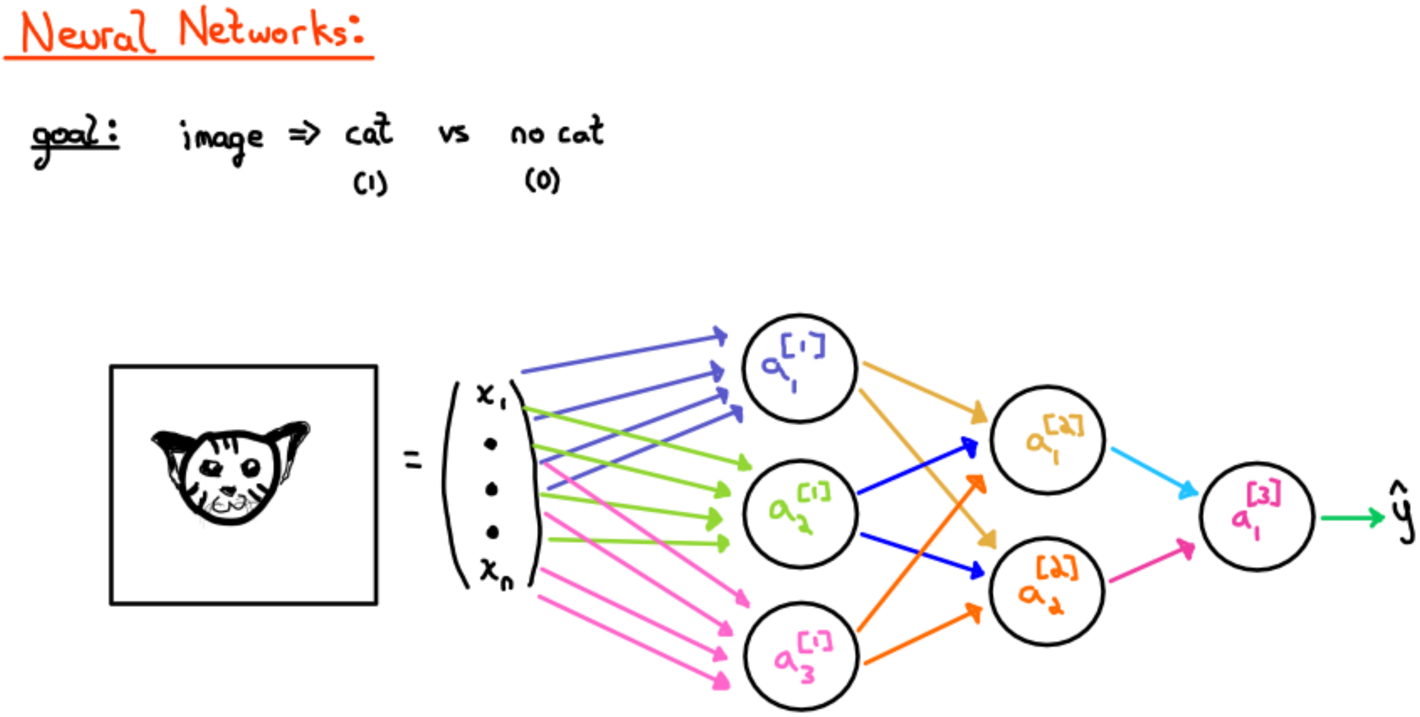
Neural Networks:

goal: image ⇒ cat (1) vs no cat (0)



$$= \begin{pmatrix} x_1 \\ \bullet \\ \bullet \\ \bullet \\ x_n \end{pmatrix}$$

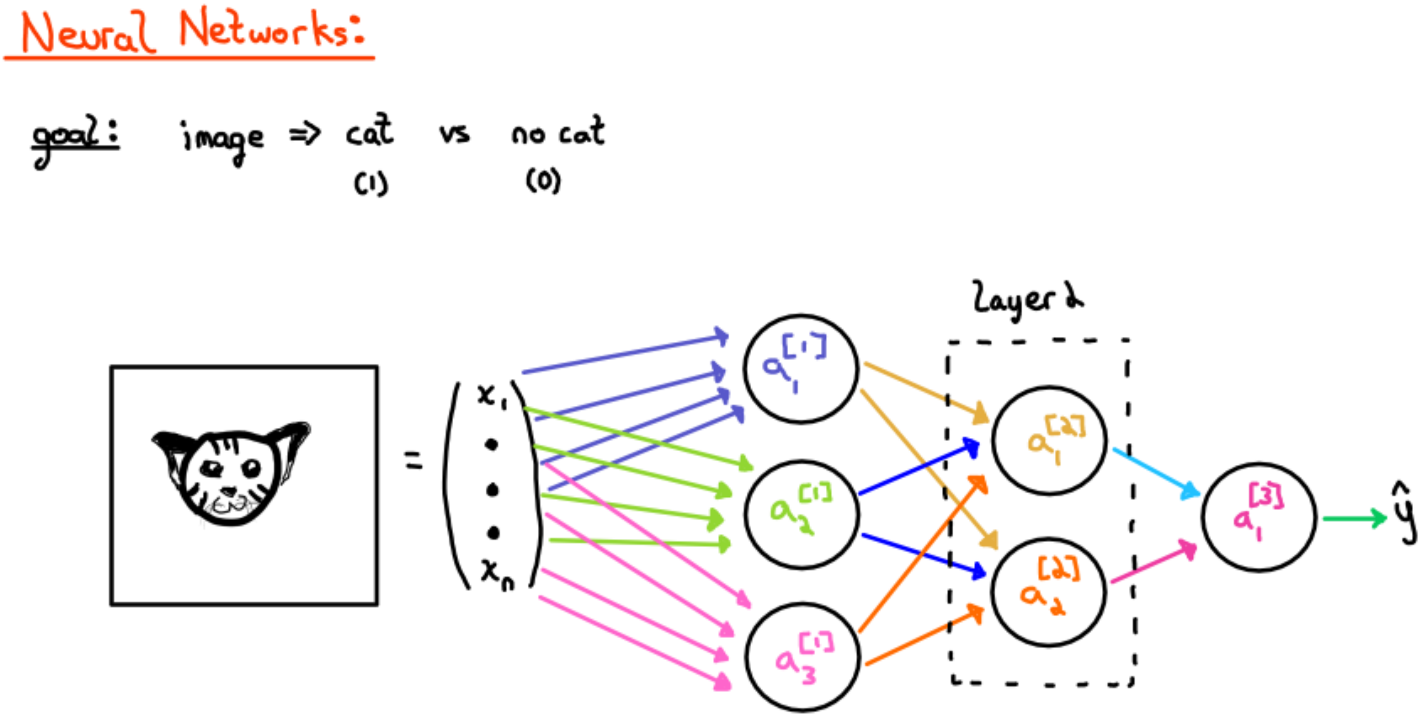
We're going to put more ***neurons*** than before:



Neural Networks:

goal: image ⇒ cat vs no cat
(1) (0)

So using the same notation, you can see that the **[]** indicates that there are different layers here



Neural Networks:

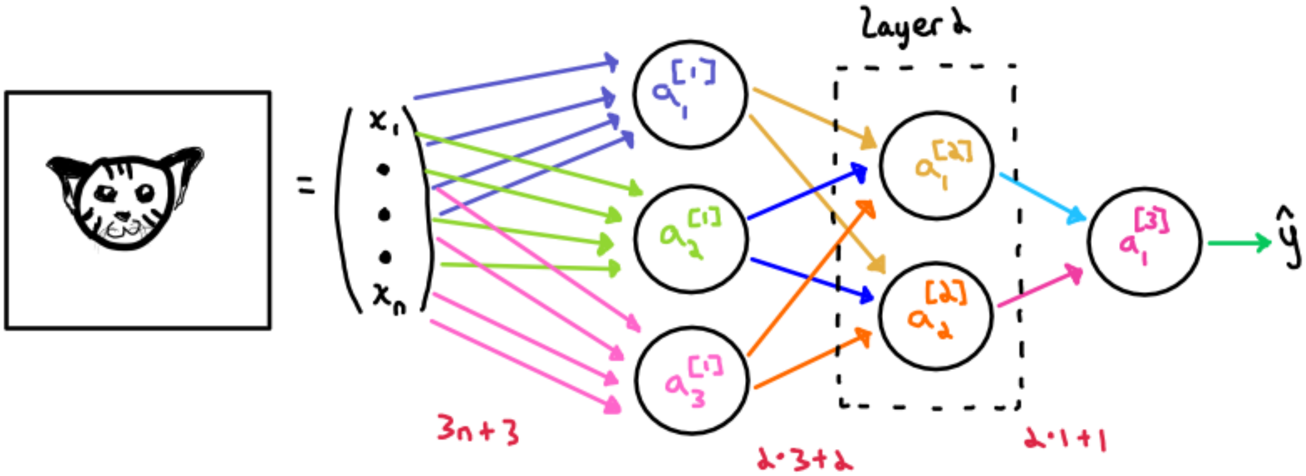
goal: image ⇒ cat vs no cat
(1) (0)

Notice that when you make a choice of ***architecture***, you have to be careful of one thing, which is that the ***output layer*** has to have the same number of neurons as you want the number of classes to be for ***reclassification*** and one for ***regression***

How many parameters does this network have?

Neural Networks:

goal: image ⇒ cat (1) vs no cat (0)



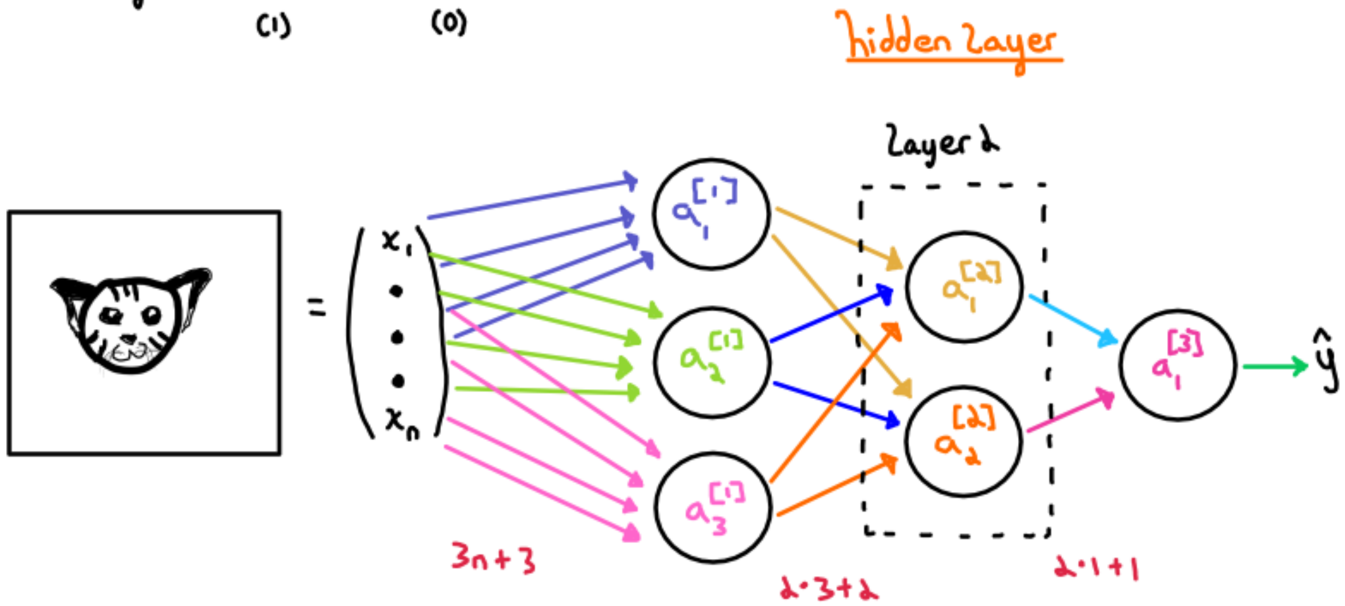
You would have **3n weights + 3 biases**, **2 x 3 weights + 2 biases**, **1 x 2 weights + 1 bias** , which is the total number of parameters

Layer denotes ***neurons*** that are not connected to each other. We call this cluster of ***neurons*** a ***layer***

Our ***network*** has three ***layers***. We would use the ***input layer*** to define the ***first layer***, the ***output layer*** to define the ***third layer*** because it directly sees the output, and we would call the ***second layer*** a ***hidden layer***

Neural Networks:

goal: image => cat vs no cat
(1) (0)



The reason we call it ***hidden*** is because the ***inputs*** and the ***outputs*** are ***hidden*** from this ***layer***. It means the only thing that this ***layer*** sees as input, is what the ***previous layer*** gave it. So it's an abstraction of the input, but it's not the inputs. It doesn't see the output, it just gives what it understood to the ***last neuron*** that will compare the output to the ground truth

Why are ***neural networks*** interesting and why do we call this a ***hidden layer***?

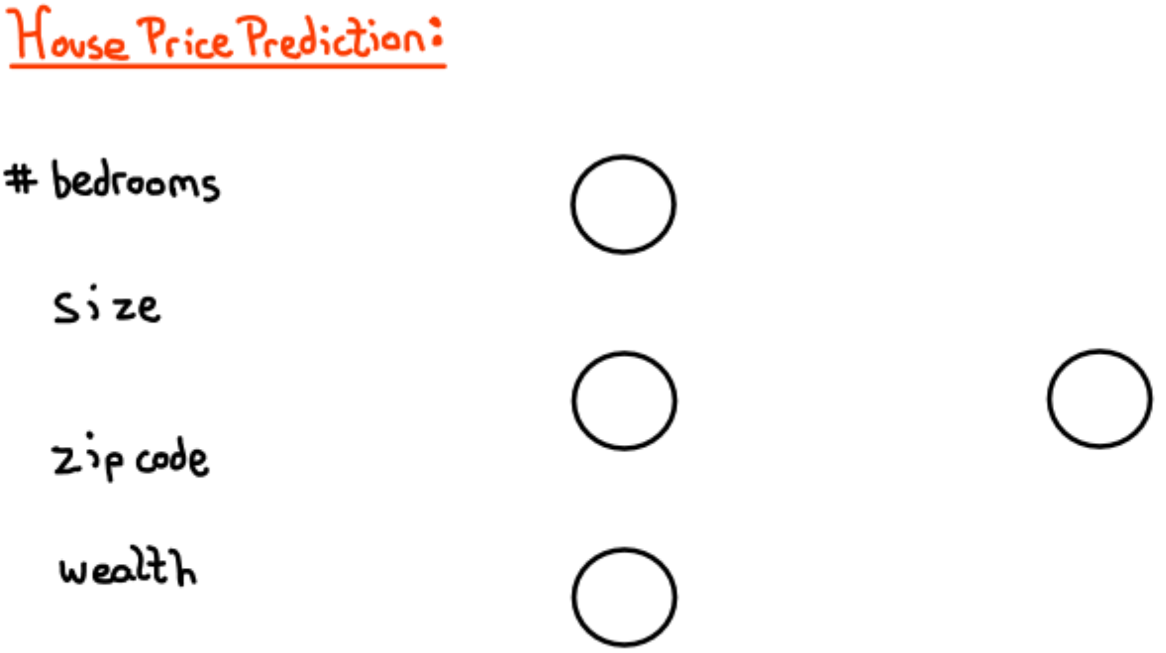
It's because if you train this network on ***cat classification*** with a lot of images of cats, you would notice that the ***first layers*** are going to understand the fundamental concepts of the image, which is the ***edges***

Each ***neuron*** is going to be able to detect different types of ***edges*** and what's going to happen is that these ***neurons*** are going to communicate what they found on the image to the ***next layer's neuron***. And this next ***neuron*** is going to use the ***edges*** that those previous ***neurons*** found to figure out that there are ears, and another ***neuron*** in this ***layer*** is going to figure out that there is a mouth, and so on if you have several ***neurons***. They're going to communicate what they understood to the ***output neuron*** that is going to construct the face of the cat based on what it received and be able to tell if there is a cat or not

The reason it's called ***hidden layer*** is because we don't really know what it's going to figure out, but with enough data, it should understand very complex information about the data. The deeper you go, the more complex information the ***neurons*** are able to understand

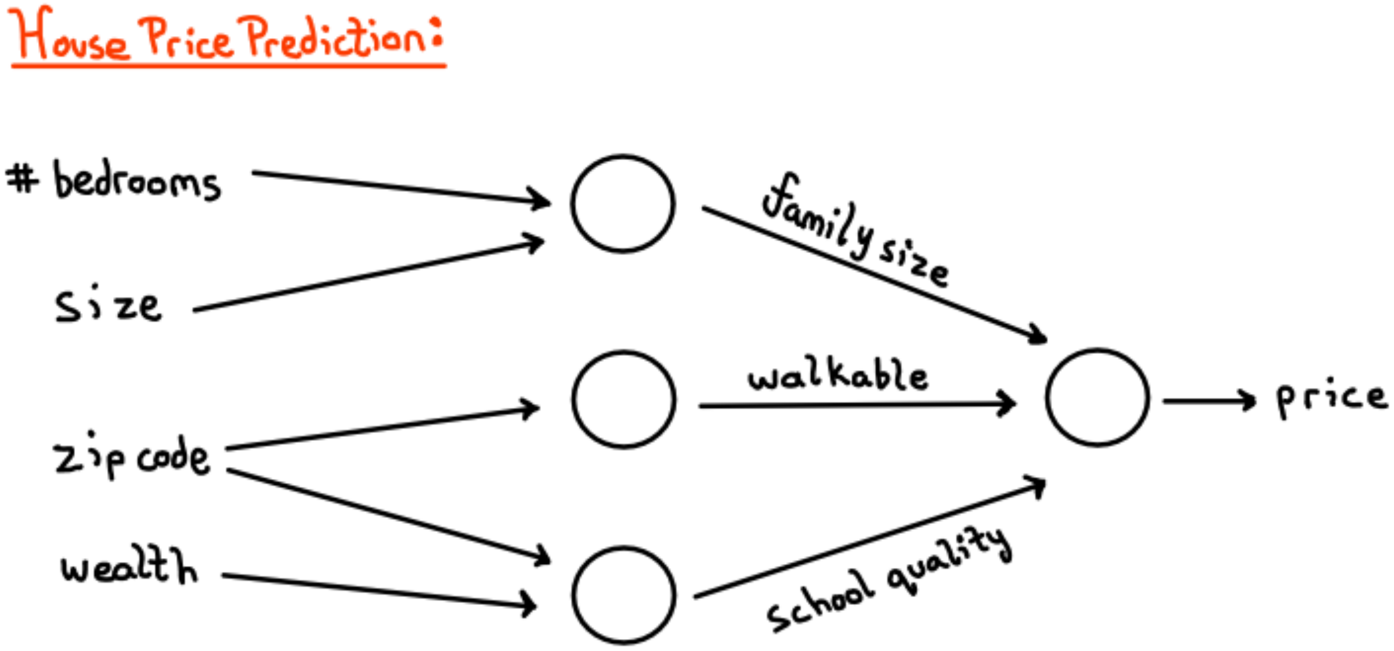
To give another example, we will use a house prediction example:

Let's assume that our inputs are **# of bedrooms**, ***size of the house***, ***zip code***, and ***wealth of the neighborhood***. What we will build is a ***network*** that has **3** ***neurons*** in the ***first layer*** and **1** ***neuron*** in the ***output layer***



What's interesting is that as a human, if you were to build this ***network*** and hand engineer it, you would say that zip code and wealth are about to tell us about the school quality in the neighborhood. The quality of the school that is next to the house probably. As a human, you would say these are probably good features to predict that. The zip code is going to tell us if the neighborhood is walkable or not, probably. The size and the number of bedrooms is going to tell us what's the size of the family that can fit in this house

These three are probably better information than these in order to finally predict the price

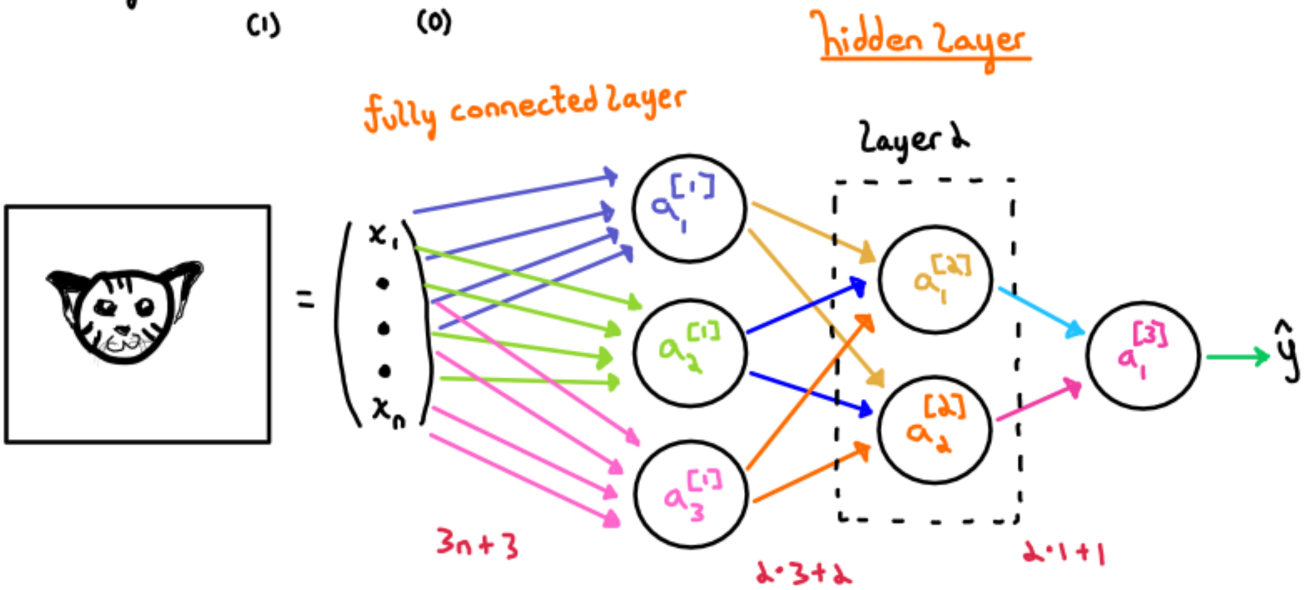


So that's a way to hand engineer that by hand as a human in order to give human knowledge to the ***network*** to figure out the price

In practice, what we do here is that we use a ***fully connected layer***

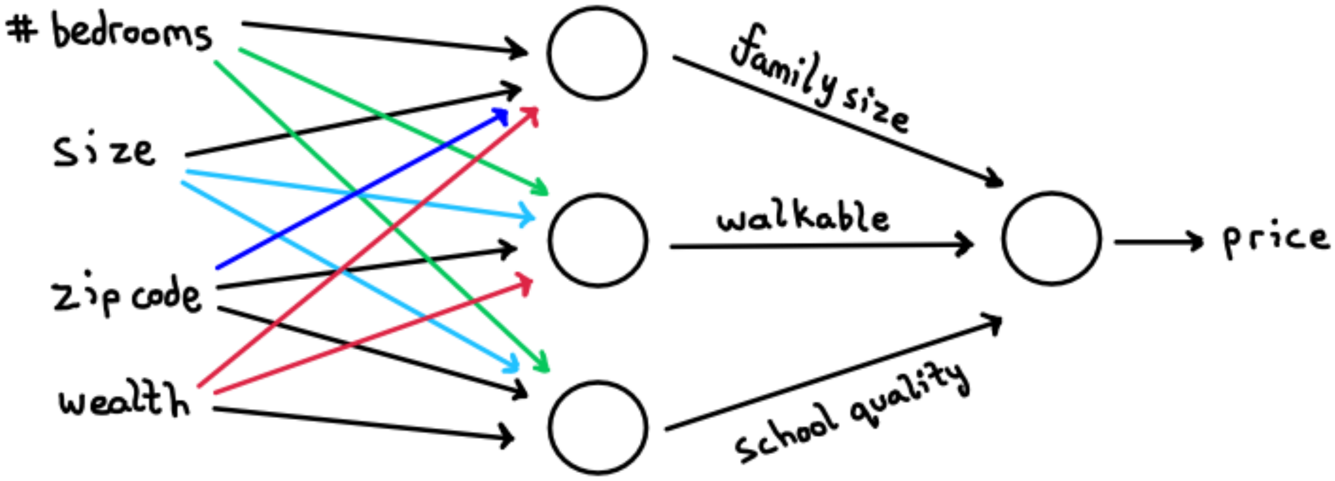
Neural Networks:

goal: image => cat vs no cat
(1) (0)

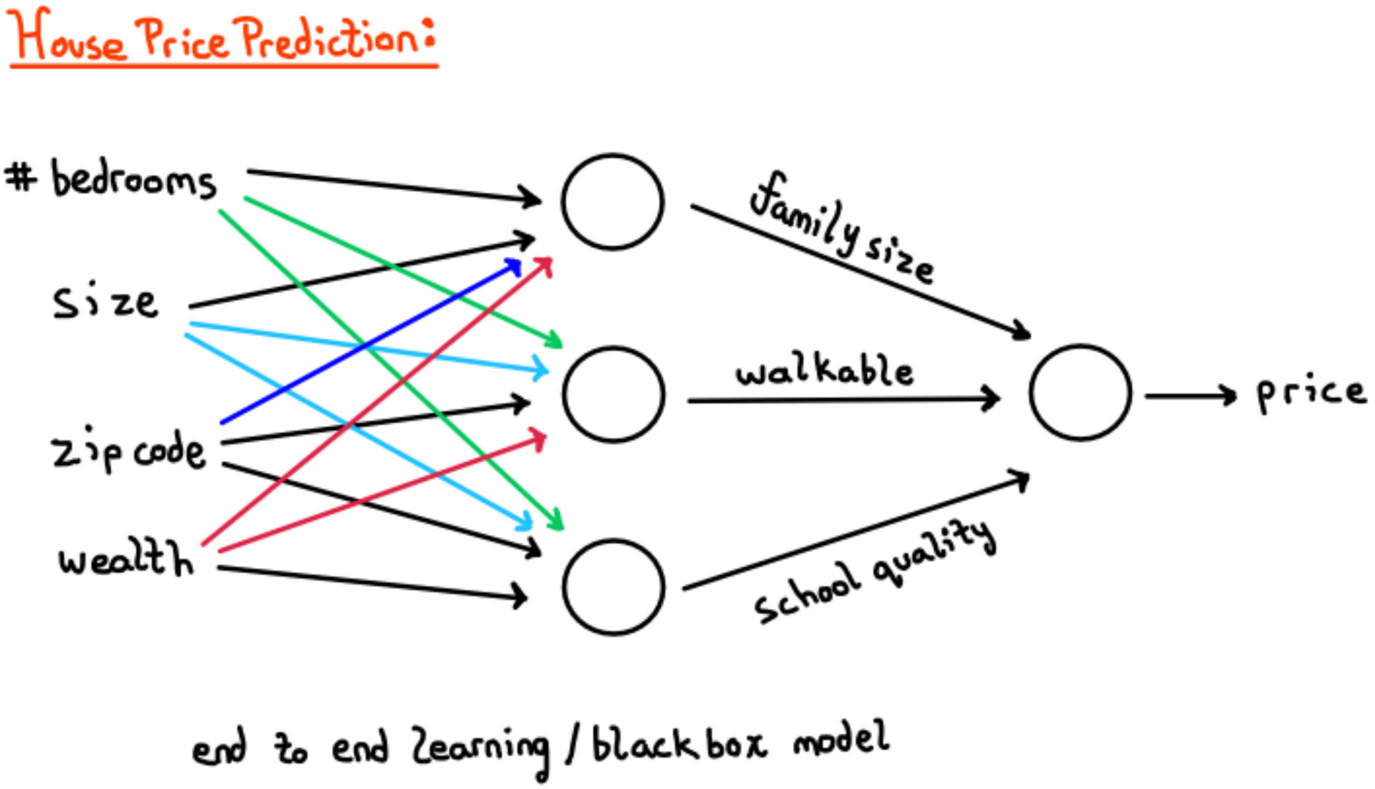


Fully connected layer means that we connect every input to the ***first layer***, every output of the ***first layer*** to the input of the ***second layer***, and so on. So all the ***neurons*** from one ***layer*** to another are connected with each other. What we're saying is that we will let the ***network*** figure these out. We will let the ***neurons*** of the ***first layer*** figure out what's interesting for the ***second layer*** to make the price prediction. So we will not tell these (***features***) to the ***network***, instead we will fully connect the ***network*** and so on

House Price Prediction:



We'll fuly connect the ***network*** and let it figure out what are the interesting ***features*** and oftentimes the ***network*** is going to be able, better than humans, to find what are the features that are representative. Sometimes you may hear ***neural networks*** referred as ***black box models***. The reason is we will not understand what an ***edge*** will correspond to. It's hard to figure out that a particular ***neuron*** is detecting a weighted average of the input features. Another word you might hear is ***end-to-end learning***. The reason we talked about ***end-to-end learning*** is because we have an input, a ground truth (price), and we don't constrain the ***network*** in the middle, we let it learn whatever it has to learn and we call it ***end-to-end learning*** because we are just training based on the input and the output



The ***neural network*** that we have here which has an ***input layer***, a ***hidden layer***, and an ***output layer***. We'll try to write down the equations that run the inputs and propagate it to the output

First we have $\mathbf{z^{[1]}}$ which is the ***linear*** part of the ***first layer*** that is computer $\mathbf{w^{[1]}x+b^{[1]}}$
Then this $\mathbf{z^{[1]}}$ is given to an ***activation*** (let's say it's ***sigmoid***)

$\mathbf{z^{[2]}}$ is then the ***linear*** part of the ***second neuron*** which is going to take the output of the ***previous layer***, multiply it by its ***weights*** and add the ***bias***
The second ***activation*** is going to take the ***sigmoid*** of $\mathbf{z^{[2]}}$

Finally we have the ***third layer*** which is going to multiply its ***weights*** with the output of the ***layer*** preceeding it and add its ***bias***
And finally we have the ***third activation*** which is simply the ***sigmoid*** of $\mathbf{z^{[3]}}$

Propagation equations:

$$z^{[1]} = w^{[1]} x + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

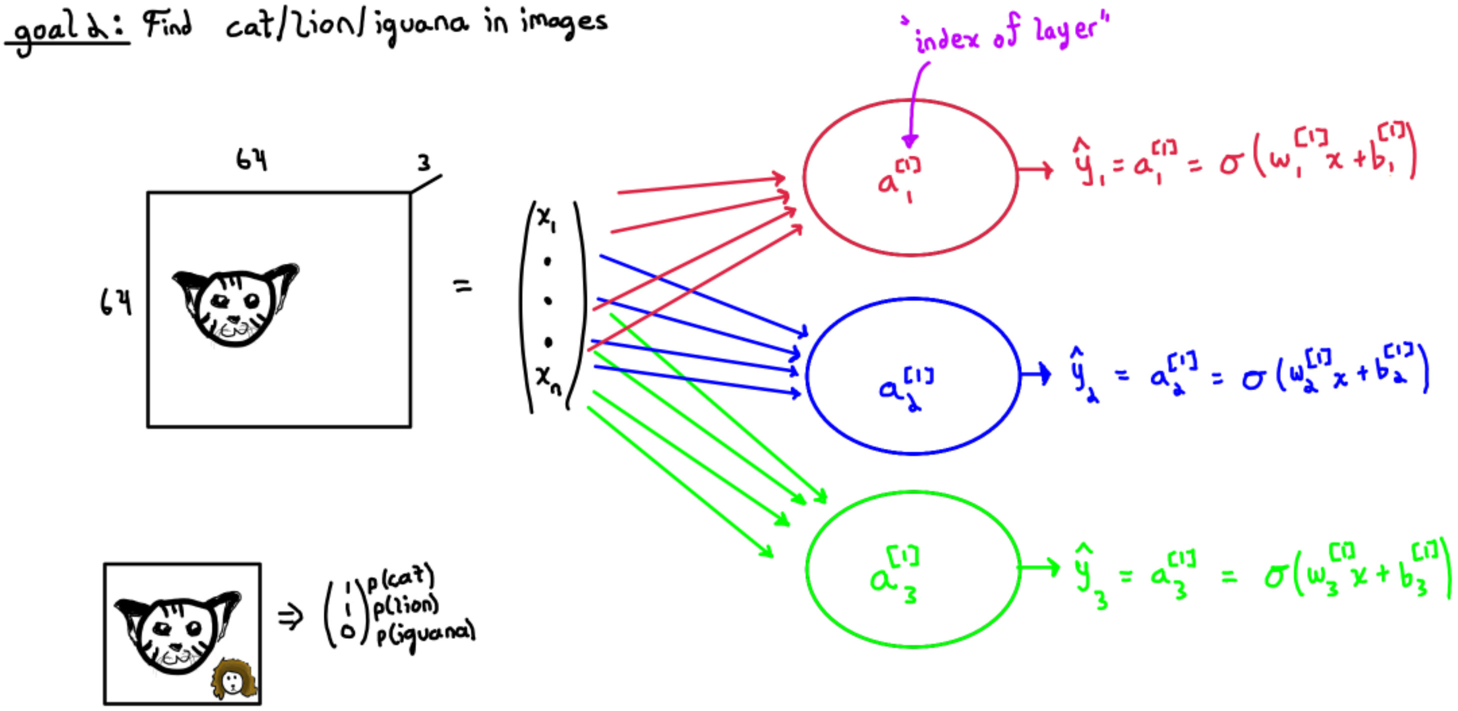
$$z^{[2]} = w^{[2]} \cdot a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

$$z^{[3]} = w^{[3]} \cdot a^{[2]} + b^{[3]}$$

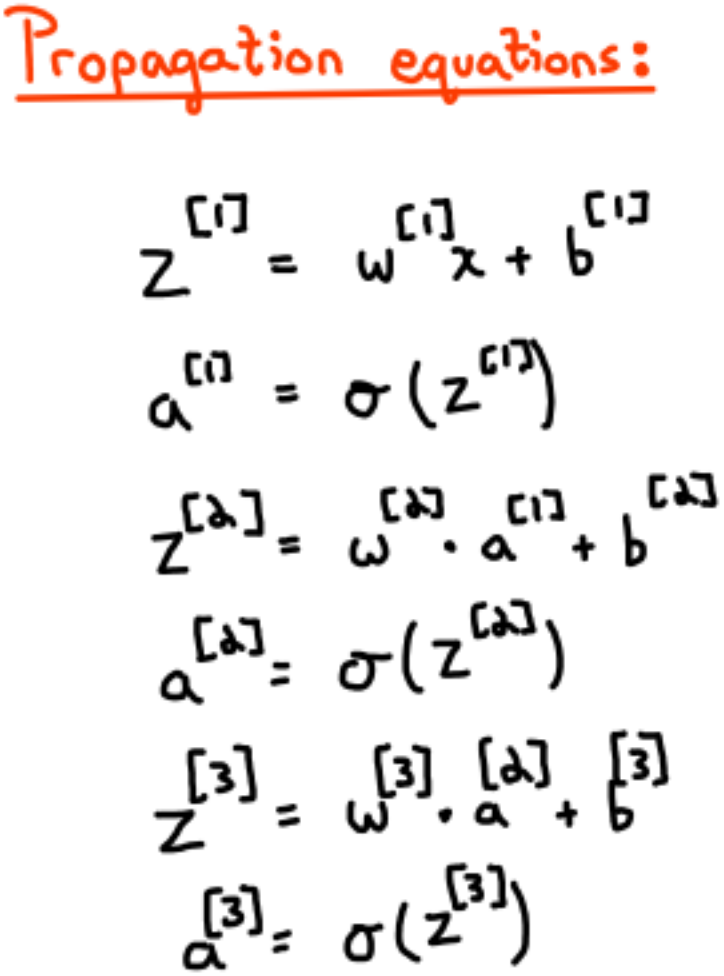
$$a^{[3]} = \sigma(z^{[3]})$$

What is interesting to notice between these equations that we wrote here (above) and the equations that we wrote here (below), is that we put everything in matrices


goal 2: Find cat/lion/iguana in images
"index of layer"
64
3
64
=
x_1
x_n
$a_1^{[1]}$
$\hat{y}_1 = a_1^{[1]} = \sigma(w_1^{[1]} x + b_1^{[1]})$
$a_2^{[1]}$
$\hat{y}_2 = a_2^{[1]} = \sigma(w_2^{[1]} x + b_2^{[1]})$
$a_3^{[1]}$
$\hat{y}_3 = a_3^{[1]} = \sigma(w_3^{[1]} x + b_3^{[1]})$
⇒
1
1
0
p(cat)
p(lion)
p(iguana)

For **3 *neurons*** we wrote three equations (above)

Here, for three ***neurons*** in the ***second layer***, we just wrote a single equation to summarize it



Propagation equations:

$$Z^{[1]} = w^{[1]}x + b^{[1]}$$

$$a^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = w^{[2]} \cdot a^{[1]} + b^{[2]}$$

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

$$Z^{[3]} = w^{[3]} \cdot a^{[2]} + b^{[3]}$$

$$a^{[3]} = \sigma(Z^{[3]})$$

But the shape of these things are going to be vectors

Propagation equations:

$$(3,1) \rightarrow z^{[1]} = \overset{(3,n)}{w^{[1]}} \, \overset{(n,1)}{x} + b^{[1]} \leftarrow (3,1)$$

$$(3,1) \rightarrow a^{[1]} = \sigma(z^{[1]})$$

$$(2,1) \rightarrow z^{[2]} = \overset{(2,3)}{w^{[2]}} \cdot \overset{(3,1)}{a^{[1]}} + b^{[2]} \leftarrow (2,1)$$

$$(2,1) \rightarrow a^{[2]} = \sigma(z^{[2]})$$

$$(1,1) \rightarrow z^{[3]} = \overset{(1,2)}{w^{[3]}} \cdot \overset{(2,1)}{a^{[2]}} + b^{[3]} \leftarrow (1,1)$$

$$(1,1) \rightarrow a^{[3]} = \sigma(z^{[3]})$$

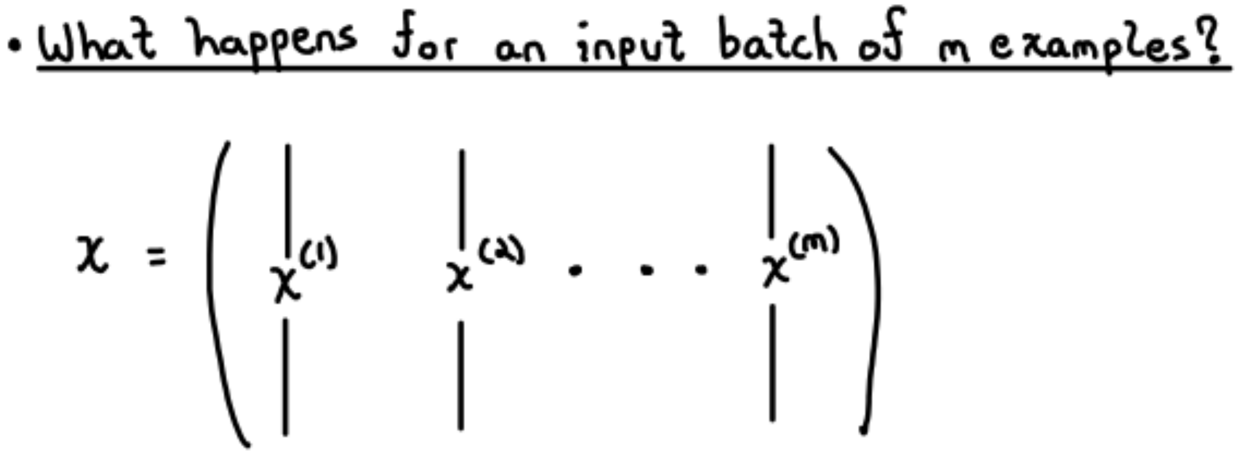
It's usually very helpful, even when coding these types of equations, to know all the shapes that are involved

What is interesting is that we will try to vectorize the code even more

The difference between ***stochastic gradient descent*** and ***gradient descent*** is that ***stochastic gradient descent*** updates the ***weights*** and the ***bias*** after you see every example, so the direction of the ***gradient*** is quite noisy. It doesn't represent very well the entire batch. While ***gradient descent*** or ***batch gradient descent*** updates after you've seen the whole batch of examples and the gradient is much more precise. It points to the direction you want to go to

What we're trying to do now is to write down these equations if instead of giving one single cat image, we had given a bunch of images that either have a cat or not have a cat

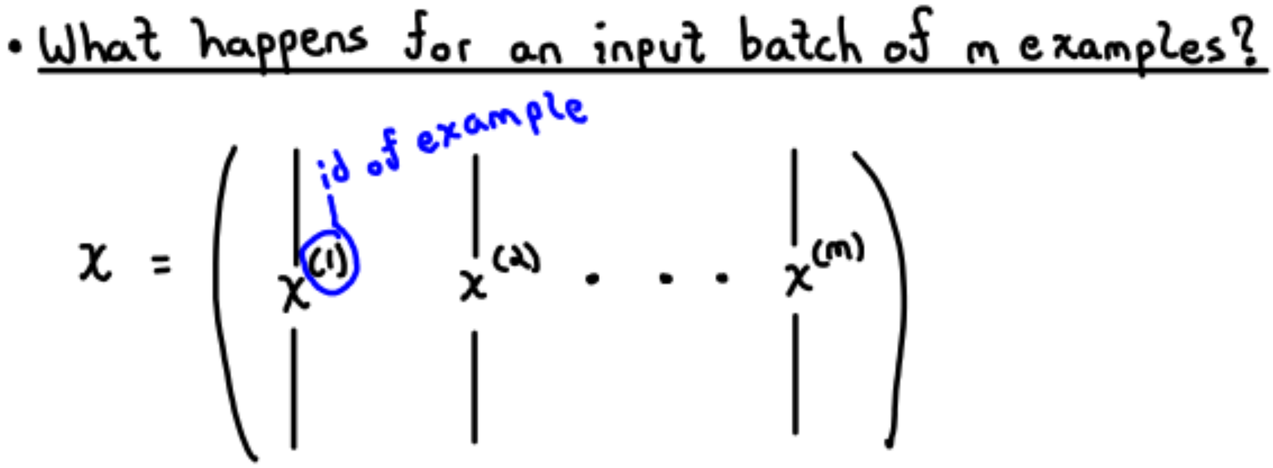
So now, our input **x** is no longer a single column vector, it's a matrix with the first image corresponding to $\mathbf{x}^{(1)}$, the second image corresponding to $\mathbf{x}^{(2)}$, and so on until the nth image corresponding to $\mathbf{x}^{(n)}$



- What happens for an input batch of m examples?

$$X = \begin{pmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{pmatrix}$$

We're introducing a new notation which is the parentheses superscript corresponding to the ***ID*** of the example



- What happens for an input batch of m examples?

$$X = \begin{pmatrix} | & | & & | \\ x^{\overset{\text{id of example}}{\boxed{(1)}}} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{pmatrix}$$

So, **[]** for the ***layer***, **()** for the ID of the example we are talking about

To give more context on what we're trying to do:

Propagation equations:

$$(3,1) \rightarrow z^{[1]} = \overset{(3,n)}{w^{[1]}}\,\overset{(n,1)}{x} + b^{[1]} \leftarrow (3,1)$$

$$(3,1) \rightarrow a^{[1]} = \sigma(z^{[1]})$$

$$(2,1) \rightarrow z^{[2]} = \overset{(2,3)}{w^{[2]}} \cdot \overset{(3,1)}{a^{[1]}} + b^{[2]} \leftarrow (2,1)$$

$$(2,1) \rightarrow a^{[2]} = \sigma(z^{[2]})$$

$$(1,1) \rightarrow z^{[3]} = \overset{(1,2)}{w^{[3]}} \cdot \overset{(2,1)}{a^{[2]}} + b^{[3]} \leftarrow (1,1)$$

$$(1,1) \rightarrow a^{[3]} = \sigma(z^{[3]})$$

We that this is a bunch of operations

We just have a ***network*** with ***input***, ***hidden***, and ***output layers***. We could have a network with **1,000 *layers***. The more layers we have, the more computation, and it quickly goes up. So what we want to do is be able to parallelize our code or our computation as much as possible by giving batches of inputs and parallelizing these equations

So let's see how these equations are modified when we give it a batch of **m** inputs

We will use capital letters to denote the equivalent of the lowercase letters, but for a batch of input

- **What happens for an input batch of m examples?**

$$X = \begin{pmatrix} | & | & & | \\ x^{\overset{\text{id of example}}{\boxed{(1)}}} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{pmatrix}$$

$$z^{[1]} = w^{[1]} x + b^{[1]}$$

$$= \begin{bmatrix} | & & | \\ z^{[1](1)} & \cdots & z^{[1](m)} \\ | & & | \end{bmatrix}$$

Then we have to figure out what have to be the shapes of these equations in order to end up with this

• What happens for an input batch of m examples?

$$X = \begin{pmatrix} | & | & & | \\ x^{\overset{\text{id of example}}{(1)}} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{pmatrix}$$

$$\underset{(3,m)}{z^{[1]}} = \overset{(3,n)}{w^{[1]}} \, \overset{(n,m)}{x} + \overset{(3,1)}{b^{[1]}}$$

$$= \begin{bmatrix} | & & | \\ z^{[1](1)} & \cdots & z^{[1](m)} \\ | & & | \end{bmatrix}$$

The shape of **w**$^{[1]}$ doesn't change. It's not because we will give **1,000** inputs to our ***network*** that the parameters are going to be more. So the parameter number stays the same even if we give more inputs, so it has to be **(3,n)** in order to match **Z**$^{[1]}$

The interesting thing is that there is an algebraic problem here. We said that the number of parameters doesn't change which means that **w** has the same shape as it had before. **b** should have the same shape as it had before as well, so it should be **(3,1)**

The problem with this equation is that we're summing a **(3,m)** matrix to a **(3,1)** vector. This is not possible in math, it doesn't work. When you do some summations or substractions, you need the two terms to be the same shape because you will do an element-wise addition or an element-wise subtraction

The trick that is used here is a technique called ***broadcasting***

Broadcasting is the fact that we don't want to change the number of parameters, it should stay the same, but we still want this operation to be able to be written in parallel version. So we still want to write this equation because we want to parallelize our code, but we don't want to add more parameters, it doesn't make sense. So what we're going to do is that we are going to create a vector **b~**$^{[1]}$ which is going to be **b**$^{[1]}$ repeated **m** times

- What happens for an input batch of m examples?

$$X = \begin{pmatrix} | & | & & | \\ \overset{\text{id of example}}{x^{(1)}} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{pmatrix}$$

Broadcasting

$$\underset{(3,m)}{Z^{[1]}} = \overset{(3,n)}{W^{[1]}}\,\overset{(n,m)}{X} + \overset{(3,1)}{b^{[1]}} \longleftarrow \tilde{b}^{[1]} = \underbrace{\begin{pmatrix} b^{[1]} & b^{[1]} & \cdots & b^{[1]} \end{pmatrix}}_{m}$$

$$= \begin{bmatrix} | & & | \\ z^{[1](1)} & \cdots & z^{[1](m)} \\ | & & | \end{bmatrix}$$

So we just keep the same number of parameters but just repeat them in order to be able to write our code in parallel. This is called ***broadcasting***

What is convenient is that in ***Python*** there is a package that is often used to code these equations, it's ***numPy***. So ***numpy*** will directly do the ***broadcasting***. It means, if you sum this **(3,m)** matrix with a **(3,1)** parameter vector, it's going to automatically reproduce the parameter vector **m** times so that the equation works. It's called ***broadcasting***

Because we're using this technique, we're able to rewrite all these equations with capital letters

# How is this different from ***principal component analysis***?

This is a supervised learning algorithm that will be used to predict the price of a house. ***Principal component analysis*** doesn't predict anything, it gets an input matrix **X**, ***normalizes*** it, computes the ***covariance matrix***, and then figures out what are the ***principal components*** by doing the ***eigenvalue decomposition***. But the outcome of ***PCA*** is you know that the most important features of your dataset **X** are going to be these features. Here we're not looking at the features, we're only looking at the output. That's what is important to us

# How do you decide how many ***neurons*** per ***layer***? How many ***layers***? What's the ***architecture*** of your ***neural network***?

There are two things to take into consideration. First, nobody knows the right answer, so you have to test it. So what we would do is we would try ten different ***architectures***, train the ***network*** on these, look at the ***validation set accuracy*** of all these, and decide which one seems to be the best. That's how we figure out what's the right ***network size***

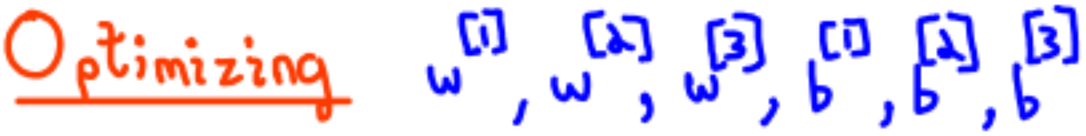
On top of that, using experience is often valuable. So if you're given a problem, you always try to gauge how complex the problem is. Based on their complexity, the more complex a problem usually is, the more data you need in order to figure out the output, the more deeper the ***network*** should be

Let's try to write the loss function for this problem

Now that we have our ***network***, we have written this ***propagation equation*** and we call it ***forward propagation*** (because it's going forward from the input to the output). ***Backward propagation*** is when we are starting from the ***loss*** and go backwards

Optimization problem:

We have a lot of stuff to optimize. We have to find the right values for these:

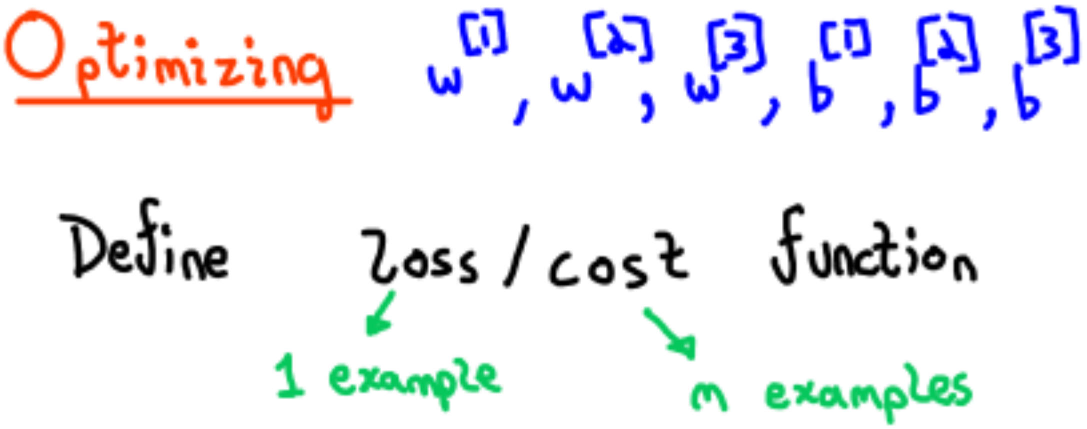


Optimizing $w^{[1]}, w^{[2]}, w^{[3]}, b^{[1]}, b^{[2]}, b^{[3]}$

Remember that ***model*** equals ***architecture*** plus ***parameter***. We have our ***architecture***, if we have our parameters, we're done

So in order to do that, we have to define an ***objective function*** (sometimes called ***loss***, sometimes called ***cost function***)

Usually we would call it ***loss*** if there is only one example in the batch, and ***cost*** if there are multiple examples in a batch



Optimizing $w^{[1]}, w^{[2]}, w^{[3]}, b^{[1]}, b^{[2]}, b^{[3]}$

Define loss / cost function

1 example
m examples

The ***cost function* J** depends on **y^** and **y**

Optimizing $w^{[1]}, w^{[2]}, w^{[3]}, b^{[1]}, b^{[2]}, b^{[3]}$

Define loss / cost function

loss ↙ 1 example

cost ↘ m examples

"cost function"

↪ $J(\hat{y}, y)$

y^ is $\mathbf{a}^{[3]}$

Propagation equations:

$$(3,1) \rightarrow Z^{[1]} = \overset{(3,n)}{W^{[1]}} \overset{(n,1)}{x} + b^{[1]} \leftarrow (3,1)$$

$$(3,1) \rightarrow a^{[1]} = \sigma(z^{[1]})$$

$$(2,1) \rightarrow Z^{[2]} = \overset{(2,3)}{W^{[2]}} \cdot \overset{(3,1)}{a^{[1]}} + b^{[2]} \leftarrow (2,1)$$

$$(2,1) \rightarrow a^{[2]} = \sigma(Z^{[2]})$$

$$(1,1) \rightarrow Z^{[3]} = \overset{(1,2)}{W^{[3]}} \cdot \overset{(2,1)}{a^{[2]}} + b^{[3]} \leftarrow (1,1)$$

$$(1,1) \rightarrow \underset{\hat{y}\ =}{a^{[3]}} = \sigma(z^{[3]})$$

And we will set it to be the sum of the ***loss functions*** $\mathbf{L}^{(i)}$ and we'll ***normalize*** it (it's not mandatory) with **1/m**

Optimizing $w^{[1]}, w^{[2]}, w^{[3]}, b^{[1]}, b^{[2]}, b^{[3]}$

Define loss / cost function

1 example

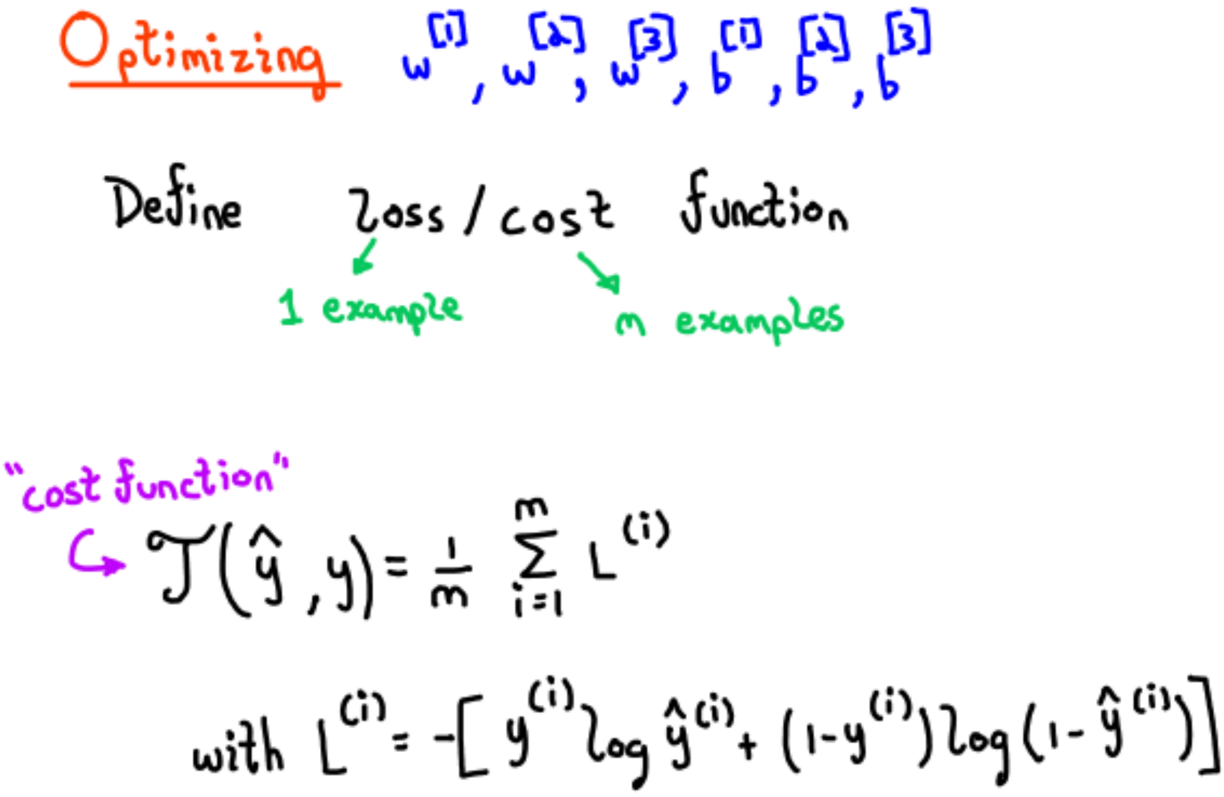
m examples

"cost function"

$$\mathcal{J}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^{m} L^{(i)}$$

This means that we are going for ***batch gradient descent***. We want to compute the ***loss function*** for the whole batch, parallelize our code, and then calculate the ***cost function*** that will be then derived to give us the direction of the gradients. That is, the average direction of all the derivations with respect to the whole input batch

And $\mathbf{L}^{(\mathbf{i})}$ will be the ***loss function*** corresponding to one input, so "what's the error on this specific one input?", and it will be the ***logistic loss***



Optimizing $w^{[1]}, w^{[2]}, w^{[3]}, b^{[1]}, b^{[2]}, b^{[3]}$

Define loss / cost function

1 example

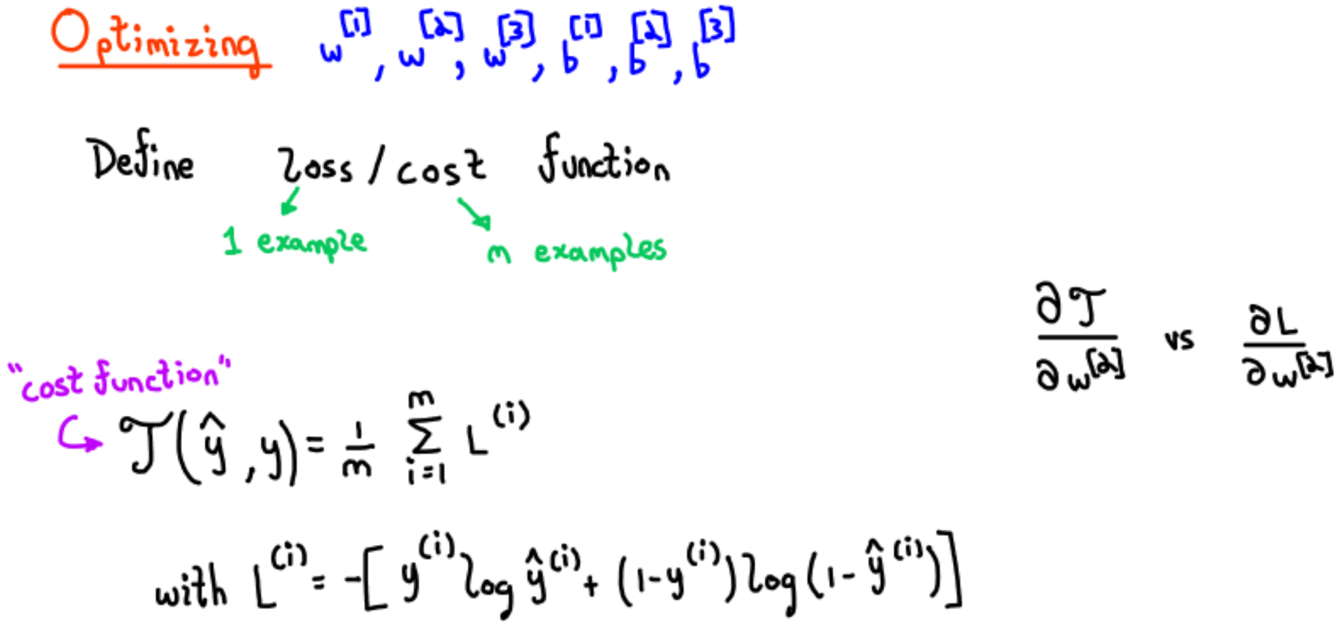
m examples

"cost function"

$$\hookrightarrow J(\hat{y}, y) = \frac{1}{m}\sum_{i=1}^{m} L^{(i)}$$

with $L^{(i)} = -[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$

Is it more complex to take a derivative of **J** with respect to the parameters? Or of **L**?



Optimizing $w^{[1]}, w^{[2]}, w^{[3]}, b^{[1]}, b^{[2]}, b^{[3]}$

Define loss / cost function

1 example

m examples

$$\frac{\partial J}{\partial w^{[2]}} \quad \text{vs} \quad \frac{\partial L}{\partial w^{[2]}}$$

"cost function"

$$\hookrightarrow J(\hat{y}, y) = \frac{1}{m}\sum_{i=1}^{m} L^{(i)}$$

with $L^{(i)} = -[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$

Which one is the hardest?

It doesn't matter because ***derivation*** is a ***linear operation***. So instead of computing all derivatives on **J**, we will compute them on **L**, but it's totally equivalent. There's just one more step at the end

We defined our ***loss function*** and the next step is to ***optimize***, so we have to compute a lot of derivatives, and that is called ***backward propagation***

Why is it called ***backward propagation***?

It's because what we want to ultimately do is this:

Backward Propagation:

$$\forall\ l = 1 \dots 3 \qquad \begin{cases} w^{[l]} = w^{[l]} - \alpha \dfrac{\partial J}{\partial w^{[l]}} \\ b^{[l]} = b^{[l]} - \alpha \dfrac{\partial J}{\partial b^{[l]}} \end{cases}$$

We want to do that for every parameter in ***layer*** **1**, **2**, and **3**. It means we have to compute all these derivatives. We have to compute derivative of the cost with respect to $\mathbf{w}^{[1]}$, $\mathbf{w}^{[2]}$, $\mathbf{w}^{[3]}$, $\mathbf{b}^{[1]}$, $\mathbf{b}^{[2]}$, $\mathbf{b}^{[3]}$

We've done it with ***logistic regression***, we're going to do it with a ***neural network***

Which derivative do we want to start with?

We start with $\mathbf{w}^{[3]}$ because if you look at this ***loss function***:

"cost function"

$$\hookrightarrow J(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^{m} L^{(i)}$$

$$\text{with } L^{(i)} = -\left[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)}) \right]$$

and if you want to understand how much we should move $\mathbf{w}^{[1]}$ in order to make the loss move, it's much more complicated than answering the question “how much should $\mathbf{w}^{[3]}$ move to move the loss?” because there's much more connections, if you want to compute with $\mathbf{w}^{[1]}$

That's why we call it ***backward propagation***. It's because we will start with the ***top layer***, the one that's the closest to the ***loss function***, derive the derivative of **J** with respect to $\mathbf{w}^{[1]}$, and once we've computed this derivative:

Backward Propagation:

$$\forall\ l = 1 \ldots 3 \qquad \begin{cases} w^{[l]} = w^{[l]} - \alpha \dfrac{\partial J}{\partial w^{[l]}} \\ b^{[l]} = b^{[l]} - \alpha \dfrac{\partial J}{\partial b^{[l]}} \end{cases}$$

$$\frac{\partial J}{\partial w^{[3]}}$$

$$\frac{\partial J}{\partial w^{[2]}}$$

we will be able to compute this one very easily. Why very easily? Because we can use the chain rule of calculus

If we had to compute this derivative:

$$\frac{\partial J}{\partial w^{[3]}}$$

what we will do is separate it into several derivatives that are easier:

$$\frac{\partial J}{\partial w^{[3]}} = \frac{\partial J}{\partial ?} \cdot \frac{\partial ?}{\partial w^{[3]}}$$

What should this **?** be?

We know that **J** depends on **y^** and we know that **y^** depends on **Z[3]**. **y^** is the same thing as **a[3]**. So we will make a quick hack and say that this derivative is the same as taking it with respect to **a[3]**:

$$\frac{\partial J}{\partial w^{[3]}} = \frac{\partial J}{\partial ?} \cdot \frac{\partial ?}{\partial w^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

We know these derivatives are pretty easy to compute. That's why we call it ***backpropagation***. It's because we will use the ***chain rule*** to compute the derivative of **w[3]** and then when we want to do it for **w[2]**, we're going to insert:

$$\frac{\partial J}{\partial w^{[3]}} = \frac{\partial J}{\partial ?} \cdot \frac{\partial ?}{\partial w^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

$$\frac{\partial J}{\partial w^{[2]}} = \frac{\partial J}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}}$$

These two are the same thing:

$$\frac{\partial J}{\partial w^{[3]}} = \frac{\partial J}{\partial ?} \cdot \frac{\partial ?}{\partial w^{[3]}} = \underbrace{\frac{\partial J}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

$$\frac{\partial J}{\partial w^{[2]}} = \boxed{\frac{\partial J}{\partial z^{[3]}}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}}$$

It means, if we want to compute the derivative of $\mathbf{w}^{[2]}$, we don't need to recompute this derivative because we already did this for $\mathbf{w}^{[3]}$

Backward Propagation:

$$\forall\ l = 1 \ldots 3 \quad \begin{cases} w^{[l]} = w^{[l]} - \alpha \dfrac{\partial J}{\partial w^{[l]}} \\ b^{[l]} = b^{[l]} - \alpha \dfrac{\partial J}{\partial b^{[l]}} \end{cases}$$

$$\frac{\partial J}{\partial w^{[3]}} = \frac{\partial J}{\partial ?} \cdot \frac{\partial ?}{\partial w^{[3]}} = \underbrace{\frac{\partial J}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

$$\frac{\partial J}{\partial w^{[2]}} = \boxed{\frac{\partial J}{\partial z^{[3]}}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}}$$

$$\frac{\partial J}{\partial w^{[1]}} = \boxed{\frac{\partial J}{\partial z^{[2]}}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial w^{[1]}}$$

What's interesting about it is that we're not going to redo the work we did, we're just going to store the right values while ***backpropagating*** and continue to derivate

One thing that you need to notice though, is that you need this ***forward propagation*** equation:

Propagation equations:

$$(3,1) \rightarrow z^{[1]} = \overset{(3,n)}{w^{[1]}}\,\overset{(n,1)}{x} + b^{[1]} \leftarrow (3,1)$$

$$(3,1) \rightarrow a^{[1]} = \sigma(z^{[1]})$$

$$(2,1) \rightarrow z^{[2]} = \overset{(2,3)}{w^{[2]}} \cdot \overset{(3,1)}{a^{[1]}} + b^{[2]} \leftarrow (2,1)$$

$$(2,1) \rightarrow a^{[2]} = \sigma(z^{[2]})$$

$$(1,1) \rightarrow z^{[3]} = \overset{(1,2)}{w^{[3]}} \cdot \overset{(2,1)}{a^{[2]}} + b^{[3]} \leftarrow (1,1)$$

$$(1,1) \rightarrow \underset{\overset{\shortparallel}{\hat{y}}}{a^{[3]}} = \sigma(z^{[3]})$$

in order to remember what should be the path to take in your chain rule, you'll want to choose the path that you're going through in the proper way so that there's no cancellation in these derivatives

Backward Propagation:

$$\forall\ l = 1 \dots 3 \qquad \begin{cases} w^{[l]} = w^{[l]} - \alpha \dfrac{\partial J}{\partial w^{[l]}} \\ b^{[l]} = b^{[l]} - \alpha \dfrac{\partial J}{\partial b^{[l]}} \end{cases}$$

$$\frac{\partial J}{\partial w^{[3]}} = \frac{\partial J}{\partial ?} \cdot \frac{\partial ?}{\partial w^{[3]}} = \underbrace{\frac{\partial J}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

$$\frac{\partial J}{\partial w^{[2]}} = \boxed{\frac{\partial J}{\partial z^{[3]}}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}}$$

can't compute

$$\frac{\partial w^{[2]}}{\partial a^{[1]}}$$

$$\frac{\partial J}{\partial w^{[1]}} = \boxed{\frac{\partial J}{\partial z^{[2]}}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial w^{[1]}}$$

You cannot compute that, you don't know it