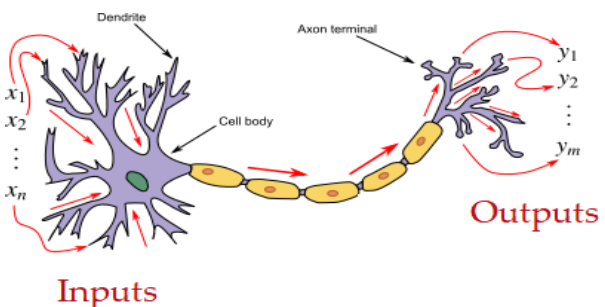
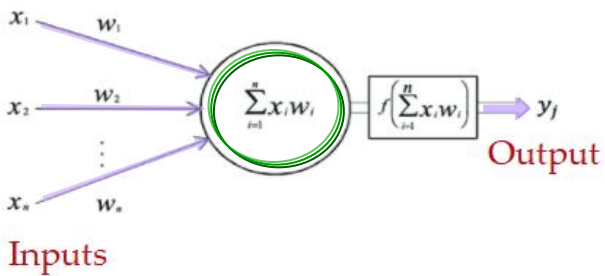
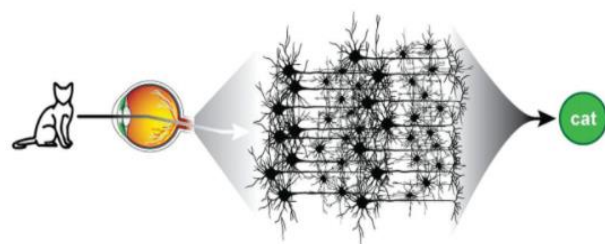
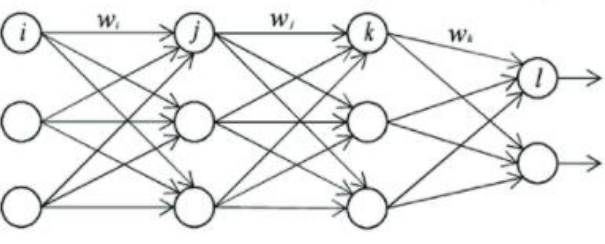


Lab 4 – Artificial Neural Networks

What is an “Artificial Neural Network” (ANN)?

It is a machine learning technique inspired by biological neural networks that constitute the brain. ANNs computationally mimic the training activities that are internally performed inside the human brain. They learn to map inputs into outputs.

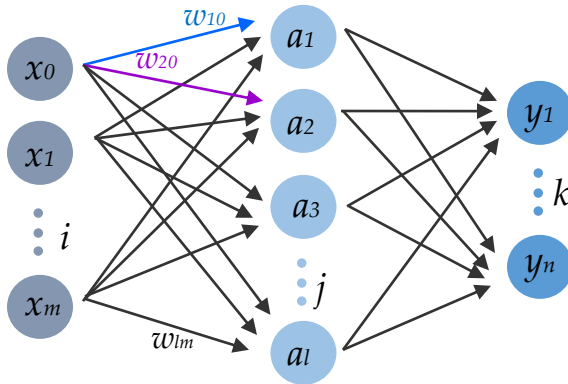
In Biology	In Computing
 <p>A neuron is a nerve cell highly specialized for the processing and transmission of cellular signals.</p>	 <p>An artificial neuron is a mathematical function that takes input and produces output.</p>
 <p>A biological neural network is composed of groups of connected neurons.</p>	 <p>An artificial neural network is composed of layers that contain neurons.</p>

There are several types and architectures of ANNs such as feedforward NNs, recurrent NNs, etc.

ANNs have a lot of various applications such as classification, prediction, pattern recognition, etc.

The structure of a feedforward neural network (FFNN):

Input layer Hidden layer Output layer



In the figure above, it is shown that:

- We have $m+1$ neurons in the input layer (m features in the data + 1 additional column x_0 whose value is 1 for all rows in the data). The input layer can be represented as a vector:

$$\mathbf{X} = \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_m \end{pmatrix}$$

- We have l neurons in the hidden layer and n neurons in the output layer (n outputs). The input neurons are connected to the hidden neurons with certain weights and the hidden neurons are connected to the output neurons with certain weights. **A neural network can have more than one hidden layer.**
- The **weights are the parameters of the model** (the parameters we wish to optimize). The weights connecting layer $(h-1)$ to layer h can be represented as a matrix \mathbf{W}^h of dimensions $(\text{length}(h) \times \text{length}((h-1)))$. For example, the weight matrix connecting the input to the hidden neurons will be:

$$\mathbf{W}^1 = \begin{pmatrix} w_{10} & w_{11} & \dots & w_{1m} \\ w_{20} & w_{21} & \dots & w_{2m} \\ \dots & w_{ji} & \dots & \\ w_{l0} & w_{l1} & \dots & w_{lm} \end{pmatrix}_{l \times m}$$

Note: Each row j in the weight matrix represents all the weights of the inputs to a neuron j in layer h . A weight of x_0 i.e. w_{j0} is considered a bias added to neuron j .

- The **bias** can be added to any hidden or output layer neuron.

The basic steps of using a FFNN:

- Collect the **data** and perform preprocessing if needed.
- Split the data into **training and testing** sets.
- **Train the NN using the training data:**
 1. **Initialize** the weights (parameters) randomly.
 2. Loop for a number of **epochs** or until an **acceptable error** is reached:
 - Loop over **each training example** in the training set:
 - a) Perform **forward propagation**.
 - b) Perform **backpropagation** and update the weights.
 - Calculate the **mean square error**.
- **Test the trained NN on the test set:**
 - Loop over **each example** in the test set:
 - a) Perform **forward propagation**.
 - b) Calculate the **error**.
 - Calculate the overall **accuracy (or error)**.
- Use the trained model on new data by applying forward propagation and getting the output.

Notes:

- The steps above use **vanilla (stochastic) backpropagation** in which an update step is performed after each **single example** (backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output example).
- **Batch backpropagation** can be used where an update is performed using the errors (or sometimes weights in parallel implementations) accumulated from **all (or a batch) of the training examples**.
- Sometimes, we split the data into training, testing and validation sets. The **validation dataset** is a sample of the data used to provide an unbiased evaluation of a model fit on the training dataset while tuning the model hyperparameters (e.g. the number of hidden units). However, sometimes the terms “test” and “validation” are used interchangeably to describe any data held back from training the model.

How forward propagation and backpropagation work:

- **Forward Propagation:**

In forward propagation, we **apply the inputs to the first hidden layer neurons to get their outputs and use them as an input to the next layer and so on until we reach the output layer.**

The **input** to a neuron is the **sum of products** of the input values and weights. The output of a neuron is computed by applying an **activation function** on its input.

➤ **The input to any neuron j in layer h is:**

$net_j^h = \sum_{i=0}^m w_{ji}^h * a_i^{h-1}$ where w_{ji}^h is the weight between neuron j in layer h and neuron i in layer (h-1) and a_i^{h-1} is the value of neuron i in the previous layer (h-1) which is considered as an input to the current neuron j. This means that if we are computing the input to the first hidden layer, a_i^{h-1} will be the actual input features x_i .

➤ **The output of neuron j is:**

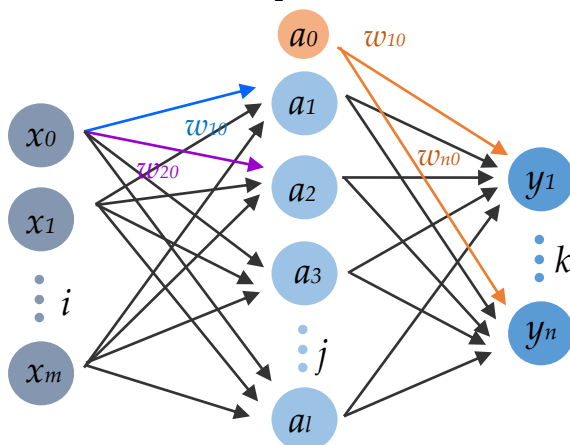
$a_j^h = f(net_j^h)$ where f is the activation function.

➤ **So, from the previous 2 equations,** the value of neuron j which is the input fed to the next layer is calculated by:

$$a_j^h = f(\sum_{i=0}^m (w_{ji}^h * a_i^{h-1}))$$

Remember:

- ✓ We can add a bias unit to any hidden or output layer neuron. For example, if we want to add a bias to all neurons in the hidden layer, we will put weights on a new input x_0 and x_0 is always 1 as shown in the FFNN structure figure. **If we want to add another bias to the output layer, we would add the neuron a_0 (value = 1) and connect it to the output nodes with weights** (considered as the bias) as shown in the figure below. The bias is usually added to allow us to **shift the activation function by adding a constant** because it is not tied to any element of the input.



- ✓ We can use the vector/matrix notation in the above equations:

$$Net^h = W^h * A^{h-1}$$

$$A^h = f(Net^h)$$

Therefore,

$$A^h = f(W^h * A^{h-1})$$

where f (activation function) is applied **elementwise** and when $h = 1$ (first hidden layer), A^{h-1} is the input vector X .

- ✓ We have many different activation functions such as:

Binary Step	$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$
Signum (Bipolar Step)	$f(x) = \begin{cases} -1 & x < 0 \\ 1 & x \geq 0 \end{cases}$
Sigmoid	$f(x) = \frac{1}{1 + e^{-x}}$
Tanh	$f(x) = \frac{2}{1 + e^{-2x}} - 1$
ReLU	$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$
Softmax	$f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}}$ for $i = 1$ to J

- **Backpropagation:**

In **backpropagation**, we perform **backward propagation of errors**. This step is considered the learning step in which we apply **gradient descent** i.e. we calculate the gradient of the error function with respect to the neural network's weights and then update the weights accordingly.

➤ **Recall the parameter update equation using gradient descent:**

$$\mathbf{w}_{ji} = \mathbf{w}_{ji} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_{ji}}$$
 where \mathbf{w}_{ji} is the parameter we wish to update, η is the learning rate, $J(\mathbf{w})$ is the cost (error) function and $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_{ji}}$ is the partial derivative of the cost function with respect to the weight being currently updated \mathbf{w}_{ji} .

In the equation above, the parameters are updated using the error gradient to ensure that **the parameters change in the direction that minimizes the error (cost function)** with a certain learning rate.

Now we know how to update the weights (parameters) in the NN. We just need to get the cost function and its partial derivative. So, the question is “what is $J(\mathbf{w})$?”.

First, let's look at the error at the output layer. The output layer may have multiple output neurons i.e. n neurons, so the error at each neuron k will be the difference between the predicted output \mathbf{a}_k and the actual output \mathbf{y}_k in the data.

➤ **So, the error at neuron k in the output layer is:**

$$E_k^o = \mathbf{a}_k^o - \mathbf{y}_k$$

➤ **And the overall error at the output layer is:**

$$E^o = \frac{1}{2} \sum_{k=1}^n (E_k^o)^2$$

The equation above computes the error at the output layer as the sum of the square errors at each output neuron (the error is divided by 2 for convenience).

➤ **So, we can update the error equation at neuron k in the output layer to:**

$$E_k^o = \frac{1}{2} (\mathbf{a}_k^o - \mathbf{y}_k)^2$$

We can say $J(\mathbf{w}) = E_k^o$ and in order to update any weight \mathbf{w}_{kj}^o directly connected to the output neuron k , we will need to get its partial derivative $\frac{\partial E_k^o}{\partial \mathbf{w}_{kj}^o}$.

However, the error is not a direct function of the weight! The error is a function of the activation output a_k^o , the activation output is a function of the activation input net_k^o and the activation input is a function of the weights. So, it's like a chain rule where:

$$\frac{\partial E_k^o}{\partial w_{kj}^o} = \frac{\partial E_k^o}{\partial a_k^o} * \frac{\partial a_k^o}{\partial net_k^o} * \frac{\partial net_k^o}{\partial w_{kj}^o}$$

➤ Let's calculate each part separately:

$$\frac{\partial E_k^o}{\partial a_k^o} = \frac{\partial \frac{1}{2}(a_k^o - y_k)^2}{\partial a_k^o} = (a_k^o - y_k) * 1 = (a_k^o - y_k) \dots\dots\dots (a)$$

$$\frac{\partial a_k^o}{\partial net_k^o} = \frac{\partial f(net_k^o)}{\partial net_k^o} = a_k^o * (1 - a_k^o) \dots\dots\dots (b)$$

$$\frac{\partial net_k^o}{\partial w_{kj}^o} = \frac{\partial \sum_{i=0}^l w_{ki}^o * a_i^{o-1}}{\partial w_{kj}^o} = a_j^{o-1} \dots\dots\dots (c)$$

- ✓ Equation (a) was obtained directly.
- ✓ To get equation (b), we used the equation $a_k^o = f(net_k^o)$ from the forward propagation step to substitute in the second part. We need the derivative of the activation function f . Since f is usually the sigmoid function, its derivative is known to be:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \rightarrow \frac{\partial \sigma(x)}{\partial x} = \sigma(x) * (1 - \sigma(x))$$

You can find the proof of this derivative at the end of this lab (In "Appendix" section).

- ✓ To get equation (c), we used the equation $net_j^h = \sum_{i=0}^m w_{ji}^h * a_i^{h-1}$ from the forward propagation step to substitute in the second part (after replacing h with o , m with l and so on). In the derivative, the only term that matters is when $i = j$ and the partial derivative will be the a_j^{o-1} .

➤ From (a), (b) & (c):

$$\frac{\partial E_k^o}{\partial w_{kj}^o} = (a_k^o - y_k) * a_k^o * (1 - a_k^o) * a_j^{o-1}$$

➤ Since:

$$w_{kj}^o = w_{kj}^o - \eta \frac{\partial J(w)}{\partial w_{kj}}$$

➤ Therefore, to update any weight connected to the output layer, we use:

$$w_{kj}^o = w_{kj}^o - \eta * (a_k^o - y_k) * a_k^o * (1 - a_k^o) * a_j^{o-1}$$

➤ Or simply:

$$w_{kj}^o = w_{kj}^o - \eta * \delta_k^o * a_j^{o-1}$$

where:

w_{kj}^o is the weight currently being updated that connects neuron k in the output layer with neuron j in the previous layer.

η is the learning rate.

a_k^o is the predicted output at neuron k .

y_k is the actual output at neuron k .

a_j^{o-1} is the value of neuron j at the layer before the output layer.

δ_k^o is the product of the error with the derivative of the activation function.

Unfortunately, if we want to update the weights at **any inner hidden layer** of the network, finding the derivative of the cost function with respect to the weight is even less obvious. It's more complicated because **it depends on the error at all the nodes this weighted connection can lead to**.

➤ So, at hidden layer h where h is $(o-1)$, the partial derivative of the error at some neuron j connected to neuron i in the previous layer will be:

$$\frac{\partial J(w)}{\partial w_{ji}^h} = \frac{\partial E_k^o}{\partial w_{ji}^h} = \left(\sum_{k=1}^n \frac{\partial E_k^o}{\partial a_k^o} * \frac{\partial a_k^o}{\partial net_k^o} * \frac{\partial net_k^o}{\partial a_j^h} \right) * \frac{\partial a_j^h}{\partial net_j^h} * \frac{\partial net_j^h}{\partial w_{ji}^h}$$

As you can see, there are more terms added in the above equation because of the following:

1. The weight w_{ji}^h contributes in calculating the value of neuron a_j^h and the value of this neuron is used to compute the value of all output neurons. That's why we add Σ to take the errors of all output neurons into consideration.
2. Even after taking the sum, net_k^o is not a direct function of w_{ji}^h (unlike the previous case in which net_k^o was directly calculated through the output weights). Here, net_k^o is a function of a_j^h (which is the value of neuron j in the hidden layer) and a_j^h is the activation function applied to net_j^h and finally net_j^h is a function of the weight w_{ji}^h .

➤ Now let's calculate each part of the equation $\frac{\partial E_k^o}{\partial w_{ji}^h}$:

$$\frac{\partial E_k^o}{\partial a_k^o} = (a_k^o - y_k) \text{ from equation (a)}$$

$$\frac{\partial a_k^o}{\partial net_k^o} = a_k^o * (1 - a_k^o) \text{ from equation (b)}$$

$$\frac{\partial net_k^o}{\partial a_j^h} = \frac{\partial \sum_{i=0}^l w_{ki}^o * a_i^{o-1}}{\partial a_j^h} = w_{kj}^o$$

$$\frac{\partial a_j^h}{\partial net_j^h} = a_j^h * (1 - a_j^h) \text{ similar to how equation (b) was computed}$$

$$\frac{\partial net_j^h}{\partial w_{ji}^h} = a_i^{h-1} \text{ similar to how equation (c) was computed}$$

➤ Combining these parts together, we get:

$$\frac{\partial E_k^o}{\partial w_{ji}^h} = (\sum_{k=1}^n (a_k^o - y_k) * (a_k^o * (1 - a_k^o)) * w_{kj}^o) * (a_j^h * (1 - a_j^h)) * a_i^{h-1}$$

$$\frac{\partial E_k^o}{\partial w_{ji}^h} = (\sum_{k=1}^n \delta_k^o * w_{kj}^o) * (a_j^h * (1 - a_j^h)) * a_i^{h-1}$$

➤ So, to update any weight connected from the input to the hidden layer, we use:

$$w_{ji}^h = w_{ji}^h - \eta * (\sum_{k=1}^n \delta_k^o * w_{kj}^o) * (a_j^h * (1 - a_j^h)) * a_i^{h-1}$$

➤ Or simply:

$$w_{ji}^h = w_{ji}^h - \eta * \delta_j^h * a_i^{h-1}$$

and since we have 1 hidden layer, $a_i^{h-1} = x_i$.

➤ Finally, we can generalize the weight update equations at any layer h (assuming we have multiple hidden layers):

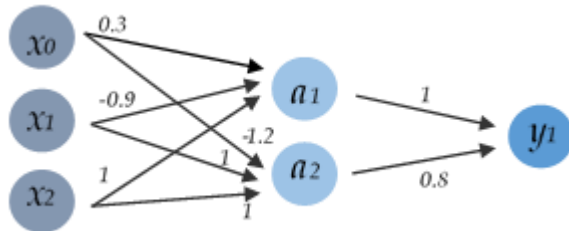
$$w_{ji}^h = w_{ji}^h - \eta * \delta_j^h * a_i^{h-1}$$

and

$$\delta_j^h = \begin{cases} \delta_k^o = (a_k^o - y_k) * \frac{\partial f(net_k^o)}{\partial net_k^o} & \text{If } h \text{ is the output layer } o \\ (\sum_{k=1}^{len(h+1)} \delta_k^{h+1} * w_{kj}^{h+1}) * \frac{\partial f(net_j^h)}{\partial net_j^h} & \text{Otherwise} \end{cases}$$

Example:

We want to train the FFNN below to perform XNOR. How will the training be performed? Perform backpropagation for to 1 training example. (Use sigmoid activation function & learning rate = 0.3)



Sol.:

Training data (XNOR truth table):

x_1	x_2	y_1
0	0	1
0	1	0
1	0	0
1	1	1

Training pseudocode for this NN:

1. Loop over **epochs**:
2. Loop over **training examples**:
 - #feedforward
 - 3. For each hidden layer neuron j :
 4. Calculate and store $a_j^h = f(\sum_{i=0}^m (w_{ji}^h * x_i))$ # $m = 2$ (features)
 - 5. For each output layer neuron k :
 6. Calculate and store $a_k^o = f(\sum_{j=1}^l (w_{kj}^o * a_j^h))$ # $l = 2$ (neurons in h)
 - #backpropagation
 - 7. For each output neuron k :
 8. Calculate and store $\delta_k^o = (a_k^o - y_k) * a_k^o * (1 - a_k^o)$
 - 9. For each hidden neuron j :
 10. Calculate and store $\delta_j^h = (\sum_{k=1}^n \delta_k^o * w_{kj}^o) * a_j^h * (1 - a_j^h)$
 - 11. For each weight w_{kj}^o going to the output layer:
 12. Update $w_{kj}^o = w_{kj}^o - \eta * \delta_k^o * a_j^h$
 - 13. For each weight w_{ji}^h going to the hidden layer:
 14. Update $w_{ji}^h = w_{ji}^h - \eta * \delta_j^h * x_i$

Now, we will perform backpropagation for to 1 training example. Let's take the second one where $x_1 = 0$, $x_2 = 1$, $y_1 = 0$ and x_0 is always 1. So, we have:

$$X = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$W^h = \begin{bmatrix} 0.3 & -0.9 & 1 \\ -1.2 & 1 & 1 \end{bmatrix}$$

$$W^o = [1 \quad 0.8]$$

Starting from steps 3 & 4: (Feedforward)

$$a_1^h = f(\sum_{i=0}^m (w_{1i}^h * x_i)) = f(0.3*1 - 0.9*0 + 1*1) = f(1.3) = \frac{1}{1 + e^{-1.3}} = 0.786$$

$$a_2^h = f(\sum_{i=0}^m (w_{2i}^h * x_i)) = f(-1.2*1 + 1*0 + 1*1) = f(-0.2) = \frac{1}{1 + e^{0.2}} = 0.45$$

Steps 5 & 6: (Feedforward)

$$a_1^o = f(\sum_{j=1}^l (w_{1j}^o * a_j^h)) = f(1*0.786 + 0.8*0.45) = f(1.146) = 0.759$$

Steps 7 & 8: (Backpropagation)

$$\delta_1^o = (a_1^o - y_1) * a_1^o * (1 - a_1^o) = (0.759 - 0) * 0.759 * (1 - 0.759) = 0.139$$

Steps 9 & 10: (Backpropagation)

$$\delta_1^h = (\sum_{k=1}^n \delta_k^o * w_{k1}^o) * a_1^h * (1 - a_1^h) = (0.139*1)*0.786*(1 - 0.786) = 0.023$$

$$\delta_2^h = (\sum_{k=1}^n \delta_k^o * w_{k2}^o) * a_2^h * (1 - a_2^h) = (0.139*0.8)*0.45*(1 - 0.45) = 0.028$$

Steps 11 & 12: (Weight update)

$$w_{11}^o = w_{11}^o - \eta * \delta_1^o * a_1^h = 1 - 0.3 * 0.139 * 0.786 = 0.967$$

$$w_{12}^o = w_{12}^o - \eta * \delta_1^o * a_2^h = 0.8 - 0.3 * 0.139 * 0.45 = 0.78$$

Steps 13 & 14: (Weight update)

$$w_{10}^h = w_{10}^h - \eta * \delta_1^h * x_0 = 0.3 - 0.3 * 0.023 * 1 = 0.29$$

$$w_{11}^h = w_{11}^h - \eta * \delta_1^h * x_1 = -0.9 - 0.3 * 0.023 * 0 = -0.9$$

$$w_{12}^h = w_{12}^h - \eta * \delta_1^h * x_2 = 1 - 0.3 * 0.023 * 1 = 0.99$$

$$w_{20}^h = w_{20}^h - \eta * \delta_2^h * x_0 = -1.2 - 0.3 * 0.028 * 1 = -1.2$$

$$w_{21}^h = w_{21}^h - \eta * \delta_2^h * x_1 = 1 - 0.3 * 0.028 * 0 = 1$$

$$w_{22}^h = w_{22}^h - \eta * \delta_2^h * x_2 = 1 - 0.3 * 0.028 * 1 = 0.99$$

Appendix:

The derivative of the sigmoid function:

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

$$\begin{aligned}\frac{\partial \sigma(x)}{\partial x} &= \frac{\partial}{\partial x} \left(\frac{1}{1 + e^{-x}} \right) = \frac{\partial}{\partial x} (1 + e^{-x})^{-1} \\&= - (1 + e^{-x})^{-2} (-e^{-x}) \\&= \frac{e^{-x}}{(1 + e^{-x})^2} \\&= \frac{1}{1 + e^{-x}} * \frac{e^{-x}}{1 + e^{-x}} \\&= \sigma(x) * \frac{1 + e^{-x} - 1}{1 + e^{-x}} \\&= \sigma(x) * \frac{1 + e^{-x}}{1 + e^{-x}} - \frac{1}{1 + e^{-x}} \\&= \sigma(x) * (1 - \sigma(x))\end{aligned}$$

Links mentioned in the lab video:

<https://playground.tensorflow.org/>

<https://www.tensorflow.org/tutorials/keras/classification>