Build Your Own MLP - XOR

In this practical you will implement a simple neural network (NN). To understand what a NN is we will limit ourselves to a very simple task, namely that of solving the exclusive or (XOR) function.

In this course, we will use Deep Learning library TensorFlow to build neural network powered applications. However, for learning purposes it is better to actually implement everything yourself in plain Python and NumPy.

Setup

Let us import the necessary libraries and configure essential building blocks such as a pseudo-random number generator:

```
import          as
import                      as
from        import

%matplotlib

          ['figure.figsize'] = 12, 6

              = 56

           seed(                  )
```

We have now imported and configured useful helper libraries such as matlibplot for plotting and NumPy - the workhorse of numerical and scientific programming.

Overview

The XOR problem is a toy example to show that linear models can fail at seemingly simple tasks. Consider this (truth-) table:

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

The task of the model is to predict the `y` column given the two input columns `x1` and `x2`. In other words, we must output `1` *iff* (if and only if) either of the inputs is `1` but not otherwise. Obviously,

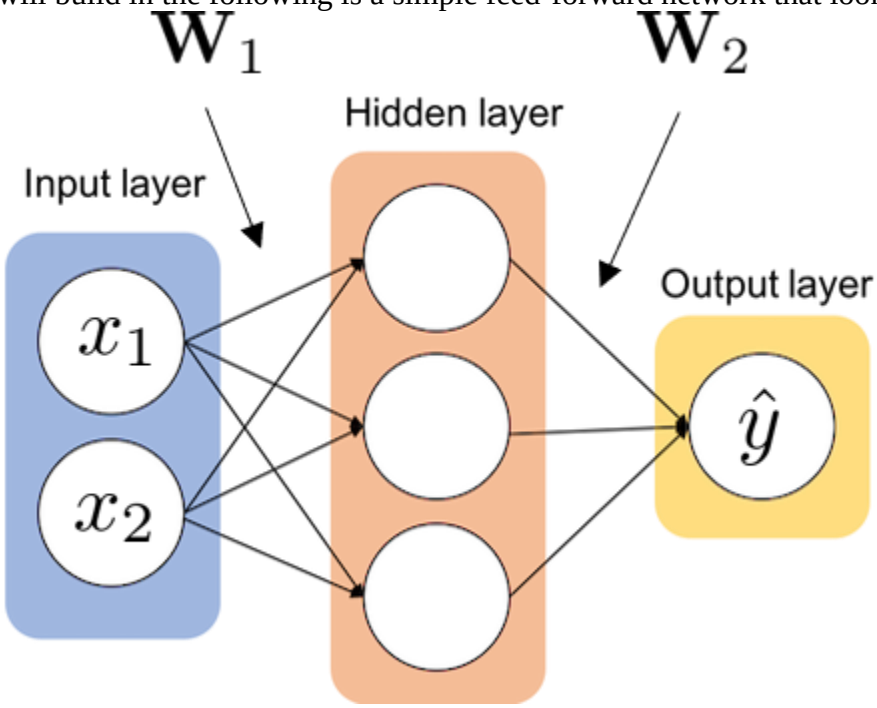
we want to learn this mapping from data, i.e. the 4 data points we have in this table. We can think of this as a classification problem that we can visualize as follows
(*green* represents `1` and *blue* represents `0`):

```
=     array([[0,0], [0,1], [1,0], [1,1]])
=     array([ [0],   [1],   [1],   [0]])

# Colors corresponding to class labels y.
       = ['green' if     == 1 else 'blue' for     in   ]

   =     figure()
   set_figwidth(6)
   set_figheight(6)
   scatter(   [:,0],   [:,1], =200, =          )
   xlabel('x1')
   ylabel('x2')
   show()
```

Plotting the problem makes it clear why linear models will fail to do so - there is simply no straight line in $R2$ that can separate the green from the blue points. The neural network that we will build in the following is a simple feed-forward network that looks like this:



In the input layer we feed the two values $x1$ and $x2$ which we summarize into an input vector $x=[x1,x2]T\in\{0,1\}2$. The hidden layer does some computation and the output layer produces the result $y^\in R$. The associated *weights* with each layer - aka learnable parameters of

the network - are matrices $W_1 \in R^{2\times 3}$ and $W_2 \in R^{3\times 1}$. We can also write this down in a more mathematical way:

$$a = g(h) = g(W_1^T X)$$
$$\hat{y} = W_2^T a$$

Here, the function g is an activation function, which is not explicitly shown in the above image. The choice of the activation function is crucial - we will come back to this later.

Building Blocks

With this preliminaries out of the way, we are ready to implement our neural network and training procedure. For this, we need to implement the following building blocks:

- An **activation function**, i.e. g
- The **forward pass**, i.e. computing $\hat{y}$.
- A **loss function**, i.e. a quantity that measures how far away $\hat{y}$ is from the true value y.

In next practical we will realize the following:

- The **backward pass**, i.e. computing the gradients w.r.t. the loss function
- An **optimizer**, i.e. an algorithm that updates the trainable weights of the network based on the gradients

---

Activation Function

Let's start with an easy task. In the lectures you have seen that in deep neural networks linear layers are commonly followed by a non-linear activation function. Without these neural networks would only be able to approximate affine functions and hence would be a lot less powerful. The perceptron (MLP) algorithm uses a step-function as activation function. However, for various reasons in DL one uses different types of activation functions. One reason is that the activation function should be differentiable and should have a derivative that leads to fast convergence.

In this practical we will be working with the sigmoid function as activation function. It has several appealing properties: it is bounded, it is fully differentiable and has a positive derivative at any point. Furthermore, the sigmoid function maps real-valued inputs to the (0,1) range. This is useful because it allows us to interpret its output as a probability value. The sigmoid function is defined by:

$$\sigma(x) = 1/1+e^{-x}$$

In order to train our neural network we will also need it's derivative which is given by:

$$\partial\sigma(x)/\partial x = \sigma(x) \odot (1-\sigma(x))$$

where $\odot$ stands for element-wise multiplication.

```
            x :
"""
Computes the sigmoid function sigm(input) = 1/(1+exp(-input))
"""
return 1 / (1 +     exp(-  ))

             y :
"""
Computes the derivative of sigmoid funtion. sigmoid(y) * (1.0 - sigmoid(y)).
The way we implemented this requires that the input y is already sigmoided
"""
return   * (1-  )
```

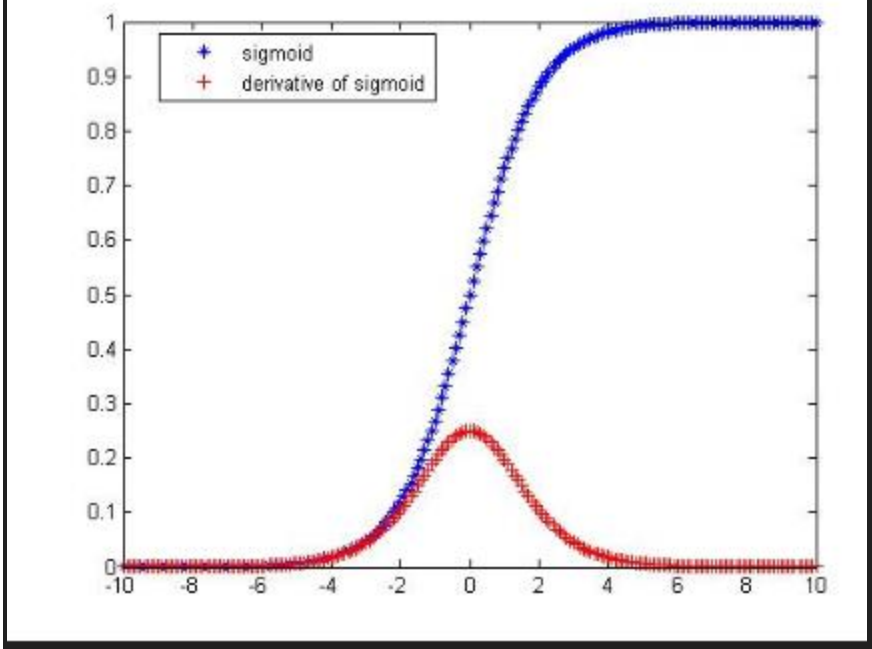
To better understand the sigmoid function and its derivative, let's plot it.

```
=     linspace(-10 , 10 ,      =100)
  = sigmoid(  )
          = sigmoid_(    )

  plot(  ,    ,       ="sigmoid")
  plot(  ,             ,       ="sigmoid prime")
  xlabel("x")
  ylabel("y")
  legend(      ={'size' : 16})
  show()
```

If you need some background info on how we got here check out the full derivation of this

How to Compute the Derivative of a Sigmoid Function (fully worked example)

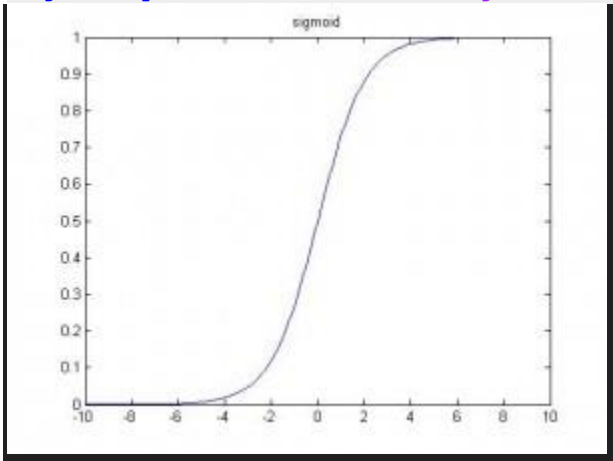


This is a sigmoid function:

$$s(x) = \frac{1}{1+e^{-x}}$$

The sigmoid function looks like this (made with a bit of MATLAB code):

```
x=-10:0.1:10;
s = 1./(1+exp(-x));
figure; plot(x,s); title('sigmoid');
```



Alright, now let's put on our calculus hats...

Here's how you compute the derivative of a sigmoid function

First, let's rewrite the original equation to make it easier to work with.

$$s(x) = \frac{1}{1+e^{-x}} = (1)(1+e^{-x})^{-1} = (1+e^{-x})^{-1}$$

Now we take the derivative:

$$\frac{d}{dx}s(x) = \frac{d}{dx}((1+e^{-x})^{-1})$$

$$\frac{d}{dx}s(x) = -1((1+e^{-x})^{(-1-1)})\frac{d}{dx}(1+e^{-x})$$

$$\frac{d}{dx}s(x) = -1((1+e^{-x})^{(-2)})(\frac{d}{dx}(1) + \frac{d}{dx}(e^{-x}))$$

$$\frac{d}{dx}s(x) = -1((1+e^{-x})^{(-2)})(0 + e^{-x}(\frac{d}{dx}(-x)))$$

$$\frac{d}{dx}s(x) = -1((1+e^{-x})^{(-2)})(e^{-x})(-1)$$

Nice! We computed the derivative of a sigmoid! Okay, let's simplify a bit.

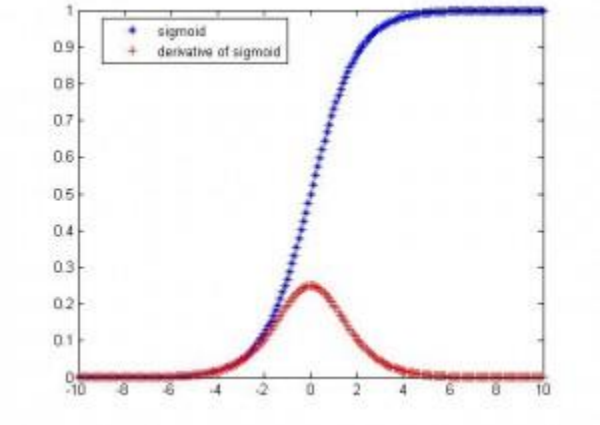
$$\frac{d}{dx}s(x) = ((1+e^{-x})^{(-2)})(e^{-x})$$

$$\frac{d}{dx}s(x) = \frac{1}{(1+e^{-x})^2}(e^{-x})$$

$$\frac{d}{dx}s(x) = \frac{(e^{-x})}{(1+e^{-x})^2}$$

Okay! That looks pretty good to me. Let's quickly plot it and see if it looks reasonable. Again here's some MATLAB code to check:

```
x=-10:0.1:10;  % Test values.
s = 1./(1+exp(-x));  % Sigmoid.
ds = (exp(-x))./((1+exp(-x)).^2); % Derivative of sigmoid.
figure; plot(x,s,'b*'); hold on; plot(x,ds,'r+');
legend('sigmoid', 'derivative-sigmoid','location','best')
```



Looks like a derivative. Good! But wait... there's more!

If you've been reading some of the neural net literature, you've probably come across text that says the derivative of a sigmoid `s(x)` is equal to `s'(x) = s(x)(1-s(x))`.

[note that $\frac{d}{dx}s(x)$ *and s'(x) are the same thing, just different notation.]*

[also note that Andrew Ng writes, f'(z) = f(z)(1 – f(z)), where f(z) is the sigmoid function, which is the exact same thing that we are doing here.]

So your next question should be, is our derivative we calculated earlier equivalent to `s'(x) = s(x)(1-s(x))`?

So, using Andrew Ng's notation...

How does the derivative of a sigmoid f(z) equal f(z)(1-(f(z))?

Swapping with our notation, we can ask the equivalent question:

How does the derivative of a sigmoid s(x) equal s(x)(1-(s(x))?

Okay we left off with...

$$\frac{d}{dx}s(x) = \frac{(e^{-x})}{(1+e^{-x})^2}$$

This part is not intuitive... but let's add and subtract a 1 to the numerator (this does not change the equation).

$$\frac{d}{dx}s(x) = \frac{(e^{-x}+1-1)}{(1+e^{-x})^2}$$

$$\frac{d}{dx}s(x) = \frac{(1+e^{-x}-1)}{(1+e^{-x})^2}$$

$$\frac{d}{dx}s(x) = \frac{(1+e^{-x})}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2}$$

$$= \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})^2}$$

$$= \frac{1}{(1+e^{-x})} - (\frac{1}{(1+e^{-x})})(\frac{1}{(1+e^{-x})}) \quad \text{// factor out a } \frac{1}{(1+e^{-x})}$$

$$= \frac{1}{(1+e^{-x})}(1 - \frac{1}{(1+e^{-x})})$$

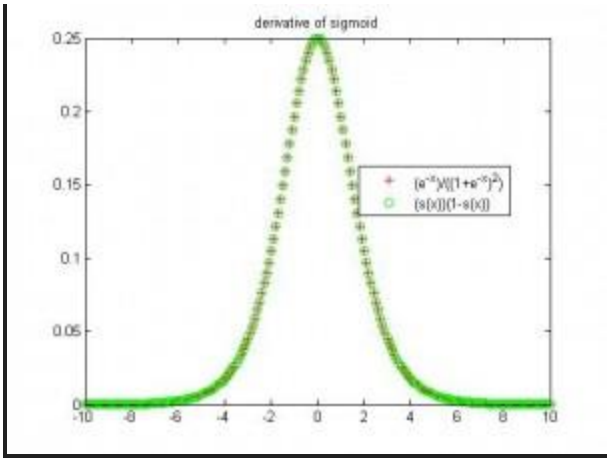
Hmmm.... look at that! There's actually two sigmoid functions there... Recall that the sigmoid function is, $s(x) = \frac{1}{1+e^{-x}}$. Let's replace them with s(x).

$$s'(x) = \frac{d}{dx}s(x) = s(x)(1 - s(x))$$

Just like Prof Ng said... ☺

And for a sanity check, do they both show the same function?

```
x=-10:0.1:10;  % Test values.
s = 1./(1+exp(-x));  % Sigmoid.
ds = (exp(-x))./((1+exp(-x)).^2); % Derivative of sigmoid.
ds1 = s.*(1-s); % Another simpler way to compute the derivative
of a sigmoid.
figure; plot(x,ds,'r+'); hold on; plot(x,ds1, 'go');
legend('(e^{-x})/((1+e^{-x})^2)','(s(x))(1-
s(x))','location','best'); title('derivative of sigmoid')
```



Yes! They perfectly match!

So there you go. Hopefully this satisfies your mathematical curiosity of why the derivative of a sigmoid s(x) is equal to s'(x) = s(x)(1-s(x)).

Forward Pass

Next, we have to implement the "body" of the neural network, in other words, how we compute the output from the inputs. For a feed-forward network like ours, this is actually just a bunch of matrix multiplications followed by sigmoid activations.

As we are usually dealing with a lot of data samples, we don't feed them to the model individually, but organise several of them together into **batches**. We can think of this as stacking the input vectors row-wise into a matrix X. In our case, we only have 4 samples, so X will have shape 4×2. This allows us to rewrite the computations as:

$$\hat{y}=\sigma(X W_1)W_2$$

Now, both $H:=XW_1$ and $A:=\sigma(H)$ have dimensionality 4×3 and each hidden unit is computed by taking a weighted average of the samples' input features. This matrix multiplication is essentially how a dense (aka linear, feed-forward) layer looks like. Let's write that in code.

[ ]

```
        inputs  weights :
    """A simple dense layer."""
    return    matmul(       ,          )
```

With this, we are now ready to define the forward pass of our network. To this end, we have to initialize the weights, so let's do that first.

```
= 2
= 3
= 1

:
# weights for hidden layer, shape: 2x3
=  uniform(  =(  ,  ))
# weights for output layer, shape: 3x1
=  uniform(  =(  ,  ))
return  ,

,  = initialize_weights()
```

Now we use these weights to define the forward pass.

[ ]

```
X :
# Step 1: Calculate weighted average of inputs (output shape: 4x3)
 = dense( ,  )

# Step 2: Calculate the result of the sigmoid activation function (shape: 4x3)
 = sigmoid(  )

# Step 3: Calculate output of neural network (output shape: 4x1)
 = dense(  ,  )

return  ,
```

Loss Function

Next we need to measure how good or bad the output of the network is - in other words, we need to define a loss function. The choice of loss function is crucial for the overall learning success as its gradients provide the supervision signal that drives the optimization. For this toy example, we simply choose the mean squared error (MSE) between the value predicted by the model and the ground-truth label.

$$L=1N\sum_{i=1N}12\|y\hat{}_i-y_i\|_{22}$$

Note that the factor of 12 is chosen for convencience only - you will see why when we compute the gradient in the next section. N denotes the number of samples, which is 4 in our case and i is used to point to an individual sample. Let's write that in code.

[ ]

```
        y_hat  y :
          =        -
        =     mean(0.5 * (            ** 2))
    return            ,
```