1. Introduction
   1. Launching the notebook server:
   2. Matrix math and Numpy refresher:
      1. Data dimensions

Scalars - 0 dimension

Vectors - 1 dimension

Matrix - 2 dimensions

Tensors – 2+ dimension – it’s tough to visualize. Each element will be represented by indices.

* + 1. Data in Numpy

Python is convenient, but it can also be slow. However, it does allow you to access libraries that execute faster code written in languages like C. NumPy is one such library: it provides fast alternatives to math operations in Python and is designed to work efficiently with groups of numbers - like matrices.NumPy is a large library and we are only going to scratch the surface of it here. If you plan on doing much math with Python, you should definitely spend some time exploring its [documentation](https://docs.scipy.org/doc/numpy/reference/) to learn more.

**Importing NumPy**

When importing the NumPy library, the convention you'll see used most often – including here – is to name it np, like so:

**import** numpy **as** np

Now you can use the library by prefixing the names of functions and types with np., which you'll see in the following examples.

**Data Types and Shapes**

The most common way to work with numbers in NumPy is through [ndarray](https://docs.scipy.org/doc/numpy/reference/arrays.html" \t "_blank) objects. They are similar to Python lists, but can have any number of dimensions. Also, ndarray supports fast math operations, which is just what we want.

Since it can store any number of dimensions, you can use ndarrays to represent any of the data types we covered before: scalars, vectors, matrices, or tensors.

**Scalars**

[Scalars in NumPy](https://docs.scipy.org/doc/numpy/reference/arrays.scalars.html) are a bit more involved than in Python. Instead of Python’s basic types like int, float, etc., NumPy lets you specify signed and unsigned types, as well as different sizes. So instead of Python’s int, you have access to types like uint8, int8, uint16, int16, and so on.

These types are important because every object you make (vectors, matrices, tensors) eventually stores scalars. And when you create a NumPy array, you can specify the type - but **every item in the array must have the same type**. In this regard, NumPy arrays are more like C arrays than Python lists.

If you want to create a NumPy array that holds a scalar, you do so by passing the value to NumPy's array function, like so:

s = np.array(5)

You can still perform math between ndarrays, NumPy scalars, and normal Python scalars, though, as you'll see in the element-wise math lesson.

You can see the shape of your arrays by checking their shape attribute. So if you executed this code:

s.shape

it would print out the result, an empty pair of parenthesis, (). This indicates that it has zero dimensions.

Even though scalars are inside arrays, you still use them like a normal scalar. So you could type:

x = s + 3

and x would now equal 8. If you were to check the type of x, you'd find it is probably numpy.int64, because its working with NumPy types, not Python types.

By the way, even scalar types support most of the array functions. so you can call x.shape and it would return () because it has zero dimensions, even though it is not an array. If you tried that with a normal Python scalar, you'd get an error.

**Vectors**

To create a vector, you'd pass a Python list to the array function, like this:

v = np.array([1,2,3])

If you check a vector's shape attribute, it will return a single number representing the vector's one-dimensional length. In the above example, v.shape would return (3,)

Now that there is a number, you can see that the shape is a tuple with the sizes of each of the ndarray's dimensions. For scalars it was just an empty tuple, but vectors have one dimension, so the tuple includes a number and a comma. (Python doesn’t understand (3) as a tuple with one item, so it requires the comma. You can read more about tuples [here](https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences))

You can access an element within the vector using indices, like this:

x = v[1]

Now x equals 2.

NumPy also supports advanced indexing techniques. For example, to access the items from the second element onward, you would say:

v[1:]

and it would return an array of [2, 3]. NumPy slicing is quite powerful, allowing you to access any combination of items in an ndarray. But it can also be a bit complicated, so you should read up on it [in the documentation](https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html).

**Matrices**

You create matrices using NumPy's array function, just you did for vectors. However, instead of just passing in a list, you need to supply a list of lists, where each list represents a row. So to create a 3x3 matrix containing the numbers one through nine, you could do this:

m = np.array([[1,2,3], [4,5,6], [7,8,9]])

Checking its shape attribute would return the tuple (3, 3) to indicate it has two dimensions, each length 3.

You can access elements of matrices just like vectors, but using additional index values. So to find the number 6 in the above matrix, you'd access m[1][2].

**Tensors**

Tensors are just like vectors and matrices, but they can have more dimensions. For example, to create a 3x3x2x1 tensor, you could do the following:

t = np.array([[[[1],[2]],[[3],[4]],[[5],[6]]],[[[7],[8]],\

[[9],[10]],[[11],[12]]],[[[13],[14]],[[15],[16]],[[17],[17]]]])

And t.shape would return (3, 3, 2, 1).

You can access items just like with matrices, but with more indices. So t[2][1][1][0] will return 16.

**Changing Shapes**

Sometimes you'll need to change the shape of your data without actually changing its contents. For example, you may have a vector, which is one-dimensional, but need a matrix, which is two-dimensional. There are two ways you can do that.

Let's say you have the following vector:

v = np.array([1,2,3,4])

Calling v.shape would return (4,). But what if you want a 1x4 matrix? You can accomplish that with the [reshape](https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html) function, like so:

x = v.reshape(1,4)

Calling x.shape would return (1,4). If you wanted a 4x1 matrix, you could do this:

x = v.reshape(4,1)

The reshape function works for more than just adding a dimension of size 1. Check out its [documentation](https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html) for more examples.

One more thing about reshaping NumPy arrays: if you see code from experienced NumPy users, you will often see them use a special slicing syntax instead of calling reshape. Using this syntax, the previous two examples would look like this:

x = v[None, :]

or

x = v[:, None]

Those lines create a slice that looks at all of the items of v but asks NumPy to add a new dimension of size 1 for the associated axis. It may look strange to you now, but it's a common technique so it's good to be aware of it.

NEXT

* + 1. Element wise matrix operations

Element wise operations can be easily carries out in matrices like the scalar values.

* + 1. Element wise operations in numpy

# Element-wise operations

### The Python way

Suppose you had a list of numbers, and you wanted to add 5 to every item in the list. Without NumPy, you might do something like this:

**values** = [1,2,3,4,5]

**for** i in range(len(**values**)):

**values**[i] += 5

*# now values holds [6,7,8,9,10]*

That makes sense, but it's a lot of code to write and it runs slowly because it's pure Python.

**Note:** Just in case you aren't used to using operators like +=, that just means "add these two items and then store the result in the left item." It is a more succinct way of writing values[i] = values[i] + 5. The code you see in these examples makes use of such operators whenever possible.

### The NumPy way

In NumPy, we could do the following:

values = [1,2,3,4,5]

values = np.array(values) + 5

**# now values is an ndarray that holds [6,7,8,9,10]**

Creating that array may seem odd, but normally you'll be storing your data in ndarrays anyway. So if you already had an ndarray named values, you could have just done:

values += 5

We should point out, NumPy actually has functions for things like adding, multiplying, etc. But it also supports using the standard math operators. So the following two lines are equivalent:

x = np.multiply(some\_array, 5)

x = some\_array \* 5

We will usually use the operators instead of the functions because they are more convenient to type and easier to read, but it's really just personal preference.

One more example of operating with scalars and ndarrays. Let's say you have a matrix m and you want to reuse it, but first you need to set all its values to zero. Easy, just multiply by zero and assign the result back to the matrix, like this:

m \*= 0

**# now every element in m is zero, no matter how many dimensions it has**

### Element-wise Matrix Operations

The same functions and operators that work with scalars and matrices also work with other dimensions. You just need to make sure that the items you perform the operation on have compatible shapes.

Let's say you want to get the squared values of a matrix. That's simply x = m \* m (or if you want to assign the value back to m, it's just m \*= m

This works because it's an element-wise multiplication between two identically-shaped matrices. (In this case, they are shaped the same because they are actually the same object.)

Here's the example from the video:

a = np.array([[1,3],[5,7]])

a

**# displays the following result:**

**# array([[1, 3],**

**# [5, 7]])**

b = np.array([[2,4],[6,8]])

b

**# displays the following result:**

**# array([[2, 4],**

**# [6, 8]])**

a + b

**# displays the following result**

**# array([[ 3, 7],**

**# [11, 15]])**

And if you try working with incompatible shapes, like the other example from the video, you'd get an error:

a = np.array([[1,3],[5,7]])

a

**# displays the following result:**

**# array([[1, 3],**

**# [5, 7]])**

c = np.array([[2,3,6],[4,5,9],[1,8,7]])

c

**# displays the following result:**

**# array([[2, 3, 6],**

**# [4, 5, 9],**

**# [1, 8, 7]])**

a.shape

**# displays the following result:**

**# (2, 2)**

c.shape

**# displays the following result:**

**# (3, 3)**

a + c

**# displays the following error:**

**# ValueError: operands could not be broadcast together with shapes (2,2) (3,3)**

You'll learn more about what that "could not be broadcast together" means in a later lesson, but for now, just notice that the two shapes are different so we can't perform the element-wise operation.

NEXT

* + 1. Matrix multiplication part 1

Matrix multiplication is the multiplication of each item in first with second matrices. For this two matrix should be of equal size.

But matrix dot product shouldn’t need the matrix of same sizes. It will just take the first row in the first matrix and multiply that with the first column in second matrix and it will proceed until it find all the calculation.

**Important Reminders About Matrix Multiplication**

* The **number** of **columns** in the **left** matrix **must equal** the **number** of **rows** in the **right** matrix.
* The **answer** matrix **always has** the **same number** of **rows** as the **left** matrix and the **same number** of **columns** as the **right** matrix.
* **Order matters**. Multiplying **A•B** is **not the same** as multiplying **B•A**.
* Data in the **left** matrix **should be** arranged as **rows**., while data in the **right** matrix **should be** arranged as **columns**.

If you keep these four points in mind, you should always be able to figure out how to properly arrange your matrix multiplications when building a neural network.

* + 1. Numpy Matrix multiplication

You've heard a lot about matrix multiplication in the last few videos – now you'll get to see how to do it with NumPy. However, it's important to know that NumPy supports several types of matrix multiplication.

**Element-wise Multiplication**

You saw some element-wise multiplication already. You accomplish that with the multiply function or the \* operator. Just to revisit, it would look like this:

* m = np.array([[1,2,3],[4,5,6]])
* m
* **# displays the following result:**
* **# array([[1, 2, 3],**
* **# [4, 5, 6]])**
* n = m \* 0.25
* n
* **# displays the following result:**
* **# array([[ 0.25, 0.5 , 0.75],**
* **# [ 1. , 1.25, 1.5 ]])**
* m \* n
* **# displays the following result:**
* **# array([[ 0.25, 1. , 2.25],**
* **# [ 4. , 6.25, 9. ]])**
* np.multiply(m, n) **# equivalent to m \* n**
* **# displays the following result:**
* **# array([[ 0.25, 1. , 2.25],**
* **# [ 4. , 6.25, 9. ]])**
* **Matrix Product**
* To find the matrix product, you use NumPy's matmul function.
* If you have compatible shapes, then it's as simple as this:
* a = np.array([[1,2,3,4],[5,6,7,8]])
* a
* **# displays the following result:**
* **# array([[1, 2, 3, 4],**
* **# [5, 6, 7, 8]])**
* a.shape
* **# displays the following result:**
* **# (2, 4)**
* b = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
* b
* **# displays the following result:**
* **# array([[ 1, 2, 3],**
* **# [ 4, 5, 6],**
* **# [ 7, 8, 9],**
* **# [10, 11, 12]])**
* b.shape
* **# displays the following result:**
* **# (4, 3)**
* c = np.matmul(a, b)
* c
* **# displays the following result:**
* **# array([[ 70, 80, 90],**
* **# [158, 184, 210]])**
* c.shape
* **# displays the following result:**
* **# (2, 3)**
* If your matrices have incompatible shapes, you'll get an error, like the following:
* np.matmul(b, a)
* **# displays the following error:**
* **# ValueError: shapes (4,3) and (2,4) not aligned: 3 (dim 1) != 2 (dim 0)**
* **NumPy's dot function**
* You may sometimes see NumPy's [dot](https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html) function in places where you would expect a matmul. It turns out that the results of dot and matmul are the same *if the matrices are two dimensional*.
* So these two results are equivalent:
* a = np.array([[1,2],[3,4]])
* a
* **# displays the following result:**
* **# array([[1, 2],**
* **# [3, 4]])**
* np.dot(a,a)
* **# displays the following result:**
* **# array([[ 7, 10],**
* **# [15, 22]])**
* a.dot(a) **# you can call `dot` directly on the `ndarray`**
* **# displays the following result:**
* **# array([[ 7, 10],**
* **# [15, 22]])**
* np.matmul(a,a)
* **# array([[ 7, 10],**
* **# [15, 22]])**
* While these functions return the same results for two dimensional data, you should be careful about which you choose when working with other data shapes. You can read more about the differences, and find links to other NumPy functions, in the [matmul](https://docs.scipy.org/doc/numpy/reference/generated/numpy.matmul.html" \l "numpy.matmul" \t "_blank) and [dot](https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html) documentation.
  + 1. Matrix transposes

A matrix with the same values as the original but it has the rows and columns switched.

You can safely use a transpose in a matrix multiplication if te data in both of your original matrices is arranged as rows.

* + 1. Transposes in numpy
    - **Transpose**
    - Getting the transpose of a matrix is really easy in NumPy. Simply access its T attribute. There is also a transpose() function which returns the same thing, but you’ll rarely see that used anywhere because typing T is so much easier. :)
    - For example:
    - m = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
    - m
    - **# displays the following result:**
    - **# array([[ 1, 2, 3, 4],**
    - **# [ 5, 6, 7, 8],**
    - **# [ 9, 10, 11, 12]])**
    - m.T
    - **# displays the following result:**
    - **# array([[ 1, 5, 9],**
    - **# [ 2, 6, 10],**
    - **# [ 3, 7, 11],**
    - **# [ 4, 8, 12]])**
    - NumPy does this without actually moving any data in memory - it simply changes the way it indexes the original matrix - so it’s quite efficient.
    - However, that also means you need to be careful with how you modify objects, because **they are sharing the same data**. For example, with the same matrix m from above, let's make a new variable m\_t that stores m's transpose. Then look what happens if we modify a value in m\_t:
    - **m\_t** = m.T
    - **m\_t**[3][1] = 200
    - **m\_t**
    - **# displays the following result:**
    - **# array([[ 1, 5, 9],**
    - **# [ 2, 6, 10],**
    - **# [ 3, 7, 11],**
    - **# [ 4, 200, 12]])**
    - m
    - **# displays the following result:**
    - **# array([[ 1, 2, 3, 4],**
    - **# [ 5, 6, 7, 200],**
    - **# [ 9, 10, 11, 12]])**
    - Notice how it modified both the transpose and the original matrix, too! That's because they are sharing the same copy of data. So remember to consider the transpose just as a different view of your matrix, rather than a different matrix entirely.
    - **A real use case**
    - I don't want to get into too many details about neural networks because you haven't covered them yet, but there is one place you will almost certainly end up using a transpose, or at least thinking about it.
    - Let's say you have the following two matrices, called inputs and weights,
    - inputs = np.array([[-0.27, 0.45, 0.64, 0.31]])
    - inputs
    - **# displays the following result:**
    - **# array([[-0.27, 0.45, 0.64, 0.31]])**
    - inputs.shape
    - **# displays the following result:**
    - **# (1, 4)**
    - weights = np.array([[0.02, 0.001, -0.03, 0.036], \
    - [0.04, -0.003, 0.025, 0.009], [0.012, -0.045, 0.28, -0.067]])
    - weights
    - **# displays the following result:**
    - **# array([[ 0.02 , 0.001, -0.03 , 0.036],**
    - **# [ 0.04 , -0.003, 0.025, 0.009],**
    - **# [ 0.012, -0.045, 0.28 , -0.067]])**
    - weights.shape
    - **# displays the following result:**
    - **# (3, 4)**
    - I won't go into what they're for because you'll learn about them later, but you're going to end up wanting to find the **matrix product** of these two matrices.
    - If you try it like they are now, you get an error:
    - np.matmul(inputs, weights)
    - **# displays the following error:**
    - **# ValueError: shapes (1,4) and (3,4) not aligned: 4 (dim 1) != 3 (dim 0)**
    - If you did the matrix multiplication lesson, then you've seen this error before. It's complaining of incompatible shapes because the number of columns in the left matrix, 4, does not equal the number of rows in the right matrix, 3.
    - So that doesn't work, but notice if you take the transpose of the weights matrix, it will:
    - np.matmul(inputs, weights.T)
    - **# displays the following result:**
    - **# array([[-0.01299, 0.00664, 0.13494]])**
    - It also works if you take the transpose of inputs instead and swap their order, like we showed in the video:
    - np.matmul(weights, inputs.T)
    - **# displays the following result:**
    - **# array([[-0.01299],#**
    - **# [ 0.00664],**
    - **# [ 0.13494]])**
    - The two answers are transposes of each other, so which multiplication you use really just depends on the shape you want for the output.

1. Neural networks:
   1. Introduction to neural networks
   2. Deep learning has been almost everywhere to predict the patterns in the data
2. Classification Problems 1
   1. For example, the student needs to be identified for the acceptance to the university based on the scores in class test and grades. The best possible line will separate the students with low mark from the high mark.
   2. How to find the best possible line is the question
3. Linear boundaries
   1. The line which separates the data has the linear equation. Like.
      * 2x1 + x2 – 18 = 0
      * Predictions
        + If the result is positive – then the student will be accepted
        + Negative – then the student will be rejected
   2. Most of the time the line will be represented as,
      * w1x1 + w2x2 + b = 0
      * Wx + b = 0
      * Where W is weights
      * X – is inputs
      * B is bias
      * Y = label : 0 or 1
      * Prediction : y hat = 1 if wx+b >= 0 and 0 if Wx+b < 0
4. Higher dimensions
   1. How we will be working for data more than 2 columns? For example, if we have data like test, grades and class rank., then we need a plane to separate the points.
   2. Equation for plane in 3 dimensions
      * W1x1+w2x2+w3x3+b = 0
   3. Boundary in n-1 dimensional hyperplane
      * w1x1+w2x2+wnxn+b = 0
      * wx +b = 0
5. Perceptron’s:
   1. It is a building block of neural networks and its just an encoding of our equation into a small graph.
   2. Weights define the linear equation. Bias can be considered as part of nodes or part of the inptus.
   3. Nodes with n inputs and n weights and the output.
   4. Step function: it will help to return 0 or 1. We have different kind of step function.
6. Why Neural networks
   1. Perceptron looks like neurons in the brain,
7. Perceptron as Logical operators
   1. AND, OR, XOR can be turned into perceptron. It differs only by inputs, weights and bias
8. Perceptron trick
   1. Find the rightly classified and misclassified items from the classification and change the line in such a way that it is closer to the points so that it will be classified properly.
9. Perceptron algorithm
   1. Start with random weights: w1, ….wn, b
   2. For every misclassified point (x1,…xn)
      * If prediction = 0
        + For I = 1…n
          - Change wi + alphaXi
        + Change b to b+alpha
      * If prediction =1:
        + For I = 1…n
          - Change wi - alphaXi
        + Change b to b - alpha
10. Non-Linear regions
    1. Perceptron algorithm sometime will not work for the non-linear dataset where we need the curve to separate the two classes.
11. Error functions
    1. Helps to identify the best possible solutions by taking smaller steps.
12. Log-Loss error function
    1. Discrete Vs Continouus
       * We need to minimize the error, by small steps. In order to do that, gradient descent our error function cannot be discrete it should be continuous. Our error function needs to be differentiable. Our error function needs to be continuous.
13. Discrete vs Continuous
    1. To move from the discrete to continuous, we just need to apply the functions.
       * **Step:** functions will give the discrete output like 0 or 1. It will give whether the student will be accepted or not.
       * **Sigmoid:** Continuous function will provide the output in probability. It will provide the output to show the probability that the student will be accepted or not.
14. Multiclass classification and Softmax
    1. If the output class has more than one class then it will be called as multiclass classification.
       * P(gift) = 0.8, P(no gift) = 0.2
       * Score(gift) = Linear function
       * P(gift) = sigmoid(Score)
    2. Multiclass classification:
       * Consider we have the problem to identify the type of animals we probably see
         + P(duck) = 0.67 score = 2 | P(beaver) = 0.24 score = 1| P(walrus) = 0.09 score = 0
       * How to convert the score to the probabilities?
         + P(duck) = 2/ 2+1+0 --- this may work but it will not work if the number is negative, so we need to convert the negative numbers to positive. Exponential function will always returns positive values.
    3. Softmax:
       * smooth approximation to the [arg max](https://en.wikipedia.org/wiki/Arg_max" \o "Arg max) function
15. One-Hot encoding
    1. Model will always accept inputs as numeric values. So, we will come up with one variable for each of the class in the features. It will result in multiple columns but there will not be dependency.
16. Maximum Likelihood
    1. The models gives the higher probabilities to the events that happened to us whether it is acceptance or rejection. We pick the model that gives the existing labels the highest probability. Maximize the probability of being correct.
    2. Identify the probability of each predictions and then multiply all the values. The model for which the value is high is the model with maximum likelihood.
       * Probability of all elements in probability space:
         + P(all) = 0.6 \* 0.2\* 0.1\* 0.7 = 0.0084
         + P(all) = 0.7\*0.9\*0.8\*0.6 = 0.3024
       * We will pick the model 2 which has the maximum likelihood.
17. Maximizing probabilities
    1. Maximizing the probability in turn wil minimize the error functions.
    2. Getting the product of all the probability will be sometime huge process, instead sum will be better. Logarithm function has the option to convert products into sums.
       * Log(ab) = log a + log b
18. Cross-Entropy
    1. Logarithm of any value below 1 will be negative. We will take the negative of the logarithm of the probabilities to get the positive values. The sums up negatives of logarithms of the probabilities is called cross entropy.
    2. Good model gives low entropy and bad model gives high entropy.
    3. There is a connection between probabilities and error functions and its called cross-entropy.
    4. if we have bunch of events and a bunch of probabilities, how likely is it those events happen based on the probabilities,
       * Low cross entropy – if it’s very likely
       * High cross entropy – if it’s unlikely
    5. Formula for cross entropy:
       * Cross-Entropy = - summation yi ln(pi) + (1-yi) ln(1-pi)
19. Multi-Class cross entropy
    1. If the problem is multi class where the possible output may be more than 2 then we need a different formula
    2. Example for multi-class cross entropy:

Finding the animal behind the door

|  |  |  |  |
| --- | --- | --- | --- |
| Animal | Door1 | Door 2 | Door 3 |
| Duck | 0.7 | 0.3 | 0.1 |
| Beaver | 0.2 | 0.4 | 0.5 |
| Walrus | 0.1 | 0.3 | 0.4 |

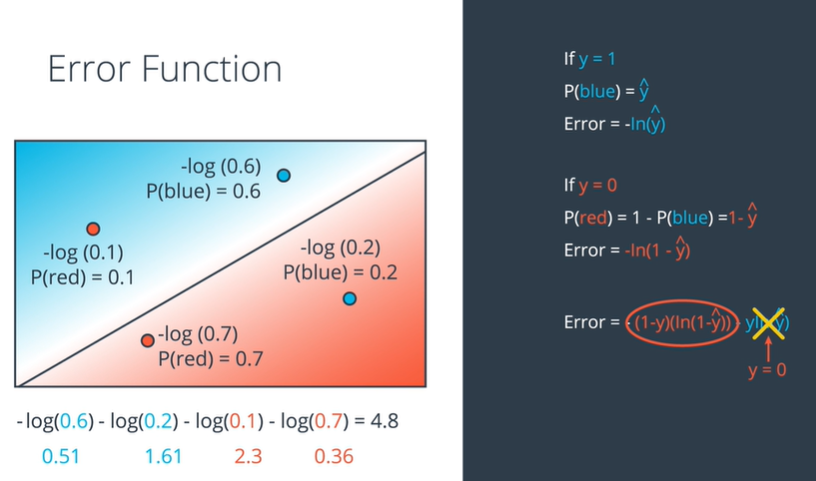
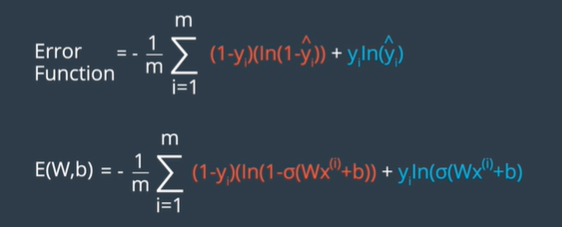
Consider the scenario, where Duck behind door1, Walrus behind 2, Walrus behind 3

P = 0.7 \* 0.3 \* 0.4 = 0.084

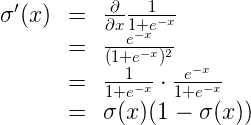
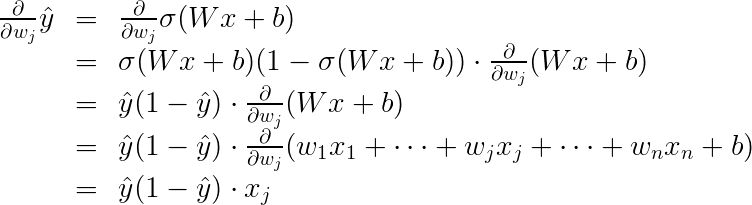
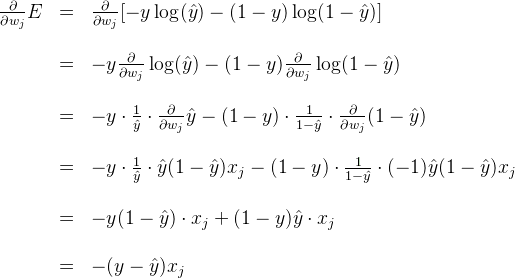
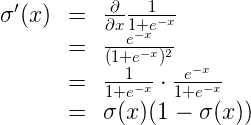
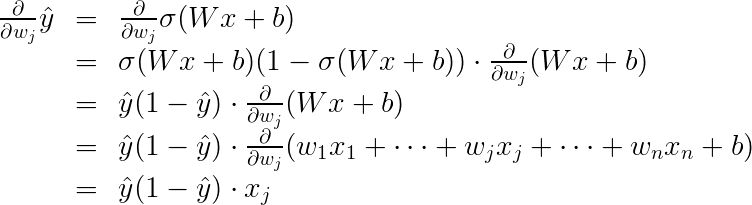
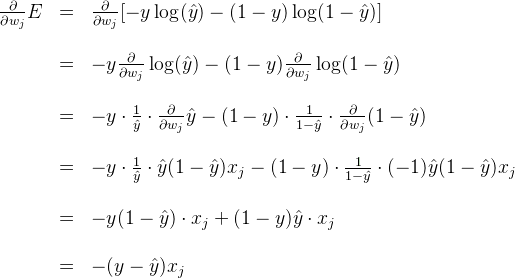
CE = -ln(0.7) + -ln(0.3) + -ln(0.4) = 2.48

y1j = 1 | y2j = 1 | y3j = 1

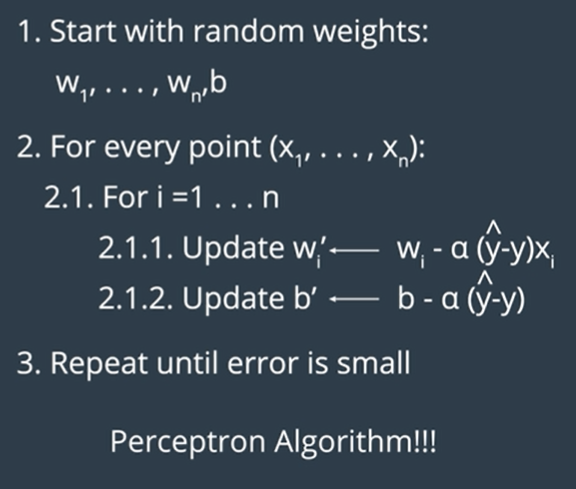
cross-entropy = - summation of i = 1 to n summarion j =1 to m (Yij In (Pij)) where m is number of classes

1. Error function
   1. 
   2. 
   3. Minimize the error function:
2. Gradient descent



* 1. Gradient – it is differential of error with respect to weights, this will help us to move towards the global minima.
  2. To reach the minima we should get the negative delta E
* **Gradient Calculation**
* In the last few videos, we learned that in order to minimize the error function, we need to take some derivatives. So let's get our hands dirty and actually compute the derivative of the error function. The first thing to notice is that the sigmoid function has a really nice derivative. Namely,
* \sigma'(x) = \sigma(x) (1-\sigma(x))*σ*′(*x*)=*σ*(*x*)(1−*σ*(*x*))
* The reason for this is the following, we can calculate it using the quotient formula:
* 
* And now, let's recall that if we have m*m* points labelled x^{(1)}, x^{(2)}, \ldots, x^{(m)},*x*(1),*x*(2),…,*x*(*m*), the error formula is:
* E = -\frac{1}{m} \sum\_{i=1}^m \left( y\_i \ln(\hat{y\_i}) + (1-y\_i) \ln (1-\hat{y\_i}) \right)*E*=−*m*1​∑*i*=1*m*​(*yi*​ln(*yi*​^​)+(1−*yi*​)ln(1−*yi*​^​))
* where the prediction is given by \hat{y\_i} = \sigma(Wx^{(i)} + b).*yi*​^​=*σ*(*Wx*(*i*)+*b*).
* Our goal is to calculate the gradient of E,*E*, at a point x = (x\_1, \ldots, x\_n),*x*=(*x*1​,…,*xn*​), given by the partial derivatives
* \nabla E =\left(\frac{\partial}{\partial w\_1}E, \cdots, \frac{\partial}{\partial w\_n}E, \frac{\partial}{\partial b}E \right)∇*E*=(∂*w*1​∂​*E*,⋯,∂*wn*​∂​*E*,∂*b*∂​*E*)
* To simplify our calculations, we'll actually think of the error that each point produces, and calculate the derivative of this error. The total error, then, is the average of the errors at all the points. The error produced by each point is, simply,
* E = - y \ln(\hat{y}) - (1-y) \ln (1-\hat{y})*E*=−*y*ln(*y*^​)−(1−*y*)ln(1−*y*^​)
* In order to calculate the derivative of this error with respect to the weights, we'll first calculate \frac{\partial}{\partial w\_j} \hat{y}.∂*wj*​∂​*y*^​. Recall that \hat{y} = \sigma(Wx+b),*y*^​=*σ*(*Wx*+*b*), so:
* 
* The last equality is because the only term in the sum which is not a constant with respect to w\_j*wj*​ is precisely w\_j x\_j,*wj*​*xj*​, which clearly has derivative x\_j.*xj*​.
* Now, we can go ahead and calculate the derivative of the error E*E* at a point x,*x*, with respect to the weight w\_j.*wj*​.
* 
* A similar calculation will show us that
* 
* This actually tells us something very important. For a point with coordinates (x\_1, \ldots, x\_n),(*x*1​,…,*xn*​), label y,*y*, and prediction \hat{y},*y*^​, the gradient of the error function at that point is \left(-(y - \hat{y})x\_1, \cdots, -(y - \hat{y})x\_n, -(y - \hat{y}) \right).(−(*y*−*y*^​)*x*1​,⋯,−(*y*−*y*^​)*xn*​,−(*y*−*y*^​)). In summary, the gradient is
* \nabla E = -(y - \hat{y}) (x\_1, \ldots, x\_n, 1).∇*E*=−(*y*−*y*^​)(*x*1​,…,*xn*​,1).
* If you think about it, this is fascinating. The gradient is actually a scalar times the coordinates of the point! And what is the scalar? Nothing less than a multiple of the difference between the label and the prediction. What significance does this have?
* **Gradient Calculation**
* In the last few videos, we learned that in order to minimize the error function, we need to take some derivatives. So let's get our hands dirty and actually compute the derivative of the error function. The first thing to notice is that the sigmoid function has a really nice derivative. Namely,
* \sigma'(x) = \sigma(x) (1-\sigma(x))*σ*′(*x*)=*σ*(*x*)(1−*σ*(*x*))
* The reason for this is the following, we can calculate it using the quotient formula:
* 
* And now, let's recall that if we have m*m* points labelled x^{(1)}, x^{(2)}, \ldots, x^{(m)},*x*(1),*x*(2),…,*x*(*m*), the error formula is:
* E = -\frac{1}{m} \sum\_{i=1}^m \left( y\_i \ln(\hat{y\_i}) + (1-y\_i) \ln (1-\hat{y\_i}) \right)*E*=−*m*1​∑*i*=1*m*​(*yi*​ln(*yi*​^​)+(1−*yi*​)ln(1−*yi*​^​))
* where the prediction is given by \hat{y\_i} = \sigma(Wx^{(i)} + b).*yi*​^​=*σ*(*Wx*(*i*)+*b*).
* Our goal is to calculate the gradient of E,*E*, at a point x = (x\_1, \ldots, x\_n),*x*=(*x*1​,…,*xn*​), given by the partial derivatives
* \nabla E =\left(\frac{\partial}{\partial w\_1}E, \cdots, \frac{\partial}{\partial w\_n}E, \frac{\partial}{\partial b}E \right)∇*E*=(∂*w*1​∂​*E*,⋯,∂*wn*​∂​*E*,∂*b*∂​*E*)
* To simplify our calculations, we'll actually think of the error that each point produces, and calculate the derivative of this error. The total error, then, is the average of the errors at all the points. The error produced by each point is, simply,
* E = - y \ln(\hat{y}) - (1-y) \ln (1-\hat{y})*E*=−*y*ln(*y*^​)−(1−*y*)ln(1−*y*^​)
* In order to calculate the derivative of this error with respect to the weights, we'll first calculate \frac{\partial}{\partial w\_j} \hat{y}.∂*wj*​∂​*y*^​. Recall that \hat{y} = \sigma(Wx+b),*y*^​=*σ*(*Wx*+*b*), so:
* 
* The last equality is because the only term in the sum which is not a constant with respect to w\_j*wj*​ is precisely w\_j x\_j,*wj*​*xj*​, which clearly has derivative x\_j.*xj*​.
* Now, we can go ahead and calculate the derivative of the error E*E* at a point x,*x*, with respect to the weight w\_j.*wj*​.
* 
* A similar calculation will show us that
* 
* This actually tells us something very important. For a point with coordinates (x\_1, \ldots, x\_n),(*x*1​,…,*xn*​), label y,*y*, and prediction \hat{y},*y*^​, the gradient of the error function at that point is \left(-(y - \hat{y})x\_1, \cdots, -(y - \hat{y})x\_n, -(y - \hat{y}) \right).(−(*y*−*y*^​)*x*1​,⋯,−(*y*−*y*^​)*xn*​,−(*y*−*y*^​)). In summary, the gradient is
* \nabla E = -(y - \hat{y}) (x\_1, \ldots, x\_n, 1).∇*E*=−(*y*−*y*^​)(*x*1​,…,*xn*​,1).
* If you think about it, this is fascinating. The gradient is actually a scalar times the coordinates of the point! And what is the scalar? Nothing less than a multiple of the difference between the label and the prediction. What significance does this have?

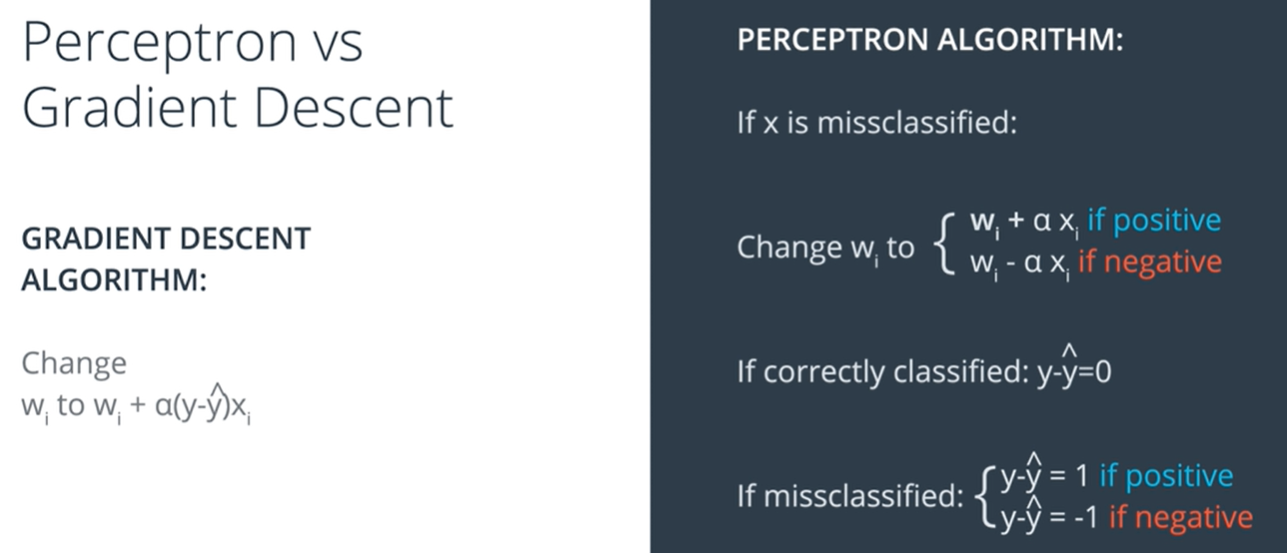
1. Logistic regression algorithm



1. Perceptron vs gradient descent

Perceptron: It will change the weight for only misclassified items. It does nothing if all points are classified correctly

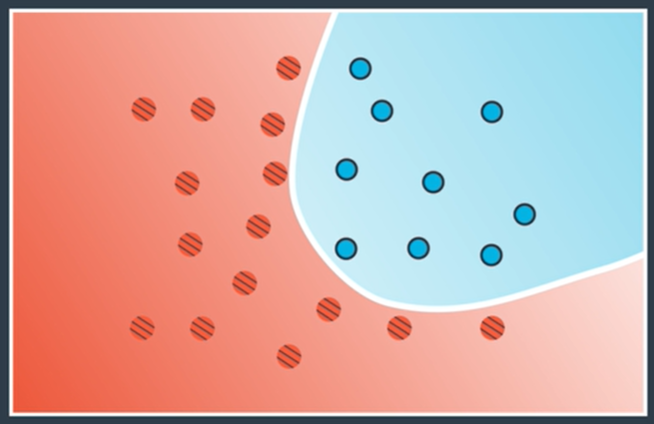
Gradient: It will change the weight for all items. The correctly marked items try to reduce the MSE by adjusting the weights.



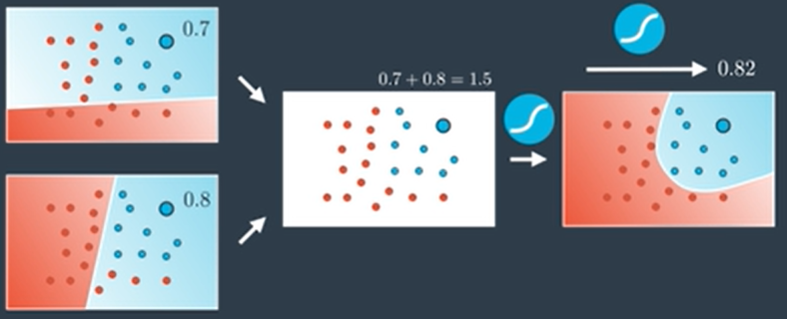
1. Continuous perceptron’s

Linear model will help to split the class. Edges by the weights and the nodes by the bias.

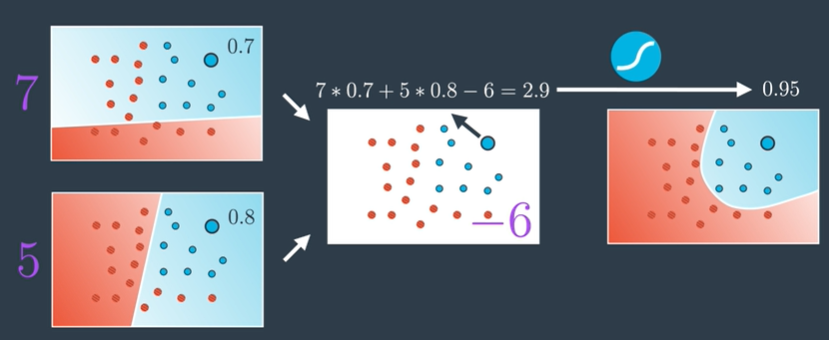
1. Non-linear data and Non-linear models
   1. For non-linear data, it will be tough to separate the data points by using just the line. It may need the curve to separate the points.



1. Neural network architecture
   1. To separate the non-linear points, we may need to combine the two linear problems and sum that up.
   2. Its necessary to apply the sigmoid function, so that the probability value may fall between 0 to 1,

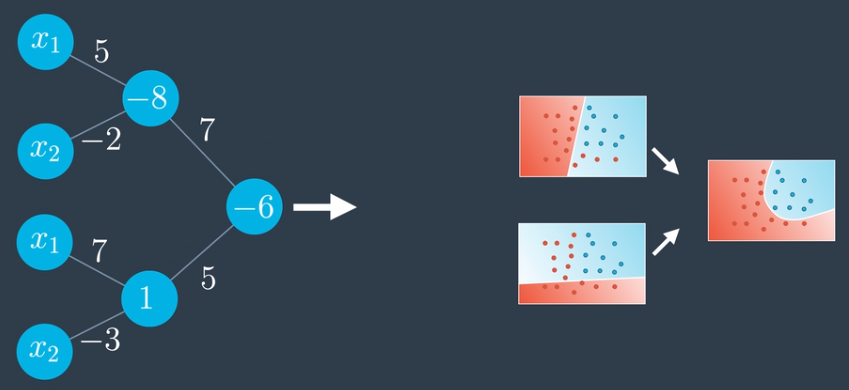


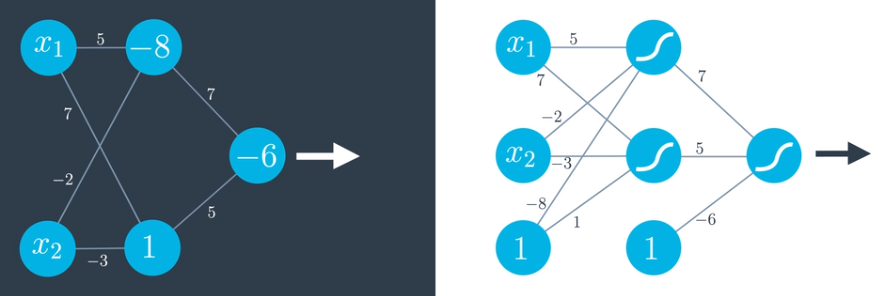
* 1. If we like the first model more than the second one. It’s the linear combination of the two linear models. The below one is the 7 times of the first model and 5 times of second model and some bias.



* 1. Combining the linear models to predict the values will be like below,

7 times first model and 5 times second model with bias -6 will give the output.



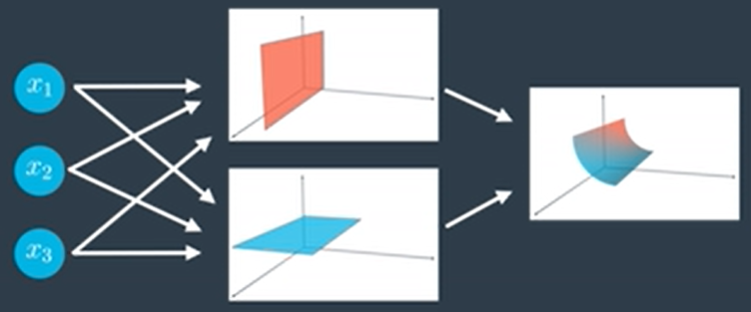


**Deep Neural network:**

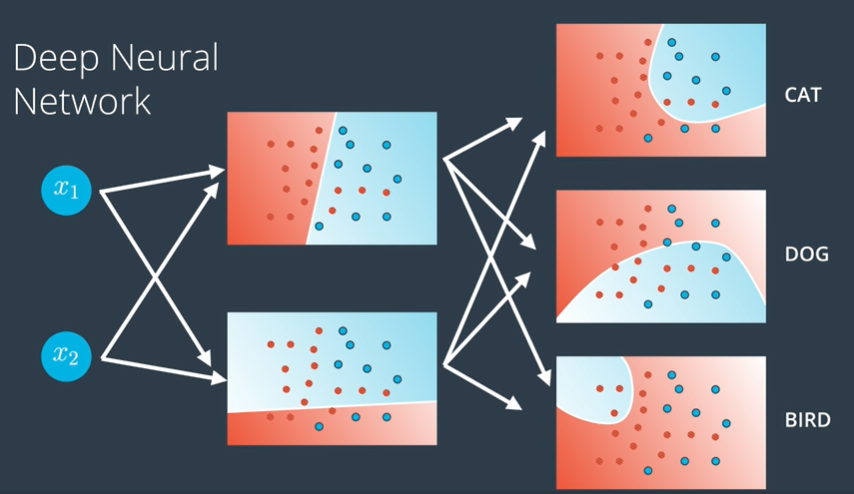
1. **Architecture:**
   1. Input layer – contains inputs. It depends on the number of features, and it decides the dimensionality of the problem.
   2. Hidden layer – set of linear models created with inputs
   3. Output layer – combination of linear models. If output has more then one nodes, it may be the multiclass classification. For example, the model predicts the output as dog, cat and bird will have 3 output nodes.

If the layer is more than we end up with deep neural network

More than one node in inputs

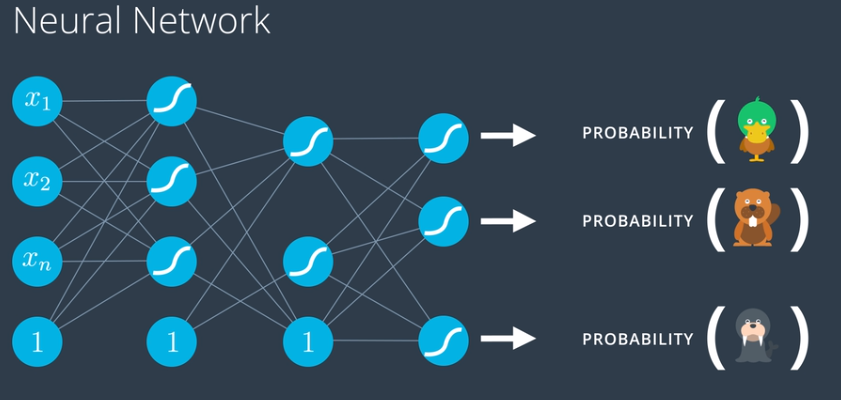


More than one outputs



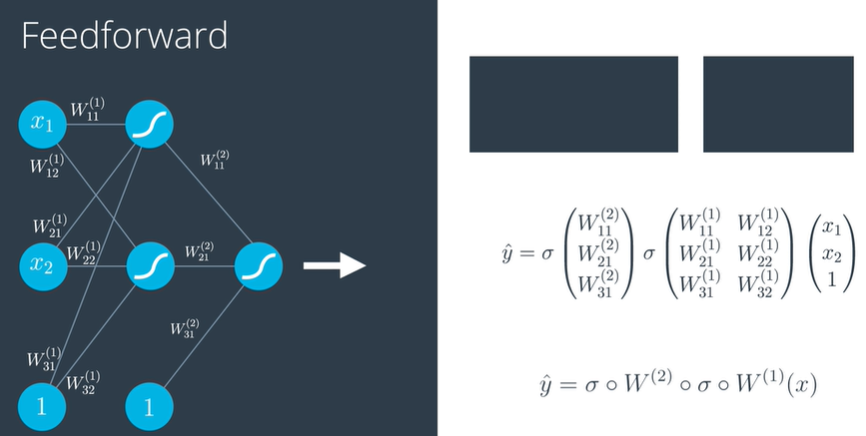
**Multiclass classification:**

1. To predict the multi class classification problem, we will apply the softmax function at the output layer to the maximum probability



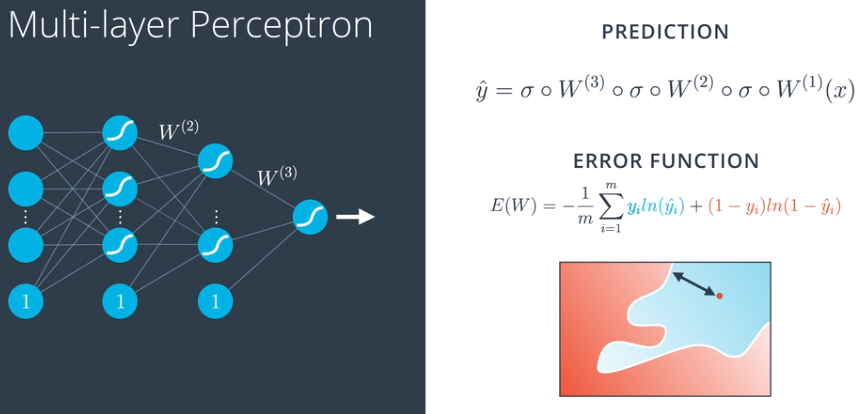
1. Feedforward
   1. **Feedforward**

Feedforward is the process neural networks use to turn the input into an output. Taking the inputs, applying the bias and weights to predict the probability of the output variable is Feed froward.



* 1. **Error function**

Error function for a neural network. It’s the calculation of distance between the misclassified items and the boundary line.

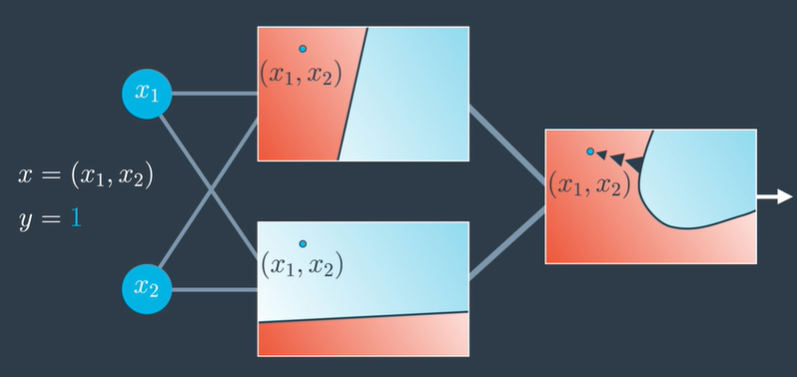


1. Backpropagation
   1. Backpropagation: Now, we're ready to get our hands into training a neural network. For this, we'll use the method known as **backpropagation**. In a nutshell, backpropagation will consist of:

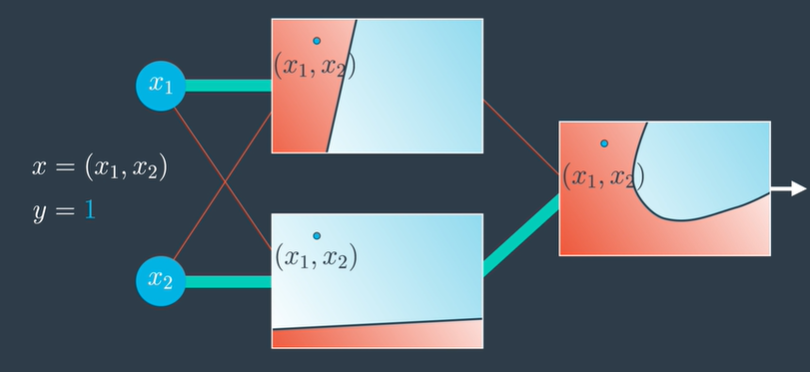
* Doing a feedforward operation.
* Comparing the output of the model with the desired output.
* Calculating the error.
* Running the feedforward operation backwards (backpropagation) to spread the error to each of the weights.
* Use this to update the weights, and get a better model.
* Continue this until we have a model that is good.

Sounds more complicated than what it actually is. Let's take a look in the next few videos. The first video will show us a conceptual interpretation of what backpropagation is

* + - When applying gradient descent, we will calculate the error and move in the negative direction of gradient to get the minimum error.

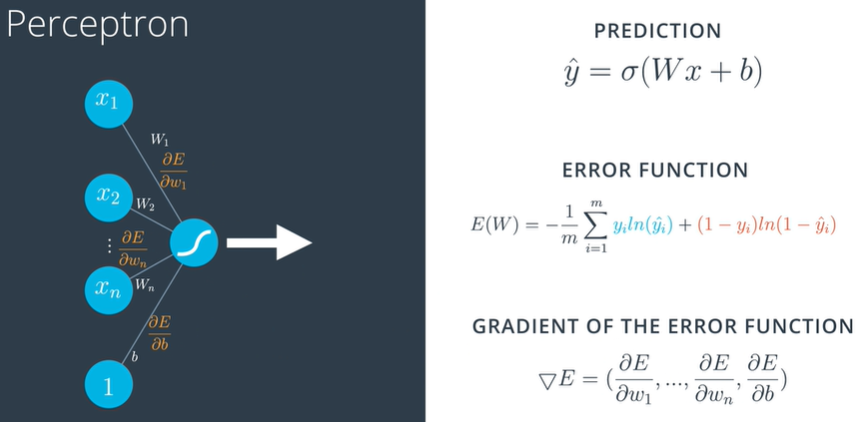


When we want less from model 1 and more from model2, we will update the weights accordingly while backpropagate to minimize the error.

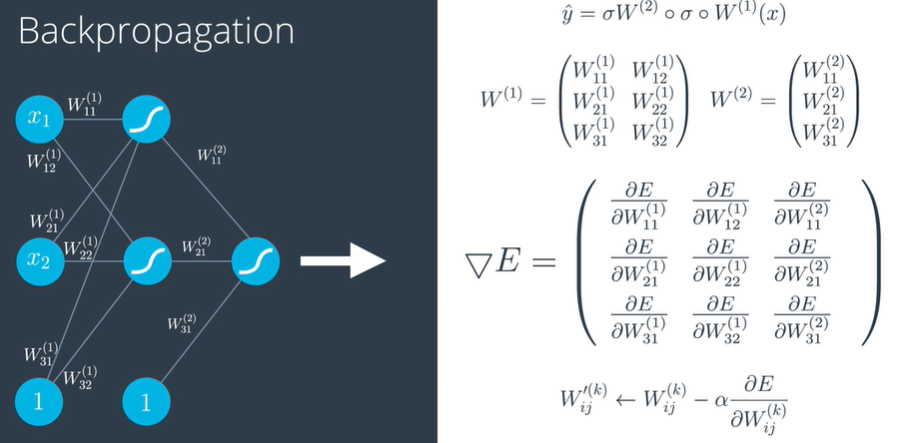


* 1. Backpropagation math
     + And the next few videos will go deeper into the math. Feel free to tune out, since this part gets handled by Keras pretty well. If you'd like to go start training networks right away, go to the next section. But if you enjoy calculating lots of derivatives, let's dive in!
     + In the video below at 1:24, the edges should be directed to the sigmoid function and not the bias at that last layer; the edges of the last layer point to the bias currently which is incorrect.

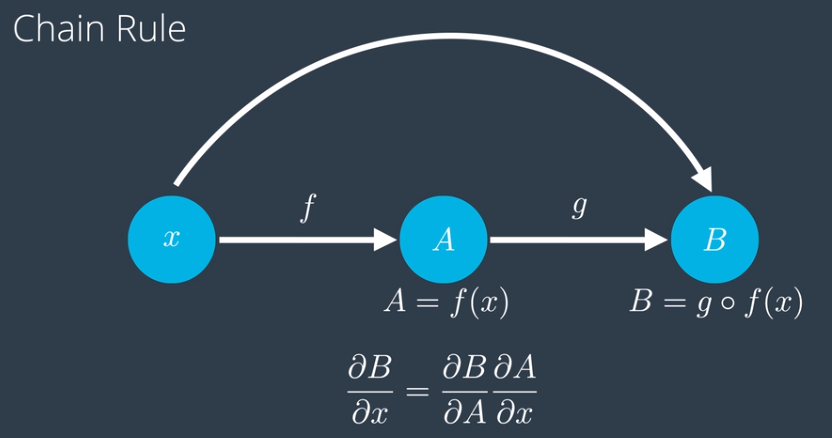
Single perceptron:



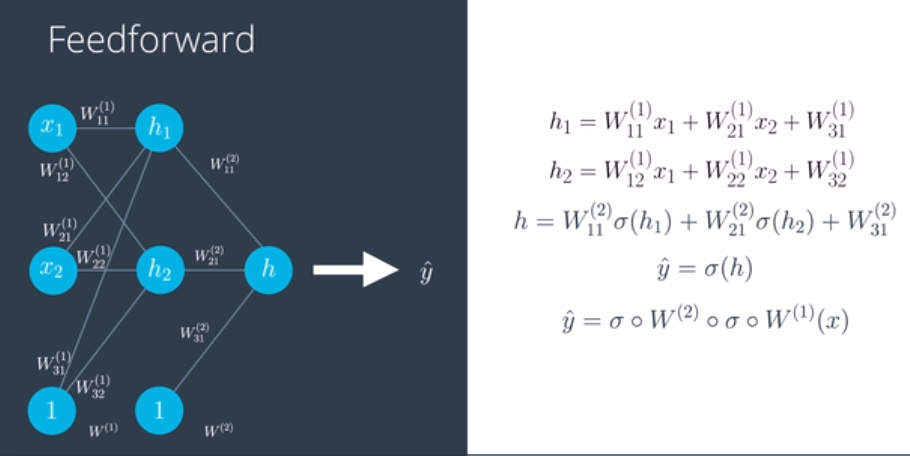
Multi-layer perceptron:



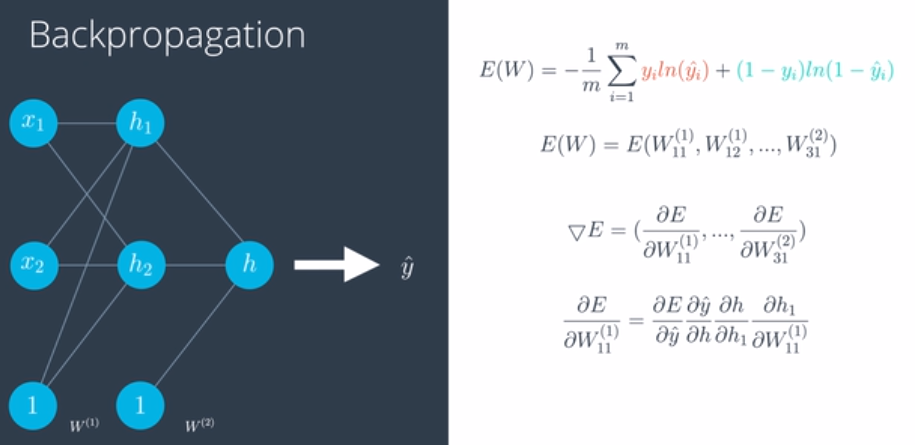
* 1. Chain rule
     + When composing function then derivative is just going to multiply.

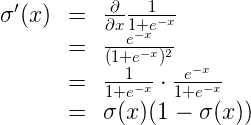


* 1. Feed forward:



* 1. Backpropagation:



* 1. Calculation of the derivative of the sigmoid function
     + Recall that the sigmoid function has a beautiful derivative, which we can see in the following calculation. This will make our backpropagation step much cleaner.
     + 
  2. Implementing gradient descent
     1. Mean squared error function

 this one is the mean of the squares of the differences between the predictions and the labels. In the following section I'll go over it in detail, then we'll get to implement backpropagation with it on the same student admissions dataset.

Why momentum really works:

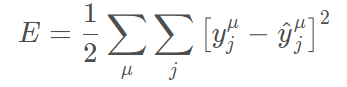
[Why Momentum Really Works (distill.pub)](https://distill.pub/2017/momentum/)

* + 1. Gradient descent
       1. Intro:

# Gradient Descent with Squared Errors

We want to find the weights for our neural networks. Let's start by thinking about the goal. The network needs to make predictions as close as possible to the real values. To measure this, we use a metric of how wrong the predictions are, the **error**. A common metric is the sum of the squared errors (SSE):

E = \frac{1}{2}\sum\_{\mu} \sum\_j \left[ y^{\mu}\_j - \hat{y} ^{\mu}\_j \right]^2*E*=21​*μ*∑​*j*∑​[*yjμ*​−*y*^​*jμ*​]2



where \hat y*y*^​ is the prediction and y*y* is the true value, and you take the sum over all output units j*j* and another sum over all data points \mu*μ*. This might seem like a really complicated equation at first, but it's fairly simple once you understand the symbols and can say what's going on in words.

First, the inside sum over j*j*. This variable j*j* represents the output units of the network. So this inside sum is saying for each output unit, find the difference between the true value y*y* and the predicted value from the network \hat y*y*^​, then square the difference, then sum up all those squares.

Then the other sum over \mu*μ* is a sum over all the data points. So, for each data point you calculate the inner sum of the squared differences for each output unit. Then you sum up those squared differences for each data point. That gives you the overall error for all the output predictions for all the data points.

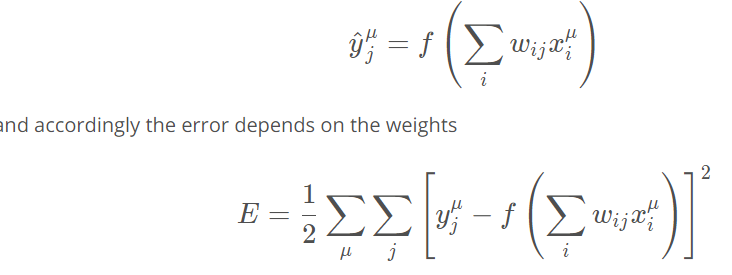
The SSE is a good choice for a few reasons. The square ensures the error is always positive and larger errors are penalized more than smaller errors. Also, it makes the math nice, always a plus.

Remember that the output of a neural network, the prediction, depends on the weights

\hat{y}^{\mu}\_j = f \left( \sum\_i{ w\_{ij} x^{\mu}\_i }\right)*y*^​*jμ*​=*f*(*i*∑​*wij*​*xiμ*​)

and accordingly the error depends on the weights

E = \frac{1}{2}\sum\_{\mu} \sum\_j \left[ y^{\mu}\_j - f \left( \sum\_i{ w\_{ij} x^{\mu}\_i }\right) \right]^2*E*=21​*μ*∑​*j*∑​[*yjμ*​−*f*(*i*∑​*wij*​*xiμ*​)]2

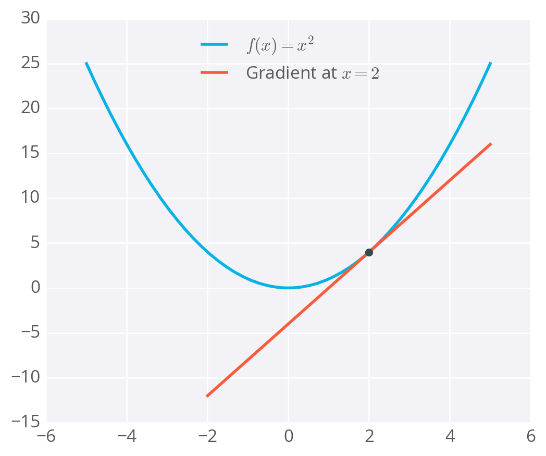


We want the network's prediction error to be as small as possible and the weights are the knobs we can use to make that happen. Our goal is to find weights w\_{ij}*wij*​ that minimize the squared error E*E*. To do this with a neural network, typically you'd use **gradient descent**.

As Luis said, with gradient descent, we take multiple small steps towards our goal. In this case, we want to change the weights in steps that reduce the error. Continuing the analogy, the error is our mountain and we want to get to the bottom. Since the fastest way down a mountain is in the steepest direction, the steps taken should be in the direction that minimizes the error the most. We can find this direction by calculating the gradient of the squared error.

Gradient is another term for rate of change or slope. If you need to brush up on this concept, check out Khan Academy's [great lectures](https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/gradient-and-directional-derivatives/v/gradient) on the topic.

To calculate a rate of change, we turn to calculus, specifically derivatives. A derivative of a function f(x)*f*(*x*) gives you another function f'(x)*f*′(*x*) that returns the slope of f(x)*f*(*x*) at point x*x*. For example, consider f(x)=x^2*f*(*x*)=*x*2. The derivative of x^2*x*2 is f'(x) = 2x*f*′(*x*)=2*x*. So, at x = 2*x*=2, the slope is f'(2) = 4*f*′(2)=4. Plotting this out, it looks like:

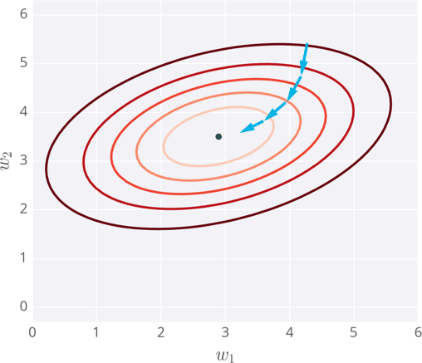


Example of a gradient

The gradient is just a derivative generalized to functions with more than one variable. We can use calculus to find the gradient at any point in our error function, which depends on the input weights. You'll see how the gradient descent step is derived on the next page.

Below I've plotted an example of the error of a neural network with two inputs, and accordingly, two weights. You can read this like a topographical map where points on a contour line have the same error and darker contour lines correspond to larger errors.

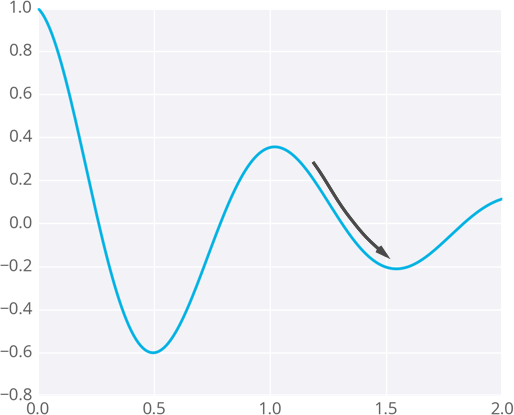
At each step, you calculate the error and the gradient, then use those to determine how much to change each weight. Repeating this process will eventually find weights that are close to the minimum of the error function, the black dot in the middle.



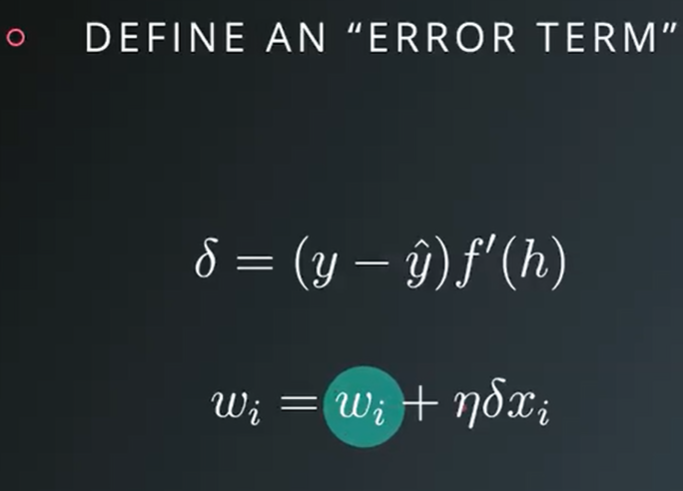
Gradient descent steps to the lowest error

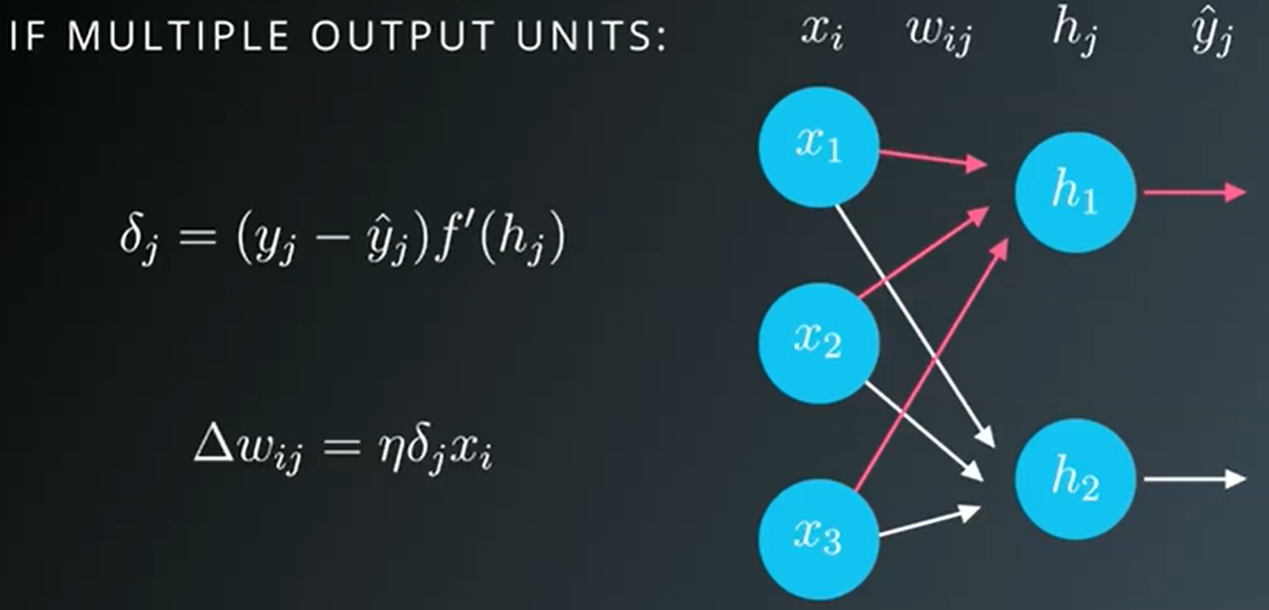
**Caveats**

Since the weights will just go wherever the gradient takes them, they can end up where the error is low, but not the lowest. These spots are called local minima. If the weights are initialized with the wrong values, gradient descent could lead the weights into a local minimum, illustrated below.



* + - 1. The math

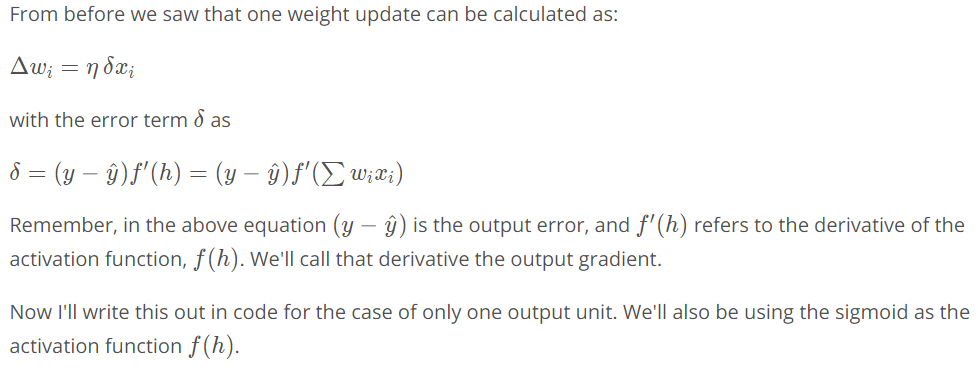




Refer Multi variate calculus:

[Multivariable Calculus | Khan Academy](https://www.khanacademy.org/math/multivariable-calculus)

* + - 1. The code



Okay, now we know how to update our weights:

You've seen how to implement that for a single update, but how do we translate that code to calculate many weight updates so our network will learn?

As an example, I'm going to have you use gradient descent to train a network on graduate school admissions data (found at [**http://www.ats.ucla.edu/stat/data/binary.csv**](https://stats.idre.ucla.edu/stat/data/binary.csv)). This dataset has three input features: GRE score, GPA, and the rank of the undergraduate school (numbered 1 through 4). Institutions with rank 1 have the highest prestige, those with rank 4 have the lowest.



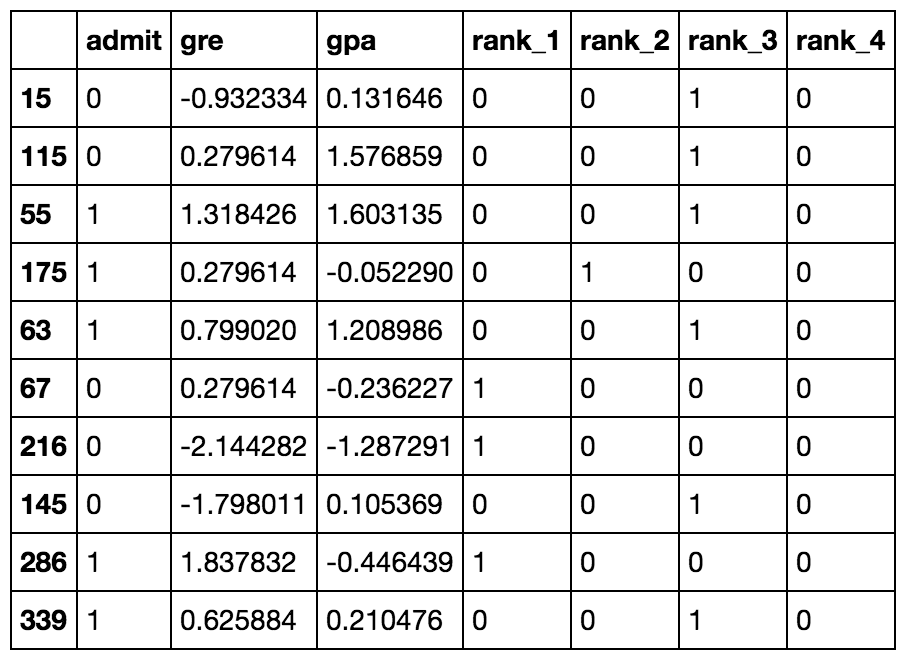
The goal here is to predict if a student will be admitted to a graduate program based on these features. For this, we'll use a network with one output layer with one unit. We'll use a sigmoid function for the output unit activation.

## Data cleanup

You might think there will be three input units, but we actually need to transform the data first. The rank feature is categorical, the numbers don't encode any sort of relative values. Rank 2 is not twice as much as rank 1, rank 3 is not 1.5 more than rank 2. Instead, we need to use [**dummy variables**](https://en.wikipedia.org/wiki/Dummy_variable_(statistics)) to encode rank, splitting the data into four new columns encoded with ones or zeros. Rows with rank 1 have one in the rank 1 dummy column, and zeros in all other columns. Rows with rank 2 have one in the rank 2 dummy column, and zeros in all other columns. And so on.

We'll also need to standardize the GRE and GPA data, which means to scale the values such that they have zero mean and a standard deviation of 1. This is necessary because the sigmoid function squashes really small and really large inputs. The gradient of really small and large inputs is zero, which means that the gradient descent step will go to zero too. Since the GRE and GPA values are fairly large, we have to be really careful about how we initialize the weights or the gradient descent steps will die off and the network won't train. Instead, if we standardize the data, we can initialize the weights easily and everyone is happy.

This is just a brief run-through, you'll learn more about preparing data later. If you're interested in how I did this, check out the data\_prep.py file in the programming exercise below.

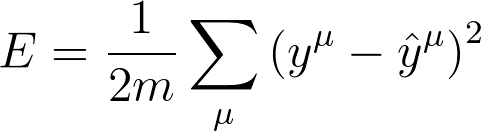


Ten rows of the data after transformations.

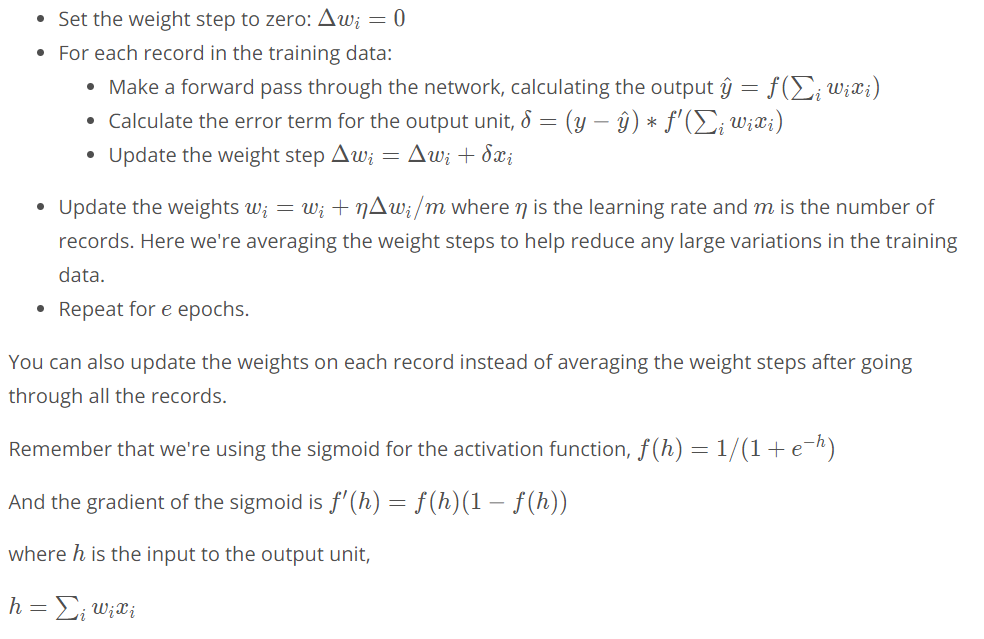
Now that the data is ready, we see that there are six input features: gre, gpa, and the four rank dummy variables.

### Mean Square Error

We're going to make a small change to how we calculate the error here. Instead of the SSE, we're going to use the **mean** of the square errors (MSE). Now that we're using a lot of data, summing up all the weight steps can lead to really large updates that make the gradient descent diverge. To compensate for this, you'd need to use a quite small learning rate. Instead, we can just divide by the number of records in our data, m*m* to take the average. This way, no matter how much data we use, our learning rates will typically be in the range of 0.01 to 0.001. Then, we can use the MSE (shown below) to calculate the gradient and the result is the same as before, just averaged instead of summed.



Here's the general algorithm for updating the weights with gradient descent:



Implementing with NumPy

For the most part, this is pretty straightforward with NumPy.

First, you'll need to initialize the weights. We want these to be small such that the input to the sigmoid is in the linear region near 0 and not squashed at the high and low ends. It's also important to initialize them randomly so that they all have different starting values and diverge, breaking symmetry. So, we'll initialize the weights from a normal distribution centered at 0. A good value for the scale is 1/\sqrt{n}1/*n*​ where n*n* is the number of input units. This keeps the input to the sigmoid low for increasing numbers of input units.

weights = np.random.normal(scale=1/n\_features\*\*.5, size=n\_features)

NumPy provides a function np.dot() that calculates the dot product of two arrays, which conveniently calculates h*h* for us. The dot product multiplies two arrays element-wise, the first element in array 1 is multiplied by the first element in array 2, and so on. Then, each product is summed.

*# input to the output layer*

output\_in = np.dot(weights, inputs)

And finally, we can update \Delta w\_iΔ*wi*​ and w\_i*wi*​ by incrementing them with weights += ... which is shorthand for weights = weights + ....

### Efficiency tip!

You can save some calculations since we're using a sigmoid here. For the sigmoid function, f'(h) = f(h) (1 - f(h))*f*′(*h*)=*f*(*h*)(1−*f*(*h*)). That means that once you calculate f(h)*f*(*h*), the activation of the output unit, you can use it to calculate the gradient for the error gradient.

Programming exercise

Below, you'll implement gradient descent and train the network on the admissions data. Your goal here is to train the network until you reach a minimum in the mean square error (MSE) on the training set. You need to implement:

* The network output: output.
* The output error: error.
* The error term: error\_term.
* Update the weight step: del\_w +=.
* Update the weights: weights +=.

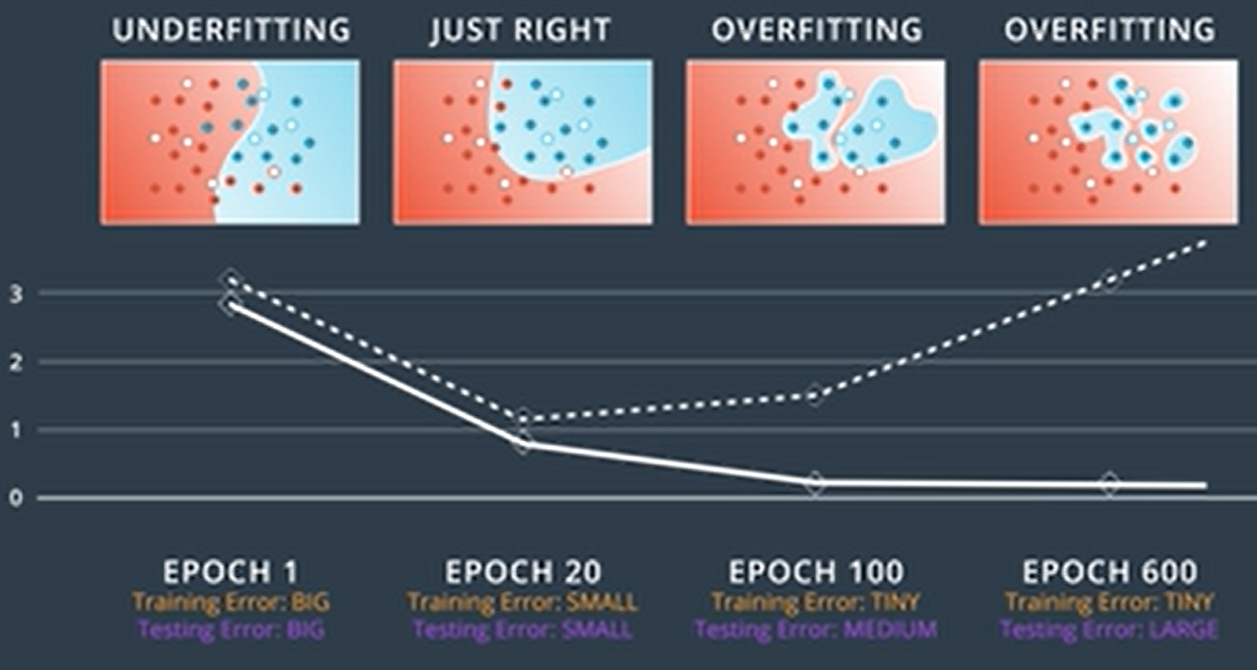
After you've written these parts, run the training by pressing "Test Run". The MSE will print out, as well as the accuracy on a test set, the fraction of correctly predicted admissions.

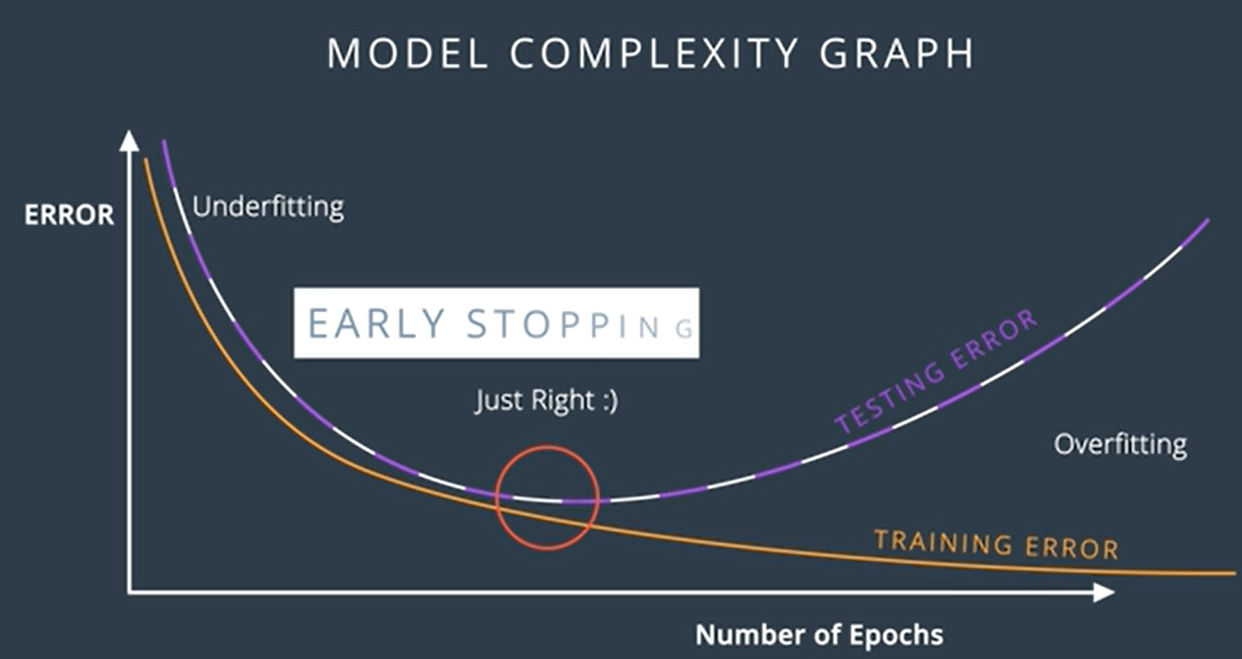
Feel free to play with the hyperparameters and see how it changes the MSE.

* + 1. Multilayer perceptron’s
    2. Backpropagation
    3. Implementing backpropagation
  1. Training neural networks
     1. Training optimization – it is mandatory to check the performance of the data to know how good the model works.
     2. Testing - To test how good the model performs we split the data set into test and train.
     3. Overfitting and underfitting
        1. Underfitting – too simple model, which can’t find the hidden trends in the data
        2. Overfitting – too complex model, which captures noisy data as well

We will learn the options to reduce over and underfitting

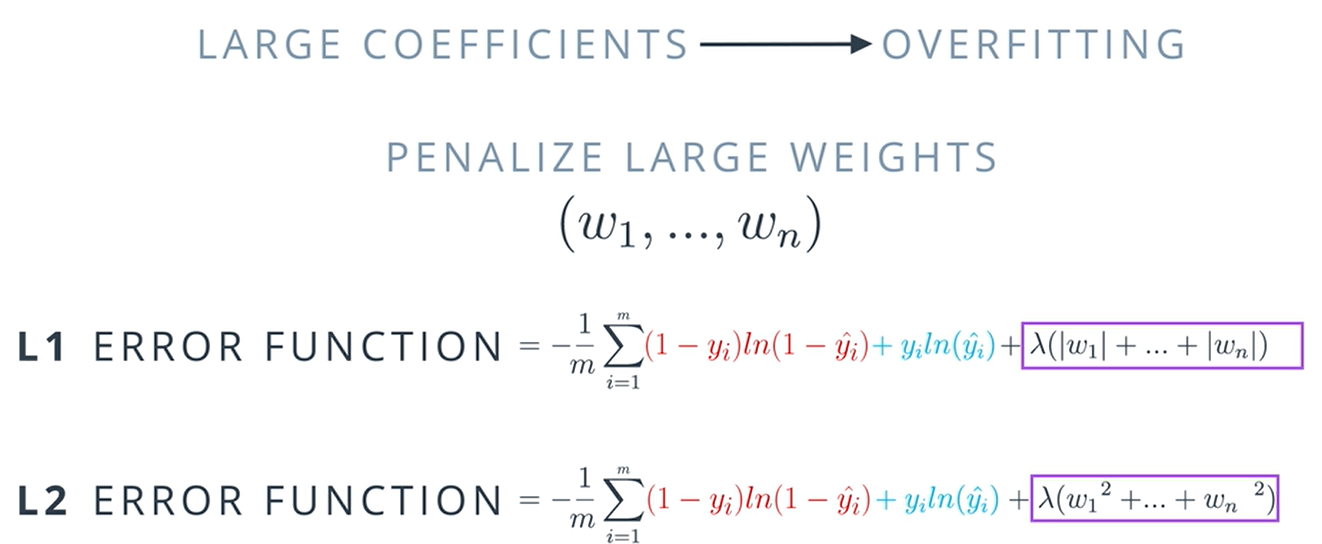
* + 1. Early stopping
       1. Model complexity graph – plotting the graph between the number of epochs and the error and observing the testing and training error. We will stop the epochs where the testing error start increases.

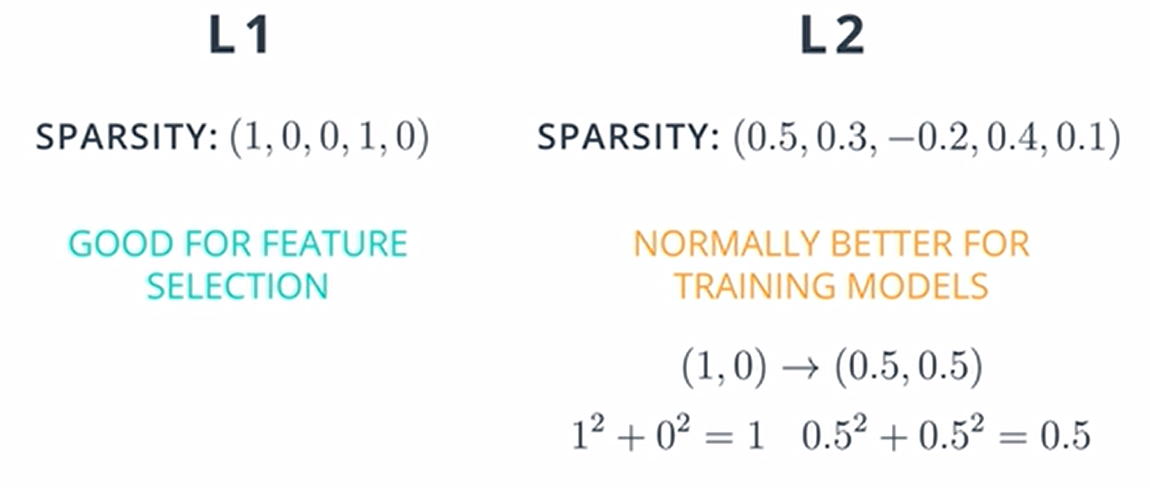




* + 1. Regularization

The technique to penalize the large coefficients is by adding regularization to error function.



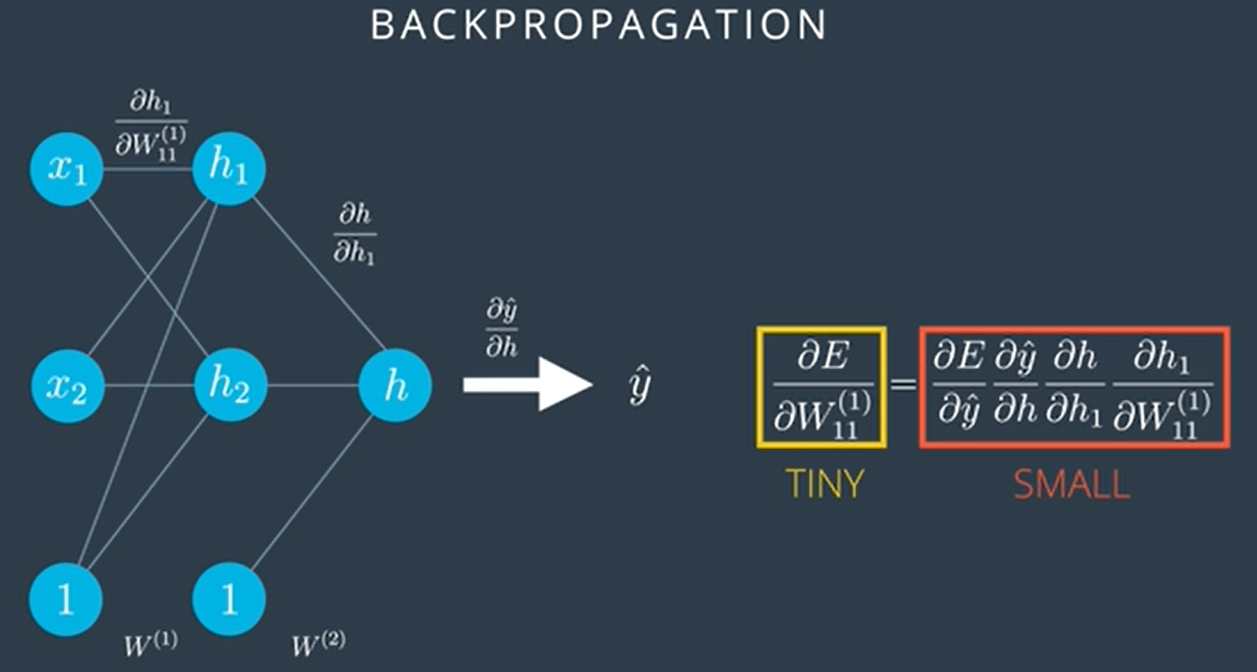


* + 1. Dropout
       1. Sometimes one part of the network may dominate the others. During training, we will randomly turn off some of the nodes, so that others can participate effectively.

If we give the algorithm the probability to turn off some of the node.

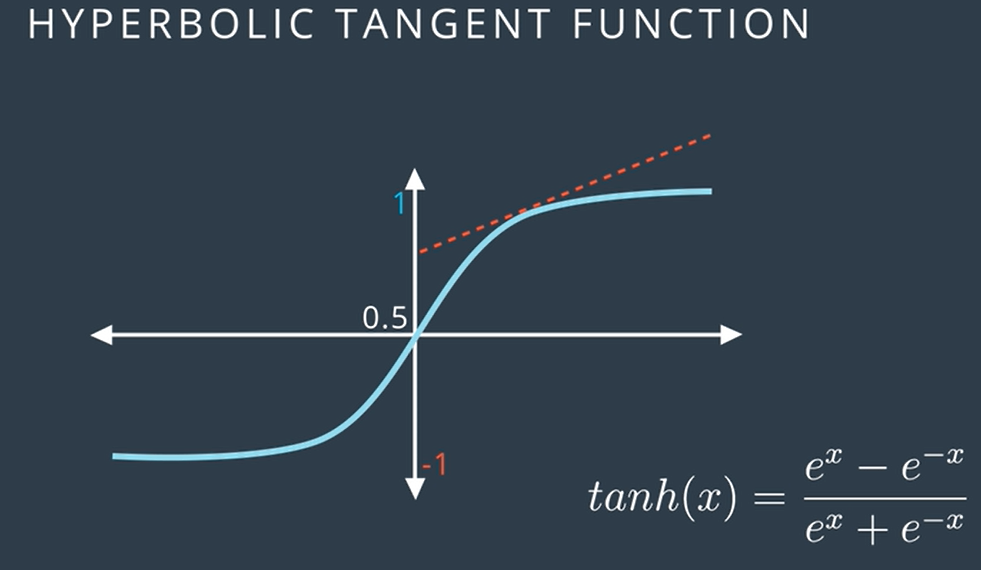
* + 1. Local Minima
       1. We will sometimes stuck in local minima, then gradient descent might not be helpful then we need some other workout.
    2. Random Restart
       1. Random restart is the solution to fix the local minima problem but randomly starting at multiple place and descent down the hill.
    3. Vanishing Gradient

When we apply the sigmoid function, it will give the very tiny gradient which may result in vanishing gradient problem, where it will never reach global minima as the gradient is very small.

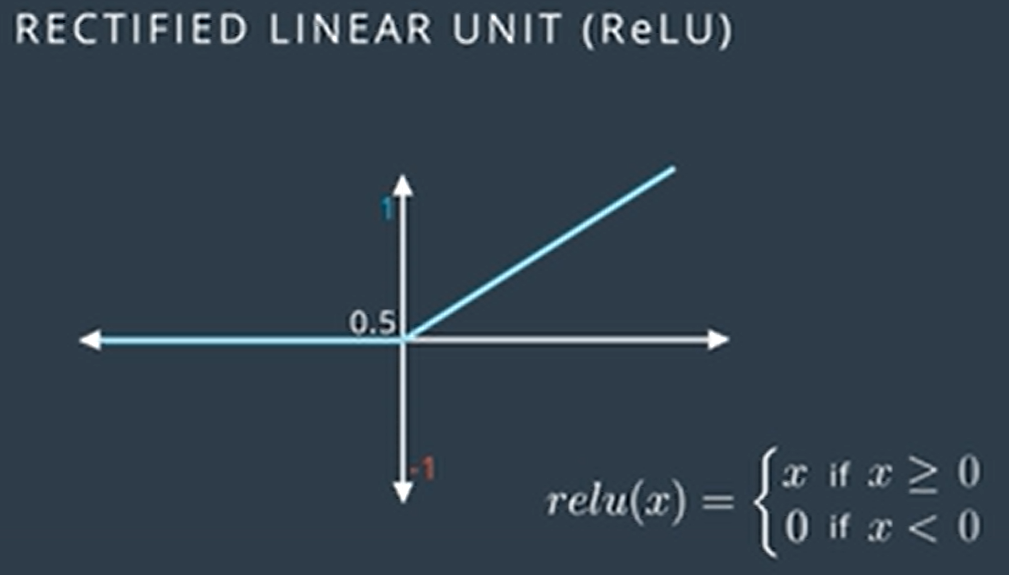


* + 1. Other activation functions

Tanh: Ranges from -1 to 1.



Relu: Rectified linear activation – provides 0 or 1. Best fit for regression problems.



By applying this activation function, we could reduce the problem of vanishing gradient problem.

* + 1. Batch vs stochastic gradient descent

With gradient descent we will apply the whole bunch of data and iterate through bunch of epochs, since it may take time with number of epochs. It is better to take the set of records from the dataset and apply the gradient descent to it. This is nothing but stochastic gradient descent.

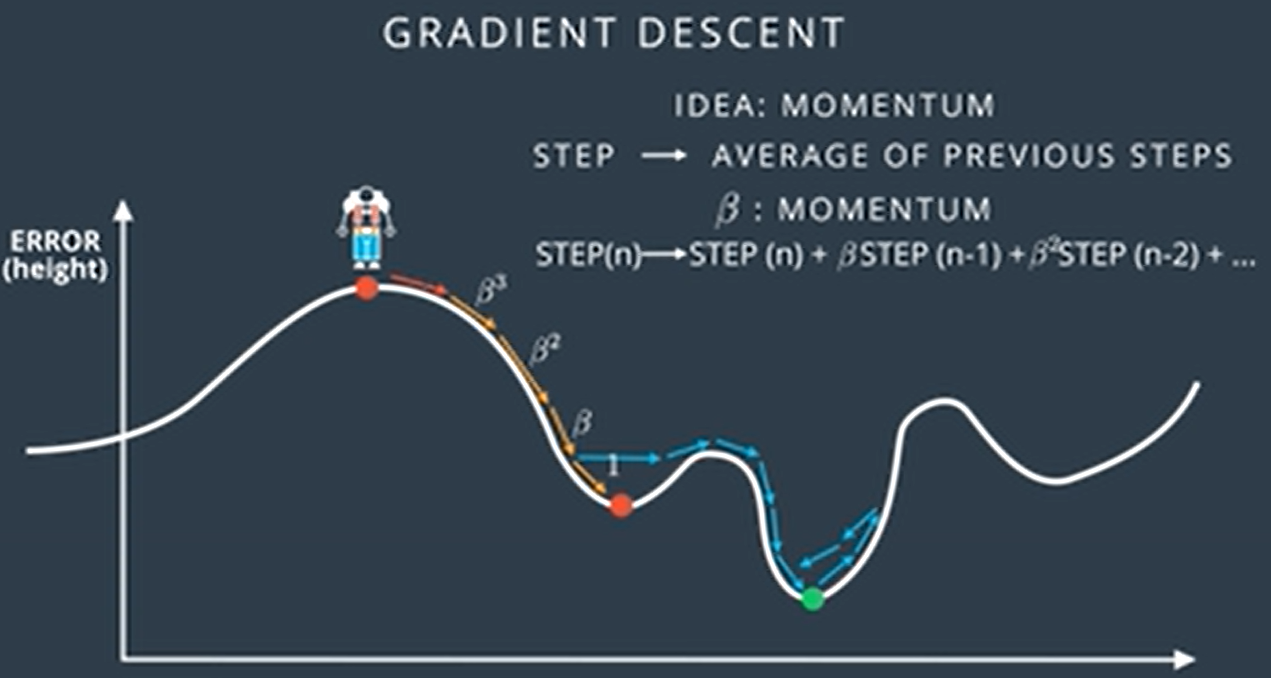
* + 1. Learning rate decay

Its better to take the small learning rate to not miss the global minima.



* + 1. Momentum

It will help us to move from the local minima by taking the value of previous steps and averaging those value.



* + 1. Error functions around the world

[Why Momentum Really Works (distill.pub)](https://distill.pub/2017/momentum/)

[Multivariable Calculus | Khan Academy](https://www.khanacademy.org/math/multivariable-calculus)

[Vector intro for linear algebra (video) | Khan Academy](https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/vectors/v/vector-introduction-linear-algebra)

* 1. GPU workspaces demo
  2. Sentiment analysis
     1. [i am trask](http://iamtrask.github.io/)
     2. Curating a dataset
     3. Trained a neural network
     4. Increased signal and reduced noise in dataset

1. Recurrent neural networks:
   1. RNN history
      1. TDNN (Time delay neural network)
      2. Elman network – has a problem to capture more than 8 steps is very tough
      3. LSTM and GRU is the solution to vanishing gradient problem

There are so many interesting applications, let's look at a few more!

* Are you into gaming and bots? Check out the [DotA 2 bot by Open AI](https://blog.openai.com/dota-2/)
* How about [automatically adding sounds to silent movies?](https://www.youtube.com/watch?time_continue=1&v=0FW99AQmMc8)
* Here is a cool tool for [automatic handwriting generation](http://www.cs.toronto.edu/~graves/handwriting.cgi?text=My+name+is+Luka&style=&bias=0.15&samples=3)
* Amazon's voice to text using high quality speech recognition, [Amazon Lex](https://aws.amazon.com/lex/faqs/).
* Facebook uses RNN and LSTM technologies for [building language models](https://code.facebook.com/posts/1827693967466780/building-an-efficient-neural-language-model-over-a-billion-words/)
* Netflix also uses RNN models - [here is an interesting read](https://arxiv.org/pdf/1511.06939.pdf)
  1. FFNN reminder:
  2. Feed forward process:

Calculating the value of the hidden states: